Computer Networks

(2.º Trabalho Laboratorial)

Redes de Computadores

LEIC - 12/2023

Turma 2 - Grupo 4

André dos Santos Faria Relva - up202108695@fe.up.pt

Rui Pedro Almeida da Silveira - up202108878@fe.up.pt

17 de dezembro de 2023

1. Sumário

Este trabalho foi desenvolvido no âmbito da disciplina de Redes de Computadores e envolve não só a implementação de uma aplicação de *download* de ficheiros através do protocolo FTP, como também da configuração gradual de uma rede na qual será testada essa mesma aplicação.

Através da exploração dos conceitos abaixo tratados, como IP, ARP, DNS, ou NAT, foi-nos possível consolidar os conhecimentos não só da configuração como também do funcionamento de redes de um modo holístico.

2. Introdução

Há muito que se possa dizer sobre o paradigma moderno que estabelece o conceito de "rede" num contexto informático. O presente trabalho visa fundamentar os conceitos principais envolvidos neste campo de conhecimento através da configuração e uso efetivo de uma rede devidamente acompanhada com análise de resultados de várias experiências a cada passo do processo, de forma a garantir uma compreensão minuciosa dos aspetos abrangidos.

As duas partes principais do trabalho—ambas a criação da aplicação e a configuração da rede—serão discutidas abaixo.

3. Aplicação de *download*

Durante o desenvolvimento do presente projeto, foi requisitado o desenvolvimento de uma simples aplicação que, ao receber um URL de FTP válido através da linha de comandos, estabeleceria uma ligação ao servidor indicado e descarregaria o ficheiro pedido automaticamente através de modo passivo.

Tal tarefa implicou a aplicação de conhecimentos sobre o protocolo FTP em si, assim como sobre *Berkeley sockets*. Para além disto, foi indispensável a implementação de um *parser* para o URL recebido—neste caso através do uso de *regex*—de modo a assegurar um funcionamento correto da aplicação.

3.1. Estrutura do código

A nossa implementação da aplicação de *download* está dividida em três componentes funcionais principais: *parse.c*, *format.c* e *ftp.c*. Estes são responsáveis por fazer *parsing* do URL, codificar os caracteres especiais nele presentes, e efetuar a comunicação cliente e servidor, respetivamente. O *driver code* está presente em *main.c*, que apenas trata de facilitar a interação entre os três componentes.

```
// parse.h / parse.c

void fill_field(char *str, regmatch_t *match, char **field);
int parse_url(char *url, url_params *params, char **filename);
long parse_port(char *buffer);
```

A principal função deste componente é `parse_url`, que tratará de receber o URL da linha de comandos, assim como um *pointer* para o *struct* a preencher com os parâmetros recolhidos do URL e para o *buffer* onde escrever o nome do ficheiro que será recolhido.

Tal como previamente indicado, o *parsing* foi implementado através de *regex* com a seguinte expressão:

```
const char *pattern = "ftp://(((.*):(.*)@)|(.*)@)?([^/]*)/(.*)";
```

Esta funciona do seguinte modo:

- Se os primeiros dois grupos de captura não estiverem vazios, o terceiro e quarto grupos conterão o nome de utilizador e palavra-passe fornecidos;
- Se algum dos dois primeiros grupos de captura estiverem vazios, então o quinto grupo de captura conterá o nome de utilizador fornecido;
- Em qualquer dos casos, o sexto grupo de captura conterá o *hostname*, enquanto que o sétimo conterá o *path* e o oitavo o nome do ficheiro.

Através desta expressão de *regex*, foi simples concluir o resto desta componente funcional. A função `fill_field` simplesmente é subsequentemente chamada por `parse_url` para preencher os campos de `params` fornecidos e deste modo temos acesso a todos os dados necessários para o passo seguinte.

```
// format.h / format.c
void format_url(url_params *url_params);
```

A função acima que trata de formatar os parâmetros é também muito simples, apenas indo buscar os campos no *struct* fornecido e colocando-os de volta com os caracteres especiais substituídos de acordo com a codificação URL.

```
// ftp.h / ftp.c
int ftp(url_params *params, char *filename);
```

Finalmente, a função `ftp` irá tratar de toda a comunicação com o servidor FTP, assim como o descarregamento do ficheiro como previamente indicado.

Isto tudo é conseguido do seguinte modo:

1. A função obtém o IP correspondente ao *host* fornecido através de uma função auxiliar;
2. É preparada a *socket* de controlo que dará início à comunicação com o servidor;
3. Ao estabelecer ligação, efetua o *login*, quer com as credenciais fornecidas pelo utilizador, ou, no caso da sua ausência, com o utilizador e palavra-passe “*anonymous*,” tal como indicado no guião do projeto;
4. É obtido, em *bytes*, o tamanho do ficheiro a ser transferido;
5. A *socket* de controlo dirige o servidor a abrir uma porta em modo passivo;

6. É preparada e ligada uma *socket* à porta obtida no passo anterior que funcionará como a recetora do ficheiro;
7. A *socket* de controlo pede ao servidor para transmitir o ficheiro através da conexão estabelecida através da *socket* recetora.
8. É criado o ficheiro com o nome fornecido, sendo este subsequentemente preenchido com os dados fornecidos do servidor.

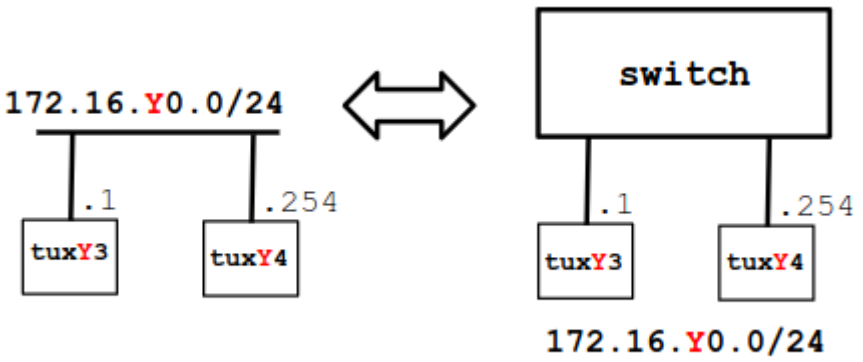
4. Configuração da rede

Adicionalmente ao desenvolvimento da aplicação acima descrita, foram também efetuadas várias experiências para conduzir a aprendizagem e o entendimento do funcionamento, configuração e componentes de uma rede.

De seguida serão detalhadas as seis experiências realizadas durante as aulas de laboratório, devidamente acompanhadas da análise dos seus resultados

Como o trabalho foi desenvolvido na *Workbench* 4 da sala I320, a nomenclatura utilizada terá como base esse aspeto.

4.1. Experiência 1 - Configuração de uma rede IP



Tal como podemos ver na imagem acima, a primeira experiência tratou-se da configuração de uma *bridge* entre duas máquinas através do *switch* disponibilizado na *workbench*.

Para começar, foram configurados os IPs das máquinas *tux43* e *tux44*:

```
# tux43
ifconfig eth0 up
ifconfig eth0 172.16.40.1/24
# tux44
ifconfig eth0 up
ifconfig eth0 172.16.40.254/24
```

Deste modo, são atribuídos a ambas as máquinas endereços de IP na mesma rede,

tendo esta o endereço base 172.16.40.0, sendo este indicado pelo "/24" presente nos comandos que sinaliza que os *bits* fixos da rede são os 24 *bits* mais significativos. Por outras palavras, a rede aqui configurada teria 255.255.255.0 como máscara.

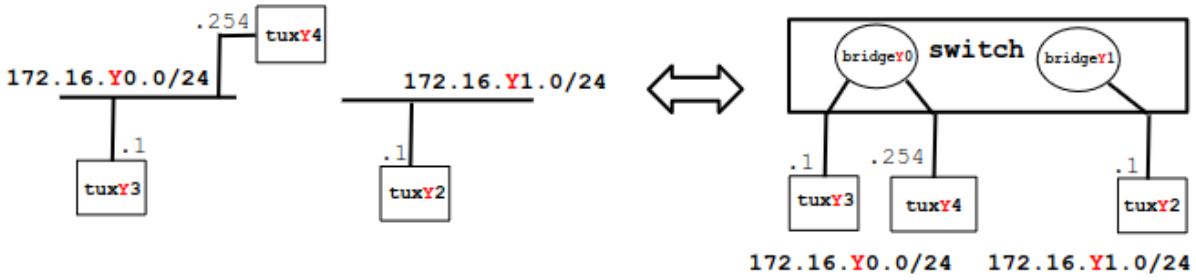
Ao executar um *ping* do *tux43* ao *tux44 com* a tabela ARP vazia (esta pode ser apagada através do comando *arp -d <IP>* e verificada com *arp -a*), podemos ver através do Wireshark que o *tux43 começa* por transmitir um pacote ARP. Desconhecendo o endereço MAC da máquina de destino, um pacote *Address Resolution Protocol* (ARP) é enviado em *broadcast*, ou seja, para todas as máquinas presentes na rede, para que a máquina a quem se destina o pacote—no nosso caso, o *tux44*—informe o *tux43* do seu endereço MAC. Para tal efeito, o pacote conterá o endereço da máquina de origem e de destino. De seguida, e como previsto, a máquina *tux44* acaba por responder ao *tux43* com o seu endereço MAC.

Os pacotes ARP aqui discutidos são usados precisamente neste contexto: para através de um endereço IP, obter o endereço MAC da máquina com que se pretende estabelecer ligação, de modo a mapear o endereço na *Internet* ao endereço físico da máquina.

Após esta troca de mensagens, o comando *ping* gerará pacotes de tipo *Internet Control Message Protocol* (ICMP). Esta diferença entre tipos de pacote é indicada no cabeçalho do mesmo, no qual também está presente o tamanho do *frame*.

Todas as máquinas contém uma interface *loopback*. Esta trata-se de uma interface virtual que, após ser configurada, nunca será modificada, assumindo que não ocorram problemas técnicos. Uma vez que é sempre acessível é muito útil para efetuar diagnósticos.

4.2. Experiência 2 - Implementação de duas *bridges* num *switch*



Nesta seguinte experiência, configuramos os IPs das máquinas *tux42*, *tux43* e *tux44* do modo semelhante ao que foi feito na experiência anterior.

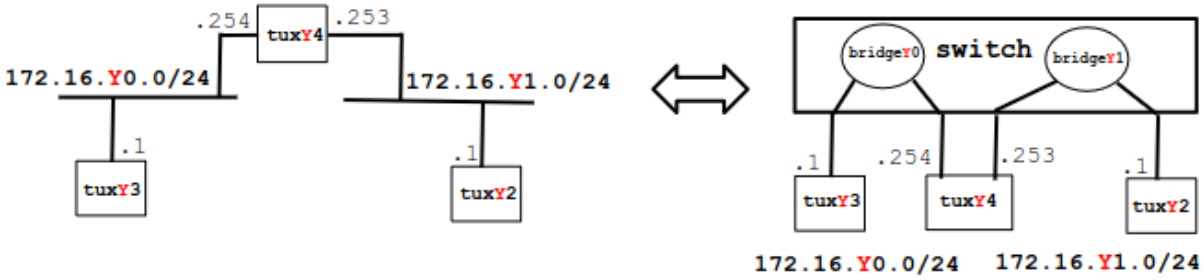
Seguidamente, através do terminal do *switch*, configuramos as duas bridges do seguinte modo:

```
# Criação das bridges
/interface bridge add name=bridge40
/interface bridge add name=bridge41

# Remoção das portas da bridge padrão
# Os números correspondentes a cada porta mudam após cada remoção,
# pelo qual é necessário verificar o número correspondente a cada
# interface com o comando /interface bridge port print
/interface bridge port remove numbers=X
```

```
/interface bridge port remove numbers=Y
/interface bridge port remove numbers=Z

# Adição das portas em uso a cada bridge
/interface bridge port add bridge=bridge40 interface=ether2
/interface bridge port add bridge=bridge40 interface=ether4
/interface bridge port add bridge=bridge41 interface=ether10
```

Ao fazer *ping* ao endereço *broadcast* de cada rede a partir do *tux42* e do *tux43*, confirmamos que, uma vez que as duas máquinas estão ligadas a *bridges* diferentes, existem dois domínios de *broadcast*, justificado pelo facto que os *logs* do Wireshark mostram que o *tux43* não obteve resposta do *tux42* e vice-versa.

4.3. Experiência 3 - Configurar um *router* em Linux



A terceira experiência adiciona um outro elemento de complexidade ao *setup* da rede: *routing*. A máquina *tux44* será o *router* nesta rede, tratando de encaminhar os pacotes entre ambas as subredes, uma vez que tem acesso a ambas.

Para tal, primeiro é necessário repetir o processo da experiência anterior, notando que a interface *eth1* do *tux44* também terá de ser configurada de modo semelhante às outras interfaces, incluindo a sua adição à *bridge41*.

Adicionalmente, será necessário correr os seguintes comandos:

```
# Ativar IP forwarding no tux44
sysctl net.ipv4.ip_forward=1
# Desativar ICMP echo ignore broadcast no tux44
sysctl net.ipv4.icmp_echo_ignore_broadcasts=0

# Adicionar a rota em tux43 para que possa alcançar o tux42
route add -net 172.16.41.0/24 gw 172.16.40.254
# Adicionar a rota em tux42 para que possa alcançar o tux43
route add -net 172.16.40.0/24 gw 172.16.41.253
```

Primeiramente, configuramos o *tux44* para funcionar como *router*, ativando o IP *forwarding*, que permitirá ao mesmo encaminhar pacotes destinados a outra máquina após reconhecer que não se destina a si. Isto efetivamente permitirá a correta atribuição de papel

de *gateway* ao *tux44*.

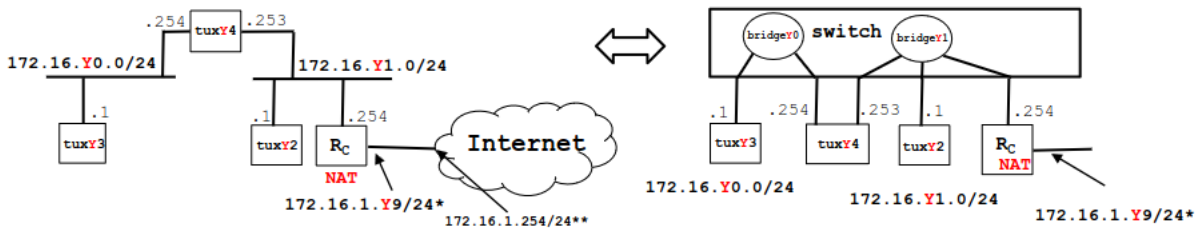
Seguidamente, é necessário configurar as rotas no *tux42* e no *tux43* tal como indicado acima. Isto é devido ao facto de que uma máquina só conseguirá enviar pacotes a máquinas pertencentes a uma rede que conheça. Estando ambas em subredes diferentes, estas não saberão para onde enviar os pacotes. Daí que configuramos uma rota manualmente; estamos essencialmente a instruir o computador a enviar os pacotes destinados à outra subrede através do *tux44*, uma vez que, ao ter uma interface ligada a cada subrede, tem acesso a ambas, podendo direcionar pacotes livremente através destas.

Após efetuar esta configuração, podemos verificar que cada entrada da *forwarding table* contém a *gateway* definida e o endereço de rede correspondente.

Adicionalmente, ao efetuar *ping* do *tux43* ao *tux42* com as tabelas ARP vazias, vemos que os pacotes ARP trocados têm os endereços MAC do *tux43* e *tux44*, uma vez que são as únicas máquinas que exibem comunicação efetiva na subrede 172.16.40.0/24, assim como são as únicas máquinas de que o *tux43* tem consciência.

Semelhantemente, os endereços MAC presentes nos pacotes ICMP são os mesmos dos endereços presentes nos pacotes ARP e pelas mesmas razões, enquanto que os endereços IP são, de facto, do *tux43* (de origem) e do *tux42* (de destino), uma vez que serão utilizados para direcionar os pacotes até ao ponto onde possam ser resolvidos para um endereço físico; daí a diferença relativamente aos endereços MAC.

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



Na quarta experiência, não só é adicionado um novo componente à rede—um *router* comercial, no caso—como também é utilizada uma tecnologia denominada *Network Address Translation* (NAT). Tal como o nome sugere, trata-se do mapeamento de um espaço de endereçamento a outro, que no caso concreto que tratamos aqui, será a tradução do endereço IP público aos endereços IP locais, que permitirá ligação com a Internet.

Repetindo, mais uma vez, o *setup* da experiência anterior, podemos avançar para os comandos seguintes:

```
# Configurar a gateway padrão no tux43
route add default gw 172.16.40.254
# Configurar a gateway padrão no tux42 e no tux44
route add default gw 172.16.41.254

# Configurar ambas as portas do router comercial após limpar a
# configuração já existente, ligando a porta ether1 ao patch panel com
```

```
# ligação à internet e a porta ether2 à bridge41
/ip address add address=172.16.2.49/24 interface=ether1
/ip address add address=172.16.41.254/24 interface=ether2

# Configurar a gateway para acessar a subrede da bridge40
# no router comercial
/ip route add dst-address=172.16.40.0/24 gateway=172.16.41.253
# Configurar a default gateway para acesso à internet
# no router comercial
/ip route add dst-address=0.0.0.0/0 gateway=172.16.2.254

# Configurar NAT na porta com acesso à internet após limpar a
# configuração no router comercial
/ip firewall nat add chain=srcnat action=masquerade out-interface=ether1
```

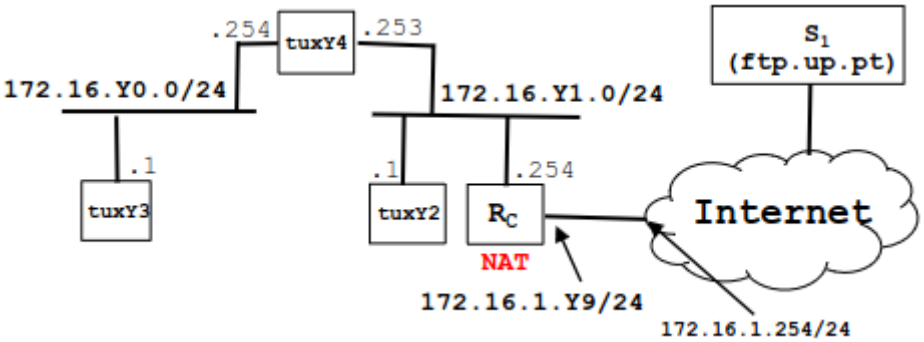
Primeiramente, configuramos a *gateway* padrão em todas as máquinas. Isto fará com que todos os pacotes destinados a endereços IP que caiam fora da gama definida previamente para as subredes da *bridge40* e *bridge*41 sejam encaminhados corretamente para o *router comercial*, uma vez que isso significará que têm destino fora da rede, nomeadamente, na Internet.

Seguidamente, já na terminal do *router* comercial, configuramos ambos os endereços IP que serão utilizados consoante o diagrama acima.

Após isto, configuramos as *gateways*, ou rotas estáticas, de modo a que os pacotes consigam chegar de volta à *bridge40* uma vez que o *router* comercial não tem acesso direto à mesma e também para direcionar os pacotes para o *router* central de modo a que cheguem à Internet, do qual será responsável a *gateway* padrão, tal como descrito na experiência anterior.

Finalmente, podemos configurar a NAT no *router* comercial e verificar que tudo funciona corretamente.

4.5. Experiência 5 - DNS



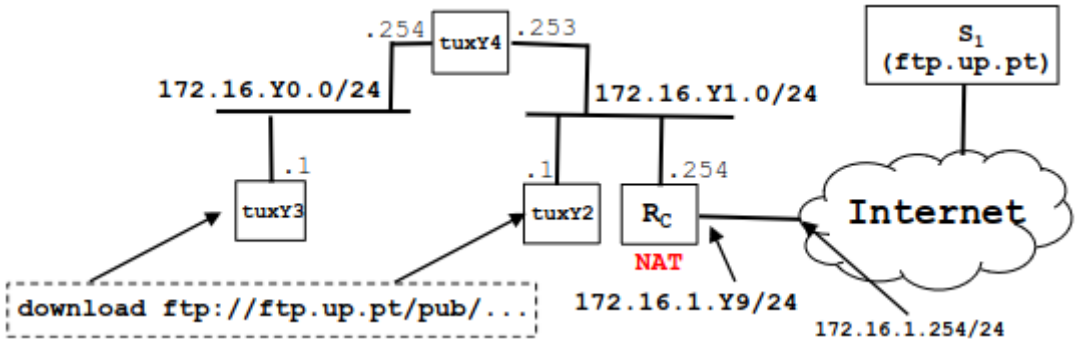
Nesta penúltima experiência, o objetivo é apenas testar a funcionalidade de resolução de *Domain Name System* (DNS). Isto trata-se da tradução de *hostnames*, como, por exemplo, *ftp.up.pt*, para endereços de IP concretos.

Para tal, apenas é necessário adicionar a seguinte linha ao ficheiro */etc/resolv.conf* de cada máquina:

```
nameserver 172.16.2.1
```

Isto indicará às máquinas que o *router* comercial será o responsável por tratar de DNS, o que faz sentido visto ter acesso direto à Internet. E, de facto, verificando os *logs*, vemos que ao correr o comando *ping* com um *hostname* em vez de um endereço IP explícito, são enviados pacotes da máquina de origem ao router comercial a requisitar que responda com o endereço IP correspondente ao *hostname* fornecido, ao qual efetivamente o *router* comercial responde. Logo de seguida, agora munido com o endereço IP de destino, a máquina começa a enviar simples pacotes ICMP como de costume.

4.6. Experiência 6 - Conexões TCP



Finalmente, na última experiência, ambas as partes do trabalho podem ser consolidadas. Para tal efeito, a aplicação desenvolvida na primeira parte será posta em uso na rede configurada ao longo das experiências da segunda parte de modo a analisar o fluxo de pacotes numa conexão TCP.

Ao analisar a rede com o Wireshark durante o funcionamento da aplicação, vemos que são abertas duas ligações TCP. Visto que foram usadas duas *sockets*—uma de controlo para enviar comandos e outra como recetor para descarregar o ficheiro—vemos o resultado esperado.

É através da primeira ligação TCP que é transportada a informação de controlo FTP, uma vez que os comandos para o servidor são enviados através da mesma, assim como são obtidas as respostas do servidor.

As ligações TCP têm três fases como podemos ver nos *logs*:

- **Estabelecimento de ligação:** o estabelecimento de ligação é efetuado através de um envio de um pacote SYN (*synchronize*) e a resposta do servidor com um pacote SYN-ACK (*synchronize-acknowledge*), ao qual o cliente acaba por responder um pacote ACK. Isto é chamado de TCP *handshake* ou *three-way handshake*;
- **Transferência de dados:** a transferência de dados ocorre através do envio de pacotes FTP, no caso da experiência em curso, aos quais o cliente responde com pacotes TCP do tipo ACK.

- **Terminação da ligação:** a terminação de ligação é feita de modo semelhante ao estabelecimento da mesma, exceto que a sequência de pacotes trocados é FIN (*finish)*, FIN-ACK e ACK.

É interessante ressaltar a variação da largura de banda ao longo da experiência: ao criar a segunda conexão TCP, esta é dividida entre as duas.

5. Conclusão

Com este trabalho, consideramos que foi de facto obtida uma maior e melhor compreensão sobre os conceitos acima explorados, e que os resultados obtidos foram os esperados: não só foi possível verificar a correta implementação da aplicação de *download* através das experiências realizadas com a análise do tráfego de rede, como também os objetivos coincidiram com os resultados obtidos sem nenhum problema.

Anexo I

- ***format.h***

```
#ifndef FORMAT_H
#define FORMAT_H

#include "url_params.h"

void format_url(url_params *url_params);

#endif
```

- ***ftp.h***

```
#ifndef FTP_H
#define FTP_H

#include "url_params.h"

int ftp(url_params *params, char *filename);

#endif
```

- ***parse.h***

```
#ifndef PARSE_H
```

```c
#define PARSE_H

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

#include "url_params.h"

#define USERPASS_FIELD 1
#define USERPASS_FIELD2 2
#define USERPASS_USER 3
#define USERPASS_PASS 4
#define USERONLY_USER 5
#define HOSTNAME 6
#define PATH 7

#define CHECK_MATCH(match) (match.rm_so != -1 && match.rm_eo != -1)

void fill_field(char *str, regmatch_t *match, char **field);

/* Parses the URL passed in by filling in the structure provided in
`match`
  alongside the filename in its tail and returns 0 on success, -1
otherwise.
  Memory is only allocated on success, not needing any freeing in
failure. */
int parse_url(char *url, url_params *params, char **filename);
long parse_port(char *buffer);

#endif
```

- ***url_params.h***

```c
#ifndef URL_PARAMS_H
#define URL_PARAMS_H

typedef struct {
   char *username;
   char *password;
   char *hostname;
   char *path;
} url_params;

void free_params(url_params *params);
```

```
#endif
```

- ***utils.h***

```
#ifndef UTILS_H
#define UTILS_H

char *get_password(void);

#endif
```

- ***format.c***

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

#include "format.h"

#define ADDED_CHARACTERS 7

char *format(char *field) {
   const char *reserved = "!#$&'()*+,/:;=?@[]";

   size_t length = strlen(field);
   size_t reserved_length = strlen(reserved);

   int counter = 0;

   for (size_t i = 0; i < length; i++) {
       for (size_t j = 0; j < reserved_length; j++) {
           if (field[i] != reserved[j]) continue;
           counter++;
           break;
       }
   }

   size_t new_length = length + 2 * counter + 1;
   char *ret = calloc(new_length, sizeof(char));

   counter = 0;
```

```
    for (size_t i = 0; i < length; i++) {
        bool is_reserved = false;
        for (size_t j = 0; j < reserved_length; j++) {
            if (field[i] != reserved[j]) continue;
            ret[i + 2 * counter] = '%';
            sprintf(&ret[i + 2 * counter + 1], "%X", reserved[j]);
            counter++;
            is_reserved = true;
            break;
        }
        if (!is_reserved) ret[i + 2 * counter] = field[i];
    }

    free(field);
    return ret;
}

void format_url(url_params *params) {
    params->hostname = format(params->hostname);
    params->path = format(params->path);
    params->username = format(params->username);
    params->password = format(params->password);
}
```

- ***ftp.c***

```
#include "ftp.h"
#include "parse.h"

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

#define SERVER_PORT 21
#define BUFFER_SIZE 1024

#define LOGIN_SUCCESS 230
```

```
#define POST_STATUS_CODE 4

#define FTP_WRITE(fd, buf, label, format, ...)      memset(buf, 0,
BUFFER_SIZE);                                                    \
                                                   sprintf(buf, format
__VA_OPT__(,) __VA_ARGS__);                             \
                                                   if (write(fd, buf,
strlen(buf)) < 0) {                                       \
                                                       printf("[ERROR]
Failed to write to socket: %s.\n", strerror(errno));     \
                                                       return_code = -1;
goto label;                                              \
                                                   }

#define FTP_RECV(fd, buf, label)                    memset(buf, 0,
BUFFER_SIZE);                                                    \
                                                   if (recv(fd, buf,
BUFFER_SIZE, 0) < 0) {                                    \
                                                       printf("[ERROR]
Failed to read from socket: %s\n.", strerror(errno));    \
                                                       return_code = -1;
goto label;                                              \
                                                   }

char *get_ip(char *hostname) {
   struct hostent *host_info;

   host_info = gethostbyname(hostname);
   if (host_info == NULL) {
       printf("[ERROR] Failed to get host's IP address: %s.\n",
hstrerror(h_errno));
       return NULL;
   }

   return inet_ntoa(*((struct in_addr *)host_info->h_addr));
}

int ftp(url_params *params, char *filename) {

   int return_code = 0;
   size_t filesize = 0;

   char *ip = get_ip(params->hostname);
   if (ip == NULL) return - 1;

   int ctrl_socket;
```

```
    struct sockaddr_in ctrlsockaddr_info;
    memset(&ctrlsockaddr_info, 0, sizeof(struct sockaddr_in));
    ctrlsockaddr_info.sin_family = AF_INET;
    ctrlsockaddr_info.sin_addr.s_addr = inet_addr(ip);
    ctrlsockaddr_info.sin_port = htons(SERVER_PORT);

    if ((ctrl_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("[ERROR] Failed to open socket: %s.\n", strerror(errno));
        return_code = -1; goto exit;
    }

    if (connect(ctrl_socket, (struct sockaddr*)&ctrlsockaddr_info,
sizeof(ctrlsockaddr_info)) < 0) {
        printf("[ERROR] Failed to connect to server: %s.\n",
strerror(errno));
        return_code = -1; goto close_ctrl;
    }

    char buffer[BUFFER_SIZE] = {0};

    FTP_RECV(ctrl_socket, buffer, close_ctrl);

    FTP_WRITE(ctrl_socket, buffer, close_ctrl, "USER %s\r\n",
params->username);
    FTP_RECV(ctrl_socket, buffer, close_ctrl);

    FTP_WRITE(ctrl_socket, buffer, close_ctrl, "PASS %s\r\n",
params->password);
    FTP_RECV(ctrl_socket, buffer, close_ctrl);

    if (strtol(buffer, NULL, 10) != LOGIN_SUCCESS) {
        printf("[ERROR] Invalid credentials.\n");
        return_code = -1; goto close_ctrl;
    }

    FTP_WRITE(ctrl_socket, buffer, close_ctrl, "TYPE I\r\n");
    FTP_RECV(ctrl_socket, buffer, close_ctrl);

    FTP_WRITE(ctrl_socket, buffer, close_ctrl, "SIZE %s\r\n", filename);
    FTP_RECV(ctrl_socket, buffer, close_ctrl);

    filesize = strtoul(&buffer[POST_STATUS_CODE], NULL, 10);

    FTP_WRITE(ctrl_socket, buffer, close_ctrl, "PASV\r\n");
    FTP_RECV(ctrl_socket, buffer, close_ctrl);
```

```
    long port;
    if ((port = parse_port(buffer)) < 0) {
        return_code = 1; goto close_ctrl;
    }

    int recv_socket;

    struct sockaddr_in recvsockaddr_info;
    memset(&recvsockaddr_info, 0, sizeof(struct sockaddr_in));
    recvsockaddr_info.sin_family = AF_INET;
    recvsockaddr_info.sin_addr.s_addr = inet_addr(ip);
    recvsockaddr_info.sin_port = htons(port);

    if ((recv_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("[ERROR] Failed to open socket: %s.\n", strerror(errno));
        return_code = -1; goto close_ctrl;
    }

    if (connect(recv_socket, (struct sockaddr*)&recvsockaddr_info,
sizeof(ctrlsockaddr_info)) < 0) {
        printf("[ERROR] Failed to connect to server: %s.\n",
strerror(errno));
        return_code = -1; goto close_recv;
    }

    FTP_WRITE(ctrl_socket, buffer, close_recv, "RETR %s\r\n",
params->path);

    int fd;

    if ((fd = open(filename, O_CREAT | O_TRUNC | O_WRONLY, 0666)) < 0) {
        printf("[ERROR] Failed to create file: %s.\n", strerror(errno));
        return_code = -1; goto close_recv;
    }

    int bytes;
    unsigned char *file_buffer = malloc(filesize);

    while ((bytes = recv(recv_socket, file_buffer, filesize, 0)) != 0) {
        if (bytes < 0) {
            printf("[ERROR] Failed to receive file: %s.\n",
strerror(errno));
            return_code = -1; break;
        }
```

```
        if (write(fd, file_buffer, bytes) < 0) {
            printf("[ERROR] Failed to write to file: %s.\n",
strerror(errno));
            return_code = -1; break;
        }
    }

    free(file_buffer);

    if (close(fd) < 0) {
        printf("[ERROR] Failed to close file: %s.\n", strerror(errno));
        return_code = -1;
    }

close_recv:

    if (close(recv_socket) < 0) {
        printf("[ERROR] Failed to close client socket: %s.\n",
strerror(errno));
        return_code = -1;
    }

close_ctrl:

    if (close(ctrl_socket) < 0) {
        printf("[ERROR] Failed to close server control socket: %s.\n",
strerror(errno));
        return_code = -1;
    }

exit:

    return return_code;
}
```

- ***parse.c***

```
#include "parse.h"
#include "utils.h"

#include <string.h>

void fill_field(char *str, regmatch_t *match, char **field){
    int fieldLength = match->rm_eo - match->rm_so;
    // Null terminator
```

```
    *field = calloc(fieldLength + 1, sizeof(char));
    (void)strncpy(*field, &str[match->rm_so], fieldLength);
}

int parse_url(char *url, url_params *params, char** filename) {

    int return_code = 0;

    if (url == NULL) {
        printf("[ERROR] Please supply an FTP URL.\n");
        return_code = -1; goto url_exit;
    }

    regex_t result;
    memset(&result, 0, sizeof(regex_t));

    /* Pattern explanation (capture groups):
       If 1 and 2 are not empty, 3 and 4 are username and password,
respectively
       If 1 or 2 are empty, 5 is username
       6 is host
       7 is path
       Allowed: user:pass@; user:@; user@; :@;
       Otherwise, whichever characters are counted as part of hostname */
    const char *pattern = "ftp://(((.*):(.*)@)|(.*)@)?([^/]*)/(.*)";

    if (regcomp(&result, pattern, REG_EXTENDED)) {
        printf("[ERROR] Failed to compile regular expression.\n");
        return_code = -1; goto url_free_preg;
    }

    regmatch_t *matches = malloc((result.re_nsub + 1) *
sizeof(regmatch_t));

    // Added one since it automatically matches the whole string at index
zero
    // re_nsub has subexpressions; matches return whole expression +
subexpressions
    if (regexec(&result, url, result.re_nsub + 1, matches, 0)) {
        printf("[ERROR] Invalid URL. Please recheck and try again.\n");
        return_code = -1; goto url_free_matches;
    }

    // Username and password provided
    if (CHECK_MATCH(matches[USERPASS_FIELD]) &&
        CHECK_MATCH(matches[USERPASS_FIELD2])) {
```

```
        fill_field(url, &matches[USERPASS_USER], &params->username);

        if (strlen(params->username) == 0) {
            printf("[ERROR] Username field is empty.\n");
            free(params->username);
            return_code = -1; goto url_free_matches;
        }

        fill_field(url, &matches[USERPASS_PASS], &params->password);

        if (strlen(params->password) == 0) {
            printf("[ERROR] Password field is empty.\n");
            free(params->password);
            return_code = -1; goto url_free_matches;
        }

    // Only username provided
    } else if (CHECK_MATCH(matches[USERONLY_USER])) {
        fill_field(url, &matches[USERONLY_USER], &params->username);
    } else {
        params->username = calloc(10, sizeof(char));
        (void)sprintf(params->username, "%s", "anonymous");
        params->password = calloc(10, sizeof(char));
        (void)sprintf(params->password, "%s", "anonymous");
    }

    fill_field(url, &matches[HOSTNAME], &params->hostname);
    fill_field(url, &matches[PATH], &params->path);

    size_t pos = 0;
    for (size_t i = 0; i < strlen(params->path); i++) if (params->path[i]
== '/') pos = i;

    *filename = calloc(strlen(&params->path[pos]) + 1, sizeof(char));
    (void)strcpy(*filename, &params->path[pos]);

url_free_matches:
    free(matches);
url_free_preg:
    regfree(&result);
url_exit:
    return return_code;
}

long parse_port(char *buffer) {
```

```
    long ret;

    regex_t result;
    memset(&result, 0, sizeof(regex_t));

    const char pattern[] = ".*\\(([0-9]+,){4}([0-9]+),([0-9]+))\\).";

    if (regcomp(&result, pattern, REG_EXTENDED)) {
        printf("[ERROR] Failed to compile regular expression.\n");
        ret = -1; goto port_free_preg;
    }

    regmatch_t *matches = malloc((result.re_nsub + 1) *
sizeof(regmatch_t));

    if (regexec(&result, buffer, result.re_nsub + 1, matches, 0)) {
        printf("[ERROR] Could not retrieve valid port from server.\n");
        ret = -1; goto port_free_matches;
    }

    ret = (strtol(&buffer[matches[3].rm_so], NULL, 10) << 8) +
           strtol(&buffer[matches[4].rm_so], NULL, 10);

port_free_matches:
    free(matches);
port_free_preg:
    regfree(&result);

    return ret;
}
```

- ***url_params.c***

```
#include "url_params.h"

#include <stdlib.h>

void free_params(url_params *params) {
    free(params->username);
    free(params->password);
    free(params->hostname);
    free(params->path);
}
```

- ***utils.c***

```c
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include "utils.h"

#define MAX_PASSWORD_SIZE 128

char *get_password(void) {
   struct termios old, new;

   tcgetattr(STDIN_FILENO, &old);
   new = old;

   new.c_lflag &= ~ECHO;
   tcsetattr(STDIN_FILENO, TCSANOW, &new);

   printf("Password: ");
   fflush(stdout);

   char *password = calloc(MAX_PASSWORD_SIZE + 1, sizeof(char));
   if (fgets(password, MAX_PASSWORD_SIZE + 1, stdin) == NULL)
password[0] = '\0';
   else password[strlen(password) - 1] = '\0';

   printf("\n");

   tcsetattr(STDIN_FILENO, TCSANOW, &old);
   return password;
}
```

- ***main.c***

```c
#include <stdlib.h>
#include <string.h>

#include "parse.h"
#include "format.h"
#include "utils.h"
#include "ftp.h"

int main(int argc, char *argv[]) {
```

```
    if (argc < 2) {
        printf("[ERROR] Please provide URL.\n");
        printf("Usage: %s ftp://[user:pass@]host/path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    url_params params;
    memset(&params, 0, sizeof(url_params));

    char *filename = NULL;

    // No memory allocated on failure
    if (parse_url(argv[1], &params, &filename) < 0) exit(EXIT_FAILURE);

    if (params.username != NULL && params.password == NULL)
params.password = get_password();

    format_url(&params);

    int return_code = ftp(&params, filename);

    free_params(&params);
    free(filename);

    return return_code == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```