Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

2.º Trabalho Laboratorial
Rede de Computadores

Universidade do Porto

Faculdade de Engenharia

FEUP

António Garcez ei10028@fe.up.pt
Joaquim Barros ei10087@fe.up.pt
Nelson Mendes ei10044@fe.up.pt

Porto, 20 de Dezembro de 2012

Sumário

Este 2.º trabalho prático foi desenvolvido para a unidade curricular Redes de Computadores em duas partes: a 1.ª parte com o propósito de implementar um cliente FTP de forma a que seja possível descarregar ficheiros de servidores FTP e a 2.ª parte com o objetivo principal de estudo de uma rede de computadores, fazendo a sua configuração e consequente análise dos resultados obtidos.

O cliente FTP foi implementado com sucesso e todas as experiências foram realizadas e analisadas devidamente.

1. Introdução

O projeto corrente tem como principal intuito a realização de duas partes: a implementação de uma aplicação para download via FTP e o estudo de uma rede de computadores.

Na 1.ª parte o programa construído deverá transferir ficheiros de um servidor para um cliente FTP através da *Internet*. Para que tal seja possível, é necessário conhecer o protocolo FTP, assim como as mensagens necessárias para a sua concretização.

Para a 2.ª parte, foi necessário montar e configurar uma rede que permitisse ligar os computadores de uma determinada forma e estudar as consequências resultantes dessas configurações.

Pretende-se com este relatório demonstrar todas as considerações necessárias para que seja possível a criação do cliente FTP assim como os passos a realizar para preparar as várias redes de computadores apresentadas nas experiências. É a partir delas que obtemos as conclusões pretendidas.

2. Aplicação para *download*

2.1 Arquitetura

Para a implementação da aplicação de download foi necessário perceber como a troca de mensagens se processa para que se faça autenticação e posterior pedido do ficheiro a transferir. De acordo com o *RFC959* e consultando o *Beej's Guide to Network Programming* percebe-se que o programa deverá seguir a seguinte sequência:

1. Validar e guardar dados introduzidos pelo utilizador respeitantes ao *host*, *path*, *username* e *password*. Estes dois últimos, caso não existam, deverão ser definidos como *anonymous*.
2. Obter endereço IP em função do *host* com a função *gethostbyname(host).*
3. Abrir *socket* para conexão TCP/IP com o IP obtido e a porta para FTP (21).
4. Obter mensagem de boas vindas.
5. Enviar “USER *username*” e esperar resposta positiva a pedir *password*.
6. Enviar “PASS *password*” e esperar resposta positiva da autenticação.
7. Enviar “PASV” e esperar resposta positiva, contendo a porta à qual se vai ligar.

8 Abrir novo *socket* para conexão de dados com o IP obtido anteriormente e a porta obtida no passo 7 que resulta da soma *V1*256 + V2*.
9 Fazer pedido de *Retrieve* com "RETR *path*" para o *socket* de controlo e obter resposta positiva.
10 Fazer *read's* sucessivos no *socket* de dados para obter o ficheiro pedido.
11 Receber respostas finais a indicar a conclusão com sucesso da transferência de dados.

Daqui se retiram algumas conclusões acerca das funções necessárias para a implementação correta da aplicação:

1 *void startConnection()* - função para abrir o *socket* de controlo a partir do *host* introduzido pelo utilizador.
2 *int getMessage(char * message)* - função para obter mensagens do *socket* de controlo. Estas são gravadas em *message* e o código respetivo é o valor de retorno.
3 *void sendStartData()* - função para iniciar a autenticação no servidor *FTP*. É aqui que são enviados o *username* e a *password* do utilizador. Caso não existam, são enviados *anonymous* nos dois campos anteriores. Por último, é enviado um pedido para entrar em modo *passive* e aberto um novo *socket* com a porta retornada deste último pedido.
4 *void receiveData()* - função para obter o ficheiro pretendido. É feito um pedido de *RETRIEVE*. Após confirmação positiva, é iniciada a transferência no *socket* de dados.
5 *int main()* - função principal que chama sequencialmente as funções anteriormente apresentadas e que fecha os *socket's* no final da transferência.

Conjugando todas estas funções consegue-se uma aplicação de download bastante eficaz.

2.2 *Downloads* bem sucedidos

Foram testados diversos links, incluindo:

- ftp://ftp.dei.uc.pt/pub/linux/xubuntu/releases/12.10/release/xubuntu-12.10-desktop-i386.iso (sem autenticação)
- ftp://ftp.up.pt/pub/robots.txt (sem autenticação)
- ftp://tom.fe.up.pt (neste foi utilizada autenticação na área pessoal)

Em todos eles verificou-se que a receção era bem sucedida e percebeu-se que a mensagem de sucesso de envio, quando o ficheiro é muito pequeno, pode aparecer mesmo antes de o obter através dos sockets. Isto levou-nos à implementação de processos que garantissem que a mensagem fosse apresentada antes ou depois da receção.

3. Configuração de redes e análise

Experiência 1

Nesta primeira experiência começou-se por retirar a ligação dos computadores à rede do laboratório. De seguida, fez-se a configuração destes através do comando *ifconfig*. No final, usamos o comando "ping" para verificar a conectividade dos dois computadores configurados: *tuxy1* e *tuxy4*.

Os pacotes ARP (*Address Resolution Protocol*) são pacotes utilizados para encontrar um endereço da camada de ligação (*MAC*) a partir do endereço da camada de rede (*IP*). *MAC* (*Media Access Control*) é um endereço físico associado à interface de comunicação, que liga um dispositivo à rede enquanto que o endereço *IP* é a identificação de um determinado dispositivo numa rede. O endereço *MAC* dos pacotes *ARP* é *00:21:5a:c7:64:8e* e os endereços *IP* de origem e de destino são, respetivamente, *172.16.40.1* e *172.16.40.254*.

Os pacotes gerados pelo comando *ping* têm como endereço *MAC 00:c0:df:08:d5:b3* e *IP*s iguais aos anteriores.

Para determinar qual o tipo de pacote recebido temos de verificar os octetos número 25 e 26 da *ethernet frame*. Através desses octetos vemos se a *frame* corresponde a *ARP*, *IP* ou *ICMP*. Também através desses octetos conseguimos é possível saber o tamanho da *frame*.

A loopback interface é uma interface de rede à qual só a própria máquina tem acesso. Possui o endereço de *IP* fixo *127.0.0.1*, no caso do *IPv4*, ou *::1*, no caso do *IPv6*. Esta interface é útil na realização de testes à stack TCP/IP mesmo que o computador não esteja ligado a nenhuma rede. Serve também para aceder mais facilmente a serviços de rede instalados na própria máquina, como por exemplo *webservers*.

Experiência 2

Depois de a primeira experiência ter sido concluída com sucesso, o objectivo da segunda experiência é a implementação de duas *virtual LANs* (*VLANs*) num *switch*. Nesta experiencia a *vlan40* vai ser constituída pelo *tuxy1* e pelo *tuxy4* enquanto a *vlan41* vai ser constituída única e exclusivamente pelo *tuxy2*.

Antes de implementarmos as *VLAN*s considerámos de boa prática fazer *reset* às configurações do *switch* e às configurações de rede dos *tuxy*'s. Para a configuração das *VLAN*s foram utilizados os seguintes comandos:

1. *configure terminal*
2. *vlan 40* (*41* para a segunda vlan)
3. *interface fastethernet 0/P* (P - porta do switch)
4. *switchport mode access*
5. switchport *access vlan 40*
6. *end*

Nesta segunda experiência como existem dois computadores na mesma *VLAN*, e um outro computador no mesmo *switch* mas noutra *VLAN*, podemos concluir que existem dois domínios de *broadcast*. A partir dos *logs* que capturámos no *tuxy1* e no *tuxy2* concluímos que

se fizermos *ping -b 172.16.40.255,* o *tuxy1* obtém uma resposta que provem do *tuxy4*. Isto só é possível pelo facto de estarem na mesma *VLAN*. Mas, se no *tuxy2* colocarmos o comando *ping -b 172.16.41.255,* este não obtém qualquer resposta porque é o único computador da segunda *VLAN*.

Experiência 3

Nesta experiência o objetivo principal é a transformação do *tuxy4* num *router*, permitindo que haja ligação entre as duas *VLANs* anteriormente criadas. Para a realização da experiência foi necessário configurar os endereços IP dos computadores, as suas rotas e as tabelas de *forwarding*. Os comandos mais utilizados para este fim foram: *route -n*, *route add*, *ifconfig* e o comando *ping* (este último para testes).

O passo fundamental nesta experiência foi a atribuição de um endereço IP ao eth1 do *tuxy4* que iria ser ligado ao *switch*, na *VLAN41*. De seguida, para que seja possivel a comunicação entre as duas *VLANs* foi necessário configurar as rotas nos computadores *tuxy1* e *tuxy2*. As rotas adicionadas às tabelas destes computadores passavam a indicar que para qualquer pedido o *default gateway* (endereço através do qual deverão ser enviados os pacotes com determinado destino) era o *172.16.40.254* no caso do *tuxy1* e *172.16.41.253* no caso do *tuxy2*. Para além disso foi necessário configurar as opções *ip_forward* e *ignore_broadcasts* no *tuxy4 c*om os seguintes comandos:

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

Configurando as tabelas de redirecionamento, verifica-se que por cada entrada são apresentados o *IP* de destino, o *IP* de *gateway*, a máscara do endereço, flags associadas, o custo do caminho, variáveis de estado (*ref* e *use*) e a interface utilizada. As mensagens *ARP* geradas logo após a ligação foram as seguintes:

- Who has 172.16.40.1? Tell 172.16.40.254 (Pedido do tuxy4)
- 172.16.40.1 is at 00:c0:df:08:d5:b3
- Who has 172.16.41.1? Tell 172.16.41.253 (Pedido do tuxy4)
- 172.16.41.1 is at 00:12:3f:4d:eb:fb
- Who has 172.16.41.253? Tell 172.16.41.1 (Pedido do tuxy2)
- 172.16.41.253 is at 00:c0:df:25:1a:f4
- Who has 172.16.40.254? Tell 172.16.40.1 (Pedido do tuxy1)
- 172.16.40.254 is at 00:21:5a:c7:64:8e

Estas mensagens aparecem porque os computadores, para comunicarem na rede *ethernet*, precisam de conhecer quais os endereços *MAC* desses mesmos dispositivos. As mensagens *ARP* são enviadas na esperança de obter uma resposta com o endereço *MAC* pretendido.

São apresentados, de seguida, alguns pacotes *ICMP* encontrados nos *logs* da experiência:

- 15 20.622984 172.16.40.1 172.16.40.254 ICMP 98 Echo (ping) request id=0xce0f, seq=1/256, ttl=64

- 16 20.623141 172.16.40.254 172.16.40.1 ICMP 98 Echo (ping) reply id=0xce0f, seq=1/256, ttl=64
- 43 41.386994 172.16.40.1 172.16.41.253 ICMP 98 Echo (ping) request id=0xdb0f, seq=1/256, ttl=64
- 44 41.387148 172.16.41.253 172.16.40.1 ICMP 98 Echo (ping) reply id=0xdb0f, seq=1/256, ttl=64
- 94 146.923228 172.16.40.1 172.16.41.1 ICMP 98 Echo (ping) request id=0x0514, seq=1/256, ttl=64
- 95 146.925034 172.16.41.1 172.16.40.1 ICMP 98 Echo (ping) reply id=0x0514, seq=1/256, ttl=63

Os endereços IP encontrados apresentam-se acima.

Os pacotes *ICMP* são utlizados para controlo de erros numa rede. Neste caso aparecem no *log* devido aos pedidos feitos pelo comando *ping* no *tuxy1*. Em cada par encontra-se um pedido e um resposta. Os *MAC addresses* dos computadores *tuxy1*, *tuxy4* e *tuxy2* são, respetivamente, *00:c0:df:08:d5:b3*, *00:21:5a:c7:64:8e* e *00:12:3f:4d:eb:fb*.

Experiência 4

A partir da configuração conseguida anteriormente, era necessário fazer uma extensão à *VLAN41*, que consiste em adicionar um *router* para comunicação à *Internet.* Esta configuração teria de ser feita inicialmente sem *NAT* e só depois com *NAT*.

Para configurar o *router* de forma a que seja possível a conexão com o exterior foram realizados os seguintes passos:

```
1  interface gigabitethernet 0/0
2  ip address 172.16.41.254 255.255.255.0
3  no shutdown
4  exit
5  interface gigabitethernet 0/1
6  ip address 172.16.1.49 255.255.255.0
7  no shutdown
8  exit
```

Para além disso foi necessário redefinir as rotas para que os vários *tuxys* pudessem ter ligação ao *router* comercial.

Fazendo *ping* do *tuxy1* para o *tuxy4* o caminho seguido pelos pacotes é do *tuxy1* para o *tuxy4*. Se mandarmos um *ping* a partir do *tuxy1* para o *tuxy2*, o trajeto dos pacotes divide-se em dois caminhos: no primeiro, o pacote vai desde o *tuxy1* até ao *tuxy4*; no segundo, devido ao reencaminhamento, o pacote vai desde o *tuxy4* até ao *tuxy2*. No caso em que o *ping* é feito do *tuxy1* para o *router* comercial, o caminho dos dados é similar ao que foi referido anteriormente, só que desta vez o *router* comercial é o local de destino.

Quando o *ping* é feito do *tuxy1* para o endereço do *router* do laboratório, o trajeto seguido pelos pacotes é o mesmo que o trajeto seguido até ao router, adicionando ainda o caminho até ao router do laboratório. No entanto, como os endereços que estamos a utilizar

fazem parte da gama de endereços privados, os pacotes não contêm informação suficiente para se saber quem foi o emissor que fez o pedido. Por isso, na experiência realizada, verificou-se que não há qualquer conectividade nos endereços externos à rede configurada. Só ativando corretamente a funcionalidade de *NAT* é que é possível ter acesso a toda a rede externa. O pacote, antes de ser enviado pela rede externa, terá de ser alterado. O endereço *IP* interno é substituído pelo endereço de acesso à rede criada, ou seja, *172.16.1.49* adicionando o número de porta correspondente ao computador que enviou. Posteriormente, quando chegar a resposta ao pedido anterior, o destino do pacote recebido no *router* é substituído pelo endereço *IP* interno do computador ao qual vai ser feita a entrega. Assim, resumidamente, o *NAT* permite que computadores de redes privadas possam aceder a redes externas sem necessitarem de um endereço público para cada um. Um endereço, tipicamente fornecido pelo ISP, é suficiente para que um conjunto de computadores possa comunicar com o exterior.

Para a configuração do *NAT* no *router* comercial do laboratório foram introduzidos os seguintes comandos no seu terminal:

1 *ip nat pool ovrld 172.16.1.49 172.16.1.49 prefix 24*
2 *ip nat inside source list 1 pool ovrld overload*
3 *access-list 1 permit 172.16.40.0 0.0.0.7*
4 *access-list 1 permit 172.16.41.0 0.0.0.7*

Para a configuração das rotas estáticas foram introduzidos os seguintes comandos:

1 *ip route 0.0.0.0 0.0.0.0 172.16.1.254*
2 *ip route 172.16.40.0 255.255.255.0 172.16.41.253*

Experiência 5

Esta experiência tem como principal objetivo a configuração do *DNS* (*Domain Name System* - Sistema de Nomes de Domínios), que é um sistema que obtém endereços *IP* a partir do nome do *host* e vice-versa, permitindo assim que o utilizador não precise de decorar os endereços das máquinas às quais se pretende conectar. Para a concretização dos objetivos delineados configuramos o servidor DNS em cada um dos computadores da nossa rede com o endereço *172.16.1.2 - lixa.netlab.fe.up.pt*. Para isso, foi necessário editar o ficheiro */etc/resolv.conf*, adicionando o seguinte conteúdo:

search netlab.fe.up.pt
nameserver 172.16.1.2

Os pacotes trocados pelo serviço de *DNS* aquando da captura do *log* no *Wireshark* são os seguintes:

- *18 27.730402 172.16.10.1 172.16.1.2 DNS 69 Standard query 0xa3d0 A google.pt*
- *19 27.732435 172.16.1.2 172.16.10.1 DNS 263 Standard query response 0xa3d0 A 173.194.41.215 A 173.194.41.216 A 173.194.41.223*

Verifica-se nestes pacotes o pedido *DNS* para se obter o endereço da máquina com *hostname google.pt*, tendo obtido o *IP 173.194.41.215*. O tipo de pacotes que são trocados

pelo *DNS* são do tipo *IP*, que contêm a origem do pedido e o *hostname* do qual se quer obter o *IP* ou o *IP* caso se pretenda saber o *hostname* correspondente.

Experiência 6

O objetivo da experiência 6 era perceber se o cliente *FTP* tinha sido construído corretamente e se era possível descarregar dados através da rede criada ao longo das experiências anteriores. Para isso foi necessário executar o cliente *FTP* no *tuxy1* e no *tuxy2* e esperar que o ficheiro fosse transferido com sucesso, enquanto decorria a captura no *Wireshark*.

Na aplicação *FTP* construída são criadas duas ligações *TCP*: na primeira, são enviadas as informações de controlo para que seja possível a obtenção dos dados pretendidos; na segunda ligação obtêm-se os pacotes que permitem resconstruír os dados pedidos.

As ligações *TCP* são divididas em três fases: "*connection establishment*", em que é estabelecida a ligação através da troca de pacotes com dados acerca da ligação, "*transfer phase*", na qual é transferida toda a informação e "*connection termination*", que fecha os circuitos virtuais estabelecidos e liberta todos os recursos que foram alocados.

O mecanismo de *ARQ* associado ao *TCP* permite assegurar a entrega segura de dados. Uma falha faz com que haja uma retransmissão de dados. Além disso o *TCP* usa os mecanismos que o *ARQ* disponibiliza para evitar situações de congestionamento da rede. Os campos relevantes para este mecansmo são o *"sequence number", "acknowledgement number", "window size" e "check sum"* (este ultimo é usado para correção de erros). A partir dos *logs* obtidos consegue-se visualizar facilmente os números de sequência e a troca de mensagens geradas de forma a que o número de pacotes em processamento não exceda o tamanho máximo da janela definida, que estes não cheguem fora de ordem, que não apareçam pacotes duplicados e que haja correção de erros. No decurso da transferência verificou-se que a velocidade aumentou até atingir um ponto em que a janela foi excedida. Nesse momento, para evitar congestionamento a velocidade diminuiu de forma a que a situação fosse resolvida, estando de acordo com o comportamento do mecanismo de controlo de congestionamento do *TCP*.

Posteriormente, foi iniciada uma nova transferência no *tuxy2* enquanto a transferência no *tuxy1* decorria. Verificou-se que a velocidade no *tuxy1* diminui, o que mostra que a largura de banda é divida pelos dois computadores. (Nota: A captura no *tuxy2* foi iniciada depois da iniciada no *tuxy1*. No entanto percebe-se a variação no *tuxy1* por volta dos 30s e no *tuxy2* por volta dos 33s.)

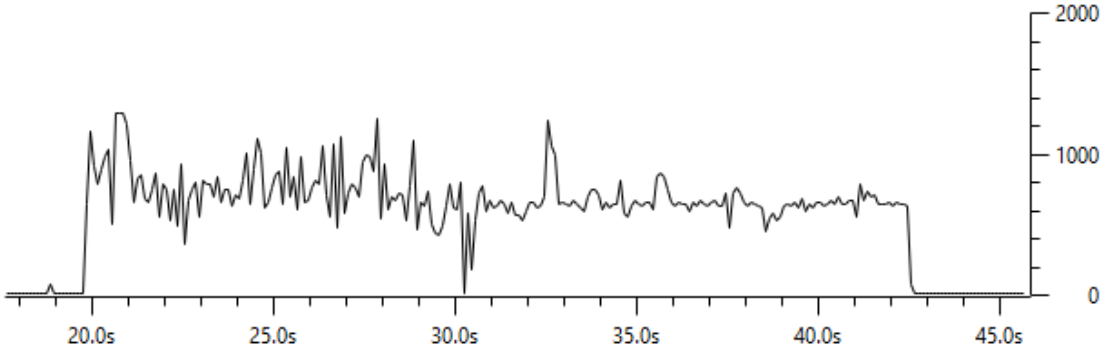

Imagem 1: Transferência no *tuxy1*

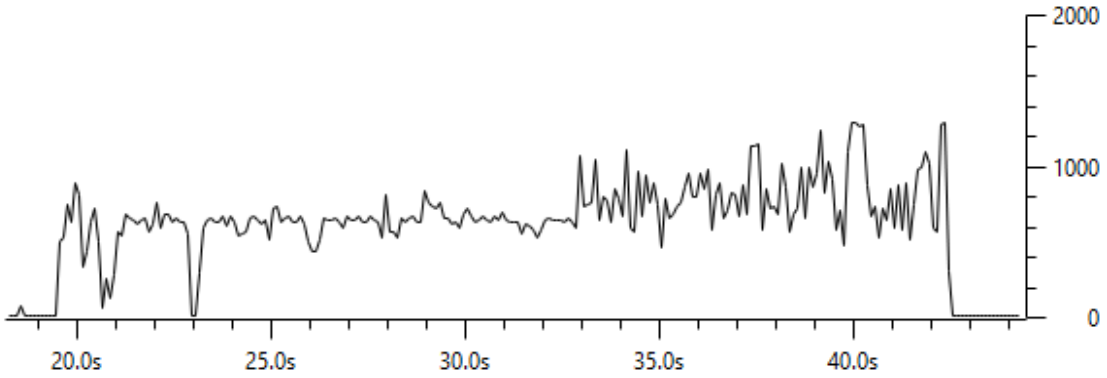

Imagem 2: Transferência no *tuxy2*

4. Conclusões

Todas as experiências propostas para este trabalho laboratorial foram realizadas com sucesso. No final obteve-se uma rede através da qual era possível a comunicação interna entre computadores e a comunicação externa, permitindo, por exemplo, o acesso a páginas *web*.

Através da configuração da rede no laboratório foi possível perceber alguns factos teóricos importantes em redes de computadores, tais como os protocolos *Ethernet* e *TCP/IP*. Para além disso, adquiriu-se conhecimentos necessários para a configuração dos dispositivos que irão fazer parte de uma rede: computadores, *routers* e *switches*.

A realização deste trabalho laboratorial permitiu obter uma nova visão sobre redes de computadores, que cada vez mais estão presentes no nosso dia, quer no computador, quer nos dispositivos móveis.

5. Referências

- Postel & Reynolds, RFC 959, http://datatracker.ietf.org/doc/rfc959/ acedido em 29 de novembro de 2012
- die.net, http://www.die.net/ acedido em 29 de novembro de 2012
- Brian Hall, Beej's Guide to Network Programming Using Internet Sockets, http://beej.us/guide/bgnet/output/print/bgnet_A4.pdf acedido em 1 de dezembro de 2012

Anexos

ftpClient.c

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

#define FTP_PORT 21
#define MAXLENGTH 1024

char user[MAXLENGTH];
char password[MAXLENGTH];
char host[MAXLENGTH];
char path[MAXLENGTH];
char* ipAddress;

int sockfd;
int datasockfd;
int portConnection;

/**
 * Function to verify the input address and save credentials, host and path into arrays.
 * @param address Address string.
 */
void verifyAddress(char* address)
{
	char protocol[7];

	bzero(protocol, 7);

	strncpy(protocol, address, 6);

	//Verify protocol
	if(strcmp(protocol, "ftp://") != 0)
	{
		printf("Not valid protocol!\n");
		exit(1);
	}

	if (strlen(address) == 6)
	{
		printf("Address not found!\n");
		exit(1);
	}

	char * separatorUser = strchr(address + 6, ':');
```

```
char * separatorAt;
char * separatorPath;

// Verify if there is an user
if(separatorUser != NULL)
{
	//verify char separators

	separatorAt = strchr(separatorUser, '@');

	if(separatorAt == NULL)
	{
		printf("Missing separator @ after user:password!\n");
		exit(1);
	}

	separatorPath = strchr(separatorAt, '/');

	if(separatorPath == NULL)
	{
		printf("Missing separator between host and path!\n");
		exit(1);
	}

	char userLength = separatorUser - address - 6;
	char passwordLength = separatorAt - separatorUser - 1;
	char hostLength = separatorPath - separatorAt - 1;
	char pathLength = address + strlen(address) - separatorPath - 1;

	if(userLength == 0)
	{
		printf("User not found\n");
		exit(1);
	}

	if(passwordLength == 0)
	{
		printf("Password not found\n");
		exit(1);
	}

	if(hostLength == 0)
	{
		printf("Host not found\n");
		exit(1);
	}

	if(pathLength == 0)
	{
		printf("Path not found\n");
		exit(1);
	}

	strncpy(user, address+6, userLength);
	strncpy(password, separatorUser + 1, passwordLength);
	strncpy(host, separatorAt + 1, hostLength);
	strcpy(path, separatorPath + 1);
}
```

```
	else if(separatorUser == NULL)
	{
		// verify char separators

		separatorAt = strchr(address, '@');

		if(separatorAt != NULL)
		{
			printf("There is no user and password!\n");
			exit(1);

		}

		separatorPath = strchr(address + 6, '/');

		if(separatorPath == NULL)
		{
			printf("Missing separator between host and path!\n");
			exit(1);
		}

		char hostLength = separatorPath - address - 6;
		char pathLength = address + strlen(address) - separatorPath - 1;

		if(hostLength == 0)
		{
			printf("Host not found\n");
			exit(1);
		}

		if(pathLength == 0)
		{
			printf("Path not found\n");
			exit(1);
		}

		strncpy(host, address + 6, hostLength);
		strcpy(path, separatorPath + 1);
	}
}

/**
 * Function to get message sent from server.
 * @param message Message received from server.
 * @return Message code.
 */
int getMessage(char * message)
{
	bzero(message, MAXLENGTH);

	char buffer[MAXLENGTH];
	char trash[MAXLENGTH];
	bzero(buffer, MAXLENGTH);
	bzero(trash, MAXLENGTH);

	sleep(1);
```

```
	int totalRead = read(sockfd, buffer, MAXLENGTH);
	printf("%s\n", buffer);
	int i = 0;

	if(buffer[3] != ' ')
	{
		while(totalRead == MAXLENGTH)
		{
			sleep(1);
			totalRead = read(sockfd, trash, MAXLENGTH);
			printf("%s\n", trash);

			if(trash[3] == ' ')
				break;
		}
	}

	if(totalRead <= 0)
	{
		shutdown(sockfd,SHUT_RDWR);
		close(sockfd);
		exit(1);
	}

	for(i = 0; i < MAXLENGTH; i++)
	{
		if(buffer[i] < '0' || buffer[i] > '9')
			break;
	}

	char tempNum[5];
	bzero(tempNum,5);

	strncpy(tempNum, buffer, i);

	//get message number
	int statusConnection = atoi(tempNum);

	buffer[totalRead] = '\0';

	strcpy(message, buffer + i);

	return statusConnection;
}

/**
 * Function to start connection with the given host.
 */
void startConnection()
{
	struct hostent *h;
	struct sockaddr_in server_addr;

	// from getip.c with some changes
	if ((h=gethostbyname(host)) == NULL) {
		printf("Error while getting ip address!\n");
		exit(1);
```

```
        }

        printf("Host name  : %s\n", h->h_name);

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

        // from clientTCP.c
        bzero((char*)&server_addr,sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = inet_addr(ipAddress);
        server_addr.sin_port = htons(FTP_PORT);

        if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
                perror("socket()");
                exit(1);
        }
        /*connect to the server*/
        if(connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0){
                perror("connect()");
                exit(1);
        }

        char message[MAXLENGTH];

        int state = getMessage(message);

        if(state != 220)
        {
                printf("Error while getting first message!\n");
                shutdown(sockfd,SHUT_RDWR);
                close(sockfd);
                exit(1);
        }
}

/**
 * Function to send credentials and open UDP socket.
 */
void sendStartData()
{
        char message[MAXLENGTH];
        char buffer[MAXLENGTH];
        int written;
        int state;

        // send credentials

        // send user
        bzero(message, MAXLENGTH);

        if(strlen(user) == 0)
        {
                strcpy(user, "anonymous");
        }

        printf("user: %s\n", user);
```

```
strcpy(message, "USER ");
strcat(message, user);
strcat(message, "\r\n");

written = write(sockfd, message, strlen(message));

if(written <= 0)
{
	printf("Error when writing user to socket\n");
	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
	exit(1);
}

bzero(buffer, MAXLENGTH);

state = getMessage(buffer);

if(state >= 400)
{
	printf("Error while getting user authentication message!\n");
	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
	exit(1);
}

// send pass

bzero(message, MAXLENGTH);

if(strlen(password) == 0)
{
	strcpy(password, "anonymous");
}

strcpy(message, "PASS ");
strcat(message, password);
strcat(message, "\r\n");

written = write(sockfd, message, strlen(message));

if(written <= 0)
{
	printf("Error when writing pass to socket\n");
	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
	exit(1);
}

bzero(buffer, MAXLENGTH);

state = getMessage(buffer);

if(state >= 400)
{
	printf("Error when receiving pass ack!\n");
	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
```

```
        exit(1);
}
bzero(buffer, MAXLENGTH);
bzero(message, MAXLENGTH);

// start passive mode
strcpy(message, "PASV\r\n");
written = write(sockfd, message, strlen(message));

if(written <= 0)
{
        printf("Error when writing PASV to socket\n");
        shutdown(sockfd,SHUT_RDWR);
        close(sockfd);
        exit(1);
}

state = getMessage(buffer);

if(state >= 400)
{
        printf("Error while getting password authentication message!\n");
        shutdown(sockfd,SHUT_RDWR);
        close(sockfd);
        exit(1);
}

int port1;
int port2;

char * commaRight = strrchr(buffer, ',');

if(commaRight == NULL)
{
        printf("ERROR when getting the comma on right\n");
        exit(1);
}

commaRight[0] = '\0';
char * commaLeft = strrchr(buffer, ',');

if(commaLeft == NULL)
{
        printf("ERROR when getting the comma on left\n");
        exit(1);
}

commaLeft[0] = '\0';
commaRight++;

port1 = atoi(commaLeft + 1);
port2 = atoi(commaRight);

portConnection = port1*256 + port2;

//data socket
struct sockaddr_in server_addr;
```

```
	// from clientTCP.c
	bzero((char*)&server_addr,sizeof(server_addr));
	server_addr.sin_family = AF_INET;
	server_addr.sin_addr.s_addr = inet_addr(ipAddress);
	server_addr.sin_port = htons(portConnection);

	if ((datasockfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
		perror("socket()");
		exit(1);
	}
	/*connect to the server*/
	if(connect(datasockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0){
		perror("connect()");
		exit(1);
	}
}

/**
 * Function to download file. It will be created in the executable folder.
 */
void receiveData(){

	int state;
	char message[MAXLENGTH];
	char buffer[MAXLENGTH];

	bzero(message, MAXLENGTH);
	bzero(buffer, MAXLENGTH);

	//send retrieve message
	strcpy(message, "RETR /");
	strcat(message, path);
	strcat(message, "\r\n");

	int written = write(sockfd, message, strlen(message));

	if(written <= 0)
	{
		printf("Error when writing RETR to socket\n");
		shutdown(sockfd,SHUT_RDWR);
		close(sockfd);
		shutdown(datasockfd,SHUT_RDWR);
		close(datasockfd);
		exit(1);
	}

	state = getMessage(buffer);

	if(state >= 400)
	{
		printf("Error while getting retrieve message!\n");
		shutdown(sockfd,SHUT_RDWR);
		close(sockfd);
		shutdown(datasockfd,SHUT_RDWR);
		close(datasockfd);
		exit(1);
	}
```

```
// Open file here

int fd;
int readchars = 1;

bzero(buffer, MAXLENGTH);

char * filename = strrchr(path, '/') + 1;

printf("filename: %s\n", filename);

//delete last file
unlink(filename);

// open new file
fd = open(filename, O_WRONLY | O_CREAT, 0777);

if (fd < 0){
	printf("Problems opening file!\n");
	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
	shutdown(datasockfd,SHUT_RDWR);
	close(datasockfd);
	exit(1);
}
printf("Progress:\n");
while(readchars>0)
{
	bzero(buffer, MAXLENGTH);

	readchars=recv(datasockfd, buffer, MAXLENGTH, 0);
	printf(":");
	fflush(stdout);
	if(readchars > 0)
	{
		written = write(fd, buffer, readchars);

		if(written <= 0)
		{
			printf("Error when writing to file\n");
			shutdown(sockfd,SHUT_RDWR);
			close(sockfd);
			shutdown(datasockfd,SHUT_RDWR);
			close(datasockfd);
			exit(1);
		}
	}
}

close(fd);
printf("\n");
bzero(buffer, MAXLENGTH);

printf("Finishing\n");
sleep(3);

// get last messages that could be in the socket
recv(sockfd, buffer, MAXLENGTH, MSG_DONTWAIT);
```

```
	printf("%s\n", buffer);
}

/**
 * Main function.
 * @return Program state.
 */
int main(int argc, char *argv[])
{
	bzero(user, MAXLENGTH);

	if(argc != 2)
	{
		printf("usage: ftpClient ftp://[<user>:<password>@]<host>/<url-path>\n");
		exit(1);
	}

	verifyAddress(argv[1]);

	if(strlen(user) != 0){
		printf("%c[%d;%dmUSER: %c[%dm",27,1,35,27,0);
		printf("%s\n", user);

	}


	if(strlen(host) != 0){
		printf("%c[%d;%dmHOST: %c[%dm",27,1,35,27,0);
		printf("%s\n", host);
	}

	if(strlen(path) != 0){
		printf("%c[%d;%dmPATH: %c[%dm",27,1,35,27,0);
		printf("%s\n", path);
	}

	startConnection();

	sendStartData();
	receiveData();

	printf("%c[%d;%dmALL OK!!%c[%dm\n",27,1,32,27,0);

	shutdown(sockfd,SHUT_RDWR);
	close(sockfd);
	shutdown(datasockfd,SHUT_RDWR);
	close(datasockfd);

	return 0;
}
```