Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

Licenciatura em Engenharia Informática e de Computadores

1º Trabalho prático de Redes e Computadores

Fase 1 - Servidor Web

Alun@s:
A49470 - Carolina
A49506 - Pedro
A49418 - Roberto

Docentes:
Nuno Miguel Luís

Abril 2022

Conteúdo

1 Introdução

1.1 Paradigma cliente-servidor

Um paradigma cliente-servidor é composto por um servidor e por clientes. O cliente é quem inicia sempre a comunicação e pode assumir vários endereços IP, porém, clientes não podem comunicar entre si. Os servidores têm que estar sempre ligados (para estarem aptos a receber sempre informação) e têm endereços IP permanentes. Estes podem ser hospedados em data centers dependendo do âmbito que lhes é dado. Com isto, é errado pensar que num paradigma cliente-servidor, só o cliente envia e só o servidor recebe porque no fim o servidor também envia ao cliente.

1.2 Protocolos TCP e UDP

Existem dois tipos de protocolos na camada de aplicação, o TCP e o UDP. O TCP tem por base um transporte fiável pois dá uma garantia de entrega da informação, ou seja, se por algum motivo a transmissão falhar ele volta a transmitir. Tem também controlo de fluxo, ou seja, o emissor tem cuidado para não sobrecarregar o recetor pois se por exemplo, o emissor tiver uma taxa de transmissão muito elevada mas o recetor estiver a receber informações de vários sitios, então o recetor não vai ter vazão para despachar tantos pedidos se o emissor não abrandar. O TCP tem controlo de congestão pois percebe quando é que a rede está congestionada e é orientado à ligação ou seja, para ser possivel enviar algo sobre o TCP, é necessário primeiro estabelecer uma sessão para garantir que estamos ligados. Por fim, este não garante nem tempo, nem segurança mas garante a entrega.

Já o protocolo UDP não dá garantias de um transporte fiável nem garante, controlo de fluxo, controlo de congestão, tempo, segurança assim como estabelecimento de sessão.

1.3 Tipos de conexões HTTP

HTTP ou hypertext transfer protocol pertence à camada de aplicação da web. Quando um computador ou um telemóvel tenta aceder um site, o browser realiza pedidos ao servidor onde se encontra esse site. Esse servidor envia uma resposta, que é recebida pelo browser novamente. Esta comunicação é feita utilizando o protocolo HTTP. Esta comunicação entre browser e servidor dá-se sob uma conexão TCP, onde o browser inicia esta conexão ao servidor, porto 80. De seguida, o servidor aceita essa ligação e as mensagens são trocadas. Por fim, a ligação TCP é encerrada. Existem dois tipos de conexões HTTP: HTTP não-persistente e HTTP persistente. Ambos estes tipos de conexões são realizados através de protocolos TCP, no entanto, o que os difere é que no HTTP não-persistente, apenas pode ser enviado 1 objeto de cada vez e no HTTP persistente podem ser enviados varios objetos simultaneamente. Como já foi referido anteriormente, existem dois tipos de mensagens HTTP. O pedido do browser e a resposta do servidor. Na mensagem de pedido, a primeira linha é iniciada por GET, PUT, HEAD, entre outros, que descrevem a ação a ser executada. A primeira linha contém também o URL e a versão de HTTP utilizada. De seguida temos o cabeçalho que contem diversas informações

como o host, linguagens aceites, entre outras. Por fim temos o corpo, embora nem todos os pedidos o contenham. Já as respostas, na primeira linha, informam sobre a versão de protocolo, seguida de um código de status (como por exemplo 200 ou 404) e por uma descrição textual breve, puramente informativa, com o objetivo de tornar a sua leitura mais simples. Logo a seguir, tal como nas mensagens de pedido, temos um cabeçalho, que contêm informações como a data atual, a data em que esse site foi modificado pela última vez, o tamanho do conteúdo em bytes, etc. Por fim temos um corpo, que tal como nas mensagens de pedido, não está cotigo em todas as mensagens de resposta.

1.4 Introdução ao trabalho

Nesta primeira fase do trabalho, o objetivo foi instalar um Servidor Web e testar a sua conexão com um programa feito por nós. O programa tem que estabelecer a ligação TCP e requisitar mensagens assim como recebê-las e mostrá-las ao cliente. O desenvolvimento do programa foi feito na linguagem de programação C. Também de importante referência o facto da utilização de sockets BSD[1] e a sua biblioteca[2] para a comunicação entre o cliente e o servidor, portanto para a execução e compilação(Appendix A) do programa será necessário um sistema Unix[3] ou um emulador, como o Cygwin[4].

2 Instalação e configuração do servidor web

O servidor instalado e configurado foi o Apache[5], através do XAMPP[6]. Para iniciar o servidor basta correrm os o seguinte comando em Linux:

```
sudo /opt/xampp/lampp start
```

enquanto que no Windows podemos iniciar o servidor utilizando a interface gráfica.

Uma das dúvidas iniciais era o porquê de os pedidos não aparecerem no Wireshark, porém depois de alguma pesquisa, descobrimos que todos os pedidos feitos a um servidor local, como neste caso o Apache, utilizam uma interface virtual, denominada `lo`.

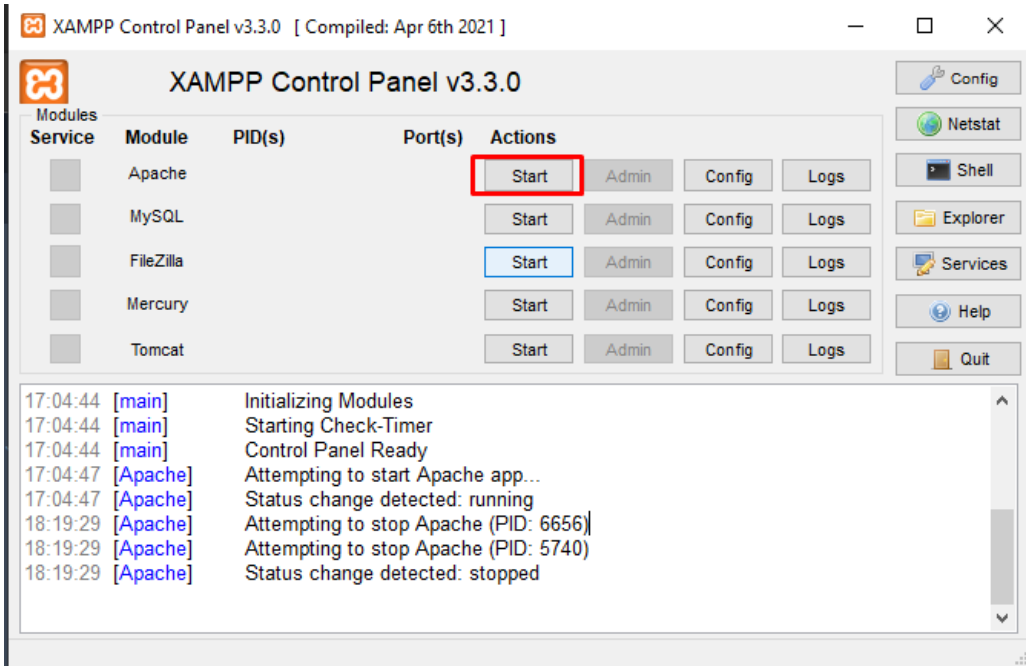

Figura 1: XAMPP em Windows

3 Aplicação

3.1 Execução

Depois de executar a aplicação especificando como paramêtros o `URL` ao qual nos queremos conectar, o programa envia uma mensagem para o servidor (neste caso, um *GET request* da página com path indicada no `URL`), e caso haja resposta, pergunta ao utilizador onde a exibir (figura 2).

A figura 3 mostra outro exemplo de execução, pedindo uma página diferente e a figura 4 mostra um exemplo de uma possível resposta do servidor.

3.2 Wireshark

Utilizamos o programa `Wireshark` para observar a comunicação feita entre o cliente e o servidor web. Para tal, usamos o filtro `http` para só aparecerem os pedidos e as respostas que utilizam o protocolo `HTTP`. Na figura 5 podemos observar a mensagem enviada para o servidor e a sua resposta.

3.2.1 Pedido HTTP (*HTTP Request*)

Ao executarmos o nosso programa e após a mensagem ser enviada, podemos observar na figura 6 o pedido no `Wireshark`.

3.2.2 Resposta HTTP (*HTTP Response*)

Também podemos observar a resposta do servidor na figura 7.

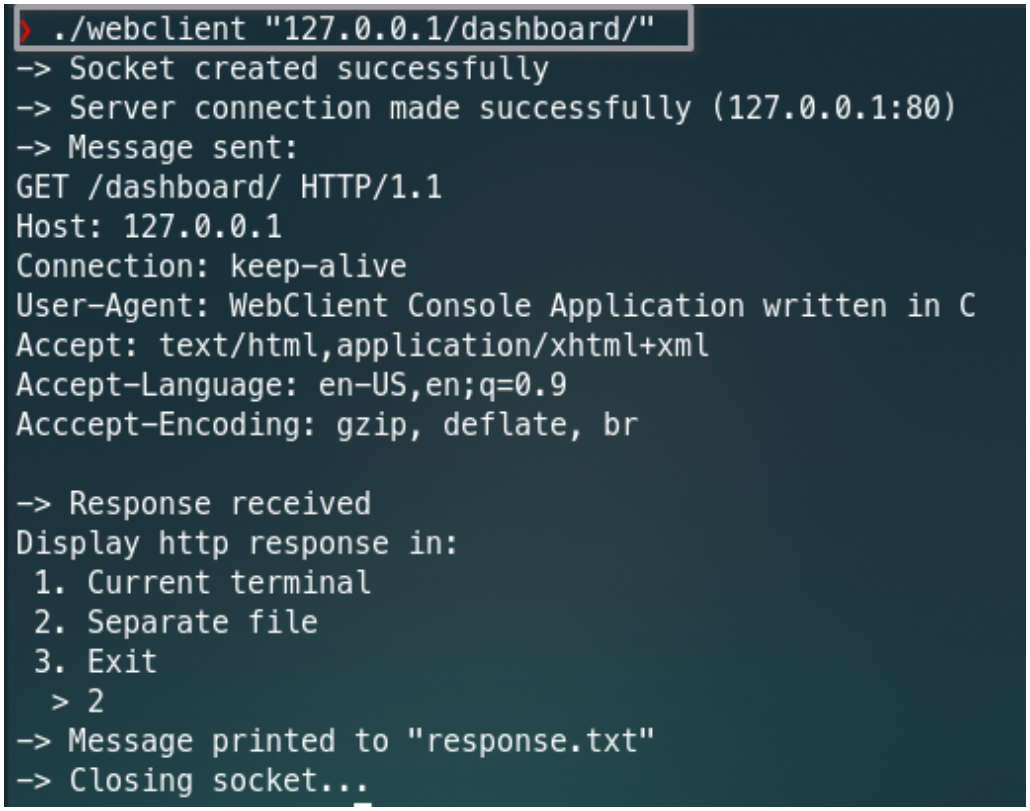

Figura 2: Execução do programa

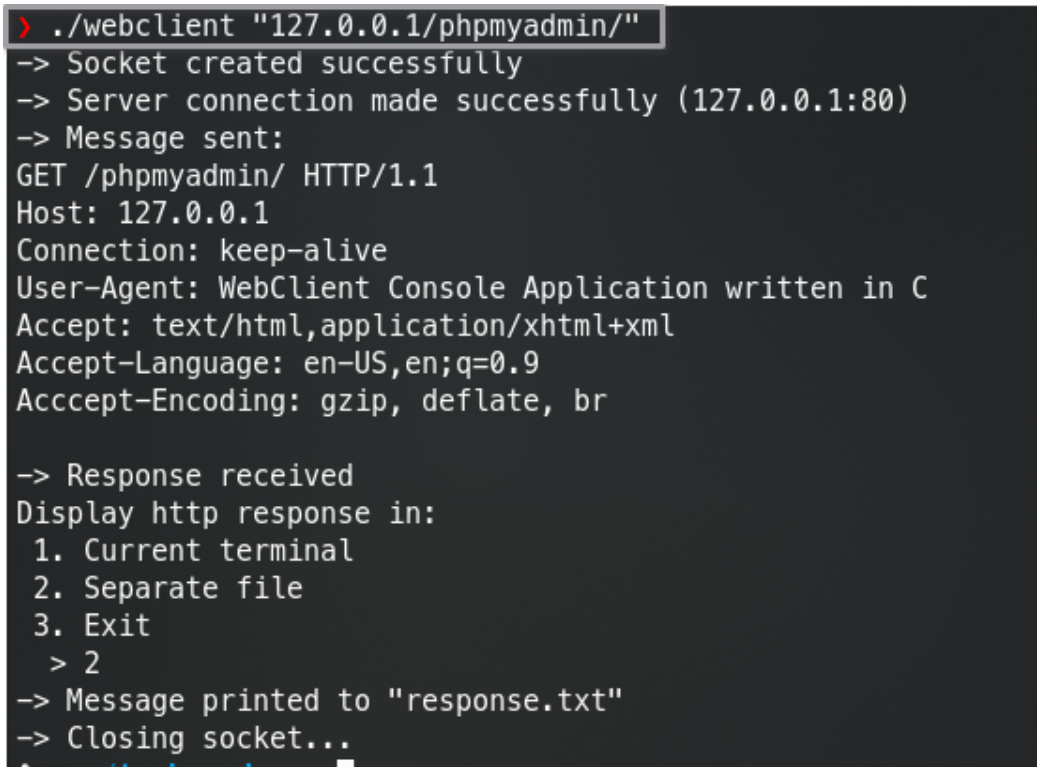

Figura 3: Outro exemplo de execução

File: **response.txt**
Size: **8.0 KB**

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2022 14:33:30 GMT
Server: Apache/2.4.53 (Unix) OpenSSL/1.1.1n PHP/8.1.4 mod_perl/2.0.12 Perl/v5.34.1
Last-Modified: Tue, 29 Mar 2022 14:57:50 GMT
ETag: "1d96-5db5ca711f780"
Accept-Ranges: bytes
Content-Length: 7574
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <!-- Always force latest IE rendering engine or request Chrome Frame -->
    <meta content="IE=edge,chrome=1" http-equiv="X-UA-Compatible">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <!-- Use title if it's in the page YAML frontmatter -->
    <title>Welcome to XAMPP</title>

    <meta name="description" content="XAMPP is an easy to install Apache distribution containing MariaDB, PHP and Perl.
" />
    <meta name="keywords" content="xampp, apache, php, perl, mariadb, open source distribution" />

    <link href="/dashboard/stylesheets/normalize.css" rel="stylesheet" type="text/css" /><link href="/dashboard/stylesh
eets/all.css" rel="stylesheet" type="text/css" />
    <link href="//cdnjs.cloudflare.com/ajax/libs/font-awesome/3.1.0/css/font-awesome.min.css" rel="stylesheet" type="te
xt/css" />

    <script src="/dashboard/javascripts/modernizr.js" type="text/javascript"></script>


    <link href="/dashboard/images/favicon.png" rel="icon" type="image/png" />
```

Figura 4: Possível resposta do servidor

http

No.	Time	Source	Destination	Protocol	Lengtl	Info
164	7.980399583	127.0.0.1	127.0.0.1	HTTP	302	GET /dashboard/ HTTP/1.1
166	7.980999213	127.0.0.1	127.0.0.1	HTTP	7979	HTTP/1.1 200 OK (text/html)

Figura 5: Mensagem e resposta no Wireshark

4 Cabeçalhos HTTP (*HTTP Headers*)

4.1 Cabeçalhos enviados

- `Host: 127.0.0.1\r\n` - Endereço do servidor
- `Connection: keep-alive\r\n` - Manter a conexão do socket TCP
- `User-Agent: WebClient Console Application written in C\r\n` - Identificador do cliente
- `Accept: text/html,application/xhtml+xml\r\n` - Tipo de dados que o cliente espera na resposta
- `Accept-Language: en-US,en;q=0.9\r\n` - Idiomas que o cliente espera na resposta
- `Acccept-Encoding: gzip, deflate, br\r\n` - Codificação do conteúdo que o cliente espera na resposta

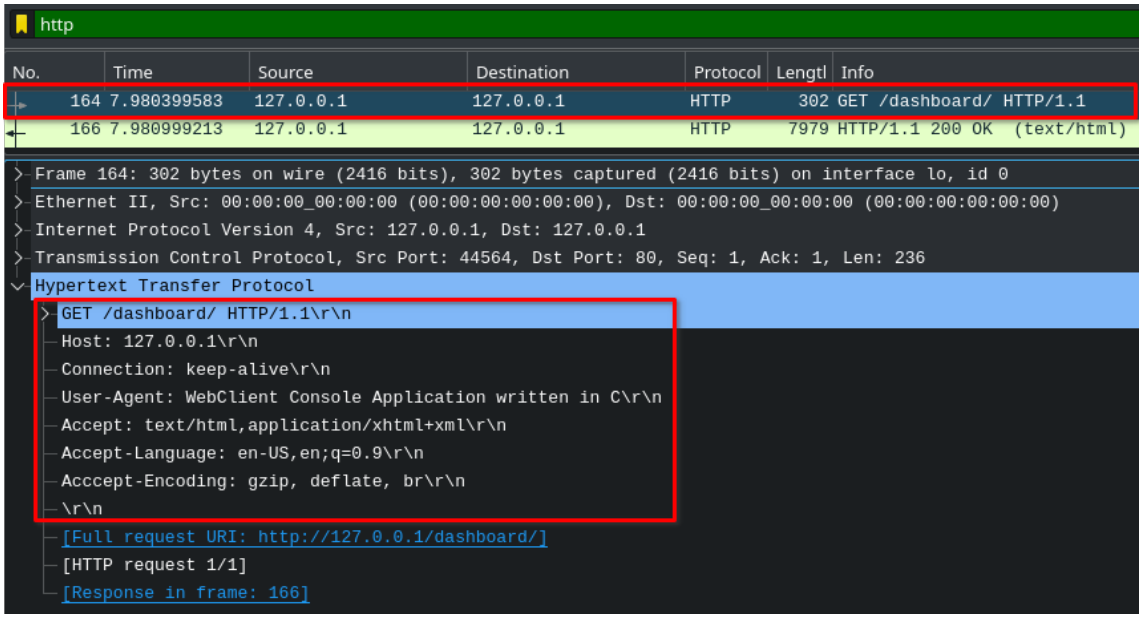

Figura 6: Pedido HTTP enviado ao servidor

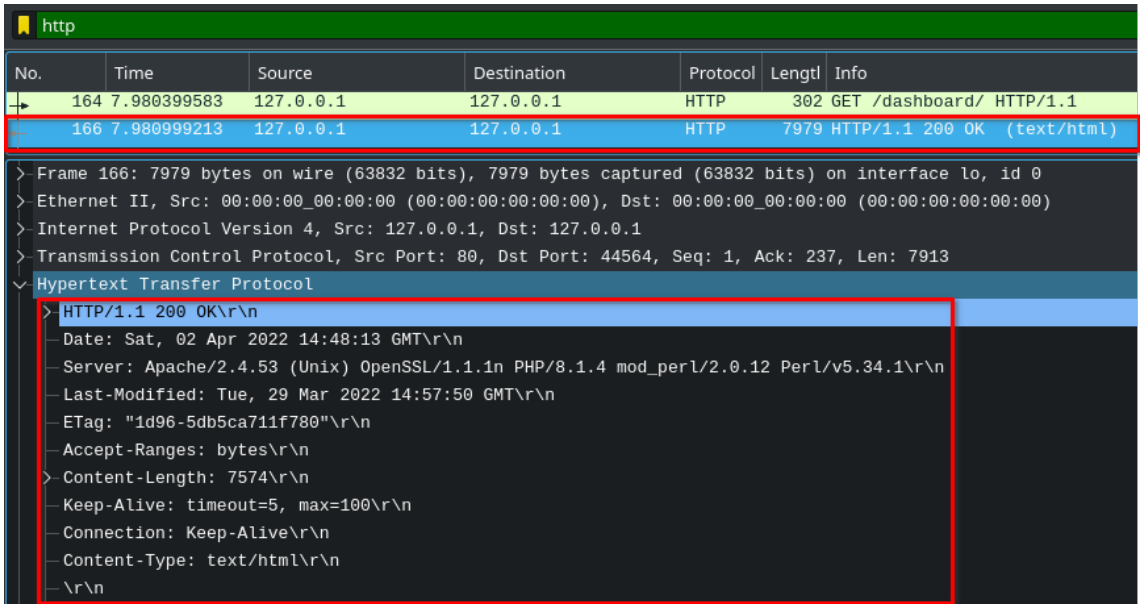

Figura 7: Resposta HTTP do servidor

- `\r\n` - Linha obrigatória no fim da mensagem

4.2 Cabeçalhos recebidos

Os cabeçalhos recebidos dependem sempre da mensagem, neste caso vamos assumir que a mensagem não tem erros e que o servidor vai responder com o estado `200 OK`.

- `Date: Sat, 02 Apr 2022 16:04:49 GMT` - Data atual
- `Server: Apache/2.4.53 (Win64) OpenSSL/1.1.1n PHP/8.1.4` - Informações do servidor

- `Last-Modified: Mon, 28 Mar 2022 14:50:08 GMT` - Data da última vez que o conteúdo da resposta foi modificado
- `ETag: "1d98-5db486db10800"` - Serve para validar se o cliente tem a versão mais recente do conteúdo
- `Accept-Ranges: bytes` - Indica que o servidor aceita pedidos parciais
- `Content-Length: 7576` - Tamanho do conteúdo, em bytes
- `Keep-Alive: timeout=5, max=100` - Indica um tempo limite da conexão, neste caso é entre 5 a 100 segundos
- `Connection: Keep-Alive` - Manter a conexão do socket TCP
- `Content-Type: text/html` - Indica o tipo dos dados

5 Implementação e código-fonte

5.1 Sobre a implementação

Como já referido anteriormente, a linguagem escolhida para desenvolver o cliente web foi C. Grande parte do ecossistema da linguagem foi desenvolvido para sistemas Unix[3], logo o programa só irá correr em sistemas desse tipo também. Implementamos a criação dos sockets e a comunicação utilizando a biblioteca BSD[2], em que os sockets funcionam de maneira diferente, pois seguem outra filosofia. Sendo assim, a simulação do programa só é possível em um ambiente Unix[3], ou então num emulador como Cygwin[4].

5.2 Código-fonte

5.2.1 Bibliotecas (*Headers*)

Aqui incluimos várias bibliotecas para o desenvolvimento do programa, sendo as mais importantes `sys/socket.h`[7] e `arpa/inet.h`[8], que contêm funções usadas principalmente para a implementação do cliente web.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
```

5.2.2 Constantes / Variáveis globais

De seguida declaramos uma constante `HTTP_RESPONSE_SIZE`, que representa o tamanho máximo para armazenar a resposta do servidor num vetor com esse tamanho e também `HTPP_PORT` que vai ser o porte do servidor a qual nos vamos conectar (`http`). Foi escolhido o valor 10000 para a constante `HTTP_RESPONSE_SIZE`, para não comprometer o conteúdo da resposta do servidor na sua totalidade. Também foi definida uma constante `msg` que irá armazenar texto, neste caso é os cabeçalhos da mensagem que iremos enviar para o servidor, com dois `placeholders (%s)`, que iremos preencher mais tarde depois de obter o paramêtro do cliente e obter a `path`(caminho) da página que queremos e o `host`(endereço do servidor).

```
#define HTTP_RESPONSE_SIZE 10000
#define HTTP_PORT 80

const char msg_template[] =
    "GET /%s HTTP/1.1\r\n"
    "Host: %s\r\n"
    "Connection: keep-alive\r\n"
    "User-Agent: WebClient Console Application written in C\r\n"
    "Accept: text/html,application/xhtml+xml\r\n"
    "Accept-Language: en-US,en;q=0.9\r\n"
```

```
    "Acccept-Encoding: gzip, deflate, br\r\n"
    "\r\n";
```

5.2.3 Declaração das funções utilizadas

Declaramos as funções antes para dar a conhecer ao compilador que estas funções serão usadas na função `main`.

```
/// creates a tcp socket
int create_tcp_socket();

/// opens the connection to the server via tcp socket
void connect_to_server(int socket_fd, const char* host, int port);

/// handle message (let client pick if it should be printed to terminal or file)
void handle_http_response(const char* http_response);
```

5.2.4 Rotina principal: main()

Definição da rotina `main`, função que é o "ponto de partida"do nosso programa, ou seja, a primeira a ser executada. Os seus paramêtros, `int argc`, `char* argv[]`, indicam a quantidade de parametros e também os seus valores, respetivamente. Quando queremos executar o programa, temos sempre que passar como parametro o `URL` da página que pretendemos. O valor (em texto) de cada argumento é o valor de `argv` no seu índice, ou seja, `argv[1]` corresponse ao `URL`. Para separarmos o endereço do servidor e o caminho da página, utilizamos a função strtok[9], e armazenamos o seu valor em `host` e `path`, respetivamente. Caso a `path` seja nula, simplesmente armazenamos um espaço lá para podermos pedir a página base (`/`).

```
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: ./webclient <URL> \n");
        return EXIT_FAILURE;
    }

    // get arguments
    char* url = argv[1];
    const char* host = strtok(url, "/");
    char* path = strtok(NULL, "");
    if (path == NULL) {
        path = malloc(1);
        strcpy(path, "");
    }
```

5.2.5 Criação da mensagem

De seguida alocamos memória para a mensagem que queremos enviar para o servidor, e utilizamos a função sprintf para formatar a string, substituindo os `%s` anteriormente

referidos na `msg_template`. Com isto, a mensagem está pronta com os dados corretos para ser enviada.

```
char* msg = malloc(sizeof(msg_template) + strlen(path) + strlen(host));
sprintf(msg, msg_template, path, host);
```

5.2.6 Criação de um socket TCP

De seguida tentamos criar um socket TCP, invocando a função `create_tcp_socket`, que por si invoca a função `socket`[10] da biblioteca `sys/socket.h`[7] que retorna um socket TCP se possível, caso contrário, o programa acaba com uma mensagem de erro.

```
int socket_fd = create_tcp_socket();
printf("-> Socket created successfully\n");

/// creates a tcp socket
int create_tcp_socket() {
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if(socket_fd == -1){
        perror("ERROR: Unable to create socket\nREASON");
        exit(EXIT_FAILURE);
    }
    return socket_fd;
}
```

5.2.7 Conectar ao servidor

De seguida tentamos conectar ao servidor utilizando o socket anteriormente criado. Para tal criamos um "objeto"do tipo `struct sockaddr_in` com os dados do nosso servidor. `sockaddr_in` é uma estrutura com funções e dados que lidam com endereços de internet. Depois utilizamos a função `connect`[11] da biblioteca `sys/socket.h`[7], que passando o objeto criado como argumento, a função tenta conectar o socket ao servidor. Novamente, caso a conexão não foi feita com sucesso, o programa acaba com uma mensagem de erro.

```
connect_to_server(socket_fd, host, port);
printf("-> Server connection made successfully (%s:%d)\n", host, port);

/// opens the connection to the server via tcp socket
void connect_to_server(int socket_fd, const char* host, int port) {
    // specify server info
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr.s_addr = inet_addr(host);

    // connect
```

```
    if(connect(socket_fd, (struct sockaddr*)&server_addr, sizeof(server_addr))
        == -1) {
        printf("ERROR: Unable to connect to %s:%d\n", host, port);
        perror("REASON");
        exit(EXIT_FAILURE);
    }
}
```

5.2.8 Enviar a mensagem ao servidor

Após a conexão, enviamos uma mensagem (`msg`) para o servidor utilizando o socket criado anteriormente. Para tal usamos outra função da biblioteca `sys/socket.h`[7], `send`[12] e a mensagem (mencionada na secção 5.2.2).

```
// send message
if(send(socket_fd, msg, strlen(msg), 0) == -1) {
    perror("ERROR: Unable to send message toserver\nREASON");
    return EXIT_FAILURE;
}
printf("-> Message sent: \n%s", msg);
```

5.2.9 Receber uma resposta do servidor

Após o envio da mensagem, o programa espera por uma resposta. Para armazenar essa resposta, criamos uma variável `char response[HTTP_RESPONSE_SIZE]` e utilizando a função recv[13] da biblioteca sys/socket.h[7], esperamos pela mensagem, que quando recebida, é armazenada nessa variável.

```
// receive and store server response
char response[HTTP_RESPONSE_SIZE];
if(recv(socket_fd, response, sizeof(response), 0) ==-1) {
    perror("ERROR: Unable to receive message fromserver\nREASON");
    return EXIT_FAILURE;
}
printf("-> Response received \n");
```

5.2.10 Exibir a resposta ao cliente

A função `handle_http_response` recebe a resposta do servidor como argumento e, dependendo da escolha do utilizador, é selecionada a opção de onde a exibir. Poderá ser exibida no próprio terminal ou num ficheiro à parte.

```
// let client pick where to print the response msg
handle_http_response(response);

/// handle message
void handle_http_response(const char* http_response) {
    int opc;
    do {
```

```
        printf("Display http response in:\n"
               "1. Current terminal\n"
               "2. Separate file\n"
               "3. Exit \n  > ");
        scanf("%d", &opc);
        switch(opc) {
            case 1: {
                printf("-> Message     received:\n%s",http_response);
                break;
            }
            case 2: {
                FILE* fp = fopen("response.txt", "w");
                fprintf(fp, http_response);
                fclose(fp);
                printf("-> Message printed     to\"response.txt\"\n");
                break;
            }
            case 3: {
                break;
            }
            default: {
                printf("Invalid option! \n\n");
            }
        }
    } while (opc != 1 && opc != 2 && opc != 3);
}
```

5.2.11 Acabar a execução

```
printf("-> Closing socket...\n");
close(socket_fd);
free(msg);
return EXIT_SUCCESS;
```

Por fim, fechamos o socket, libertamos a memória alocada anteriormente para a mensagem e acabamos com a execução do programa.

6 Conclusão

Concluindo o trabalho, conseguimos aprofundar os nossos conhecimentos em redes de computadores, principalmente acerca de tópicos como sockets tcp, protocolo http e a comunicação entre um cliente e um servidor na Internet.

Referências

[1] BSD Sockets. Berkeley sockets is an application programming interface (api) for internet sockets and unix domain sockets. https://en.wikipedia.org/wiki/Berkeley_sockets.

[2] BSD API. Bsd socket api functions. https://en.wikipedia.org/wiki/Berkeley_sockets#History_and_implementations.

[3] Unix. Unix is an operating system. https://pt.wikipedia.org/wiki/Unix.

[4] Cygwin. Cygwin is a collection of open source tools that allows unix or linux applications to be compiled and run on a microsoft windows operating system. https://www.cygwin.com/.

[5] Apache. Apache http server is a free and open-source web server that delivers web content through the internet. https://www.apachefriends.org/index.html.

[6] XAMPP. Xampp is an apache distribution containing mariadb, php, and perl. https://en.wikipedia.org/wiki/Berkeley_sockets#History_and_implementations.

[7] sys/socket.h. Core socket functions and data structures. https://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html.

[8] arpa/inet.h. Functions for manipulating numeric ip addresses. https://pubs.opengroup.org/onlinepubs/7908799/xns/arpainet.h.html.

[9] strtok. Split string into tokens. https://www.cplusplus.com/reference/cstring/strtok/.

[10] socket. Create an endpoint for communication. https://man7.org/linux/man-pages/man2/socket.2.html.

[11] connect. Initiate a connection on a socket. https://man7.org/linux/man-pages/man2/connect.2.html.

[12] send. Send a message on a socket. https://man7.org/linux/man-pages/man2/send.2.html.

[13] recv. Receive a message from a socket. https://man7.org/linux/man-pages/man2/recv.2.html.

[14] gcc. The gnu compiler collection, commonly known as gcc, is a set of compilers and development tools. https://en.wikipedia.org/wiki/Berkeley_sockets#History_and_implementations.

A Appendix: Compilar e executar o programa

Para compilar e executar o programa, precisamos do emulador Cygwin[4] ou de um sistema operativo baseado em Unix (e.g Linux) com o compilador `gcc`[14] instalado. Assumindo que já nos encontramos na mesma pasta que o ficheiro webclient.c, podemos compilar com o seguinte comando:

```
gcc webclient.c -o client
```

E para executarmos:

```
./client 127.0.0.1 80
```

As figuras 8 e 9 exemplificam os dois passos, num sistema operativo Linux e no Cygwin[4] respetivamente.

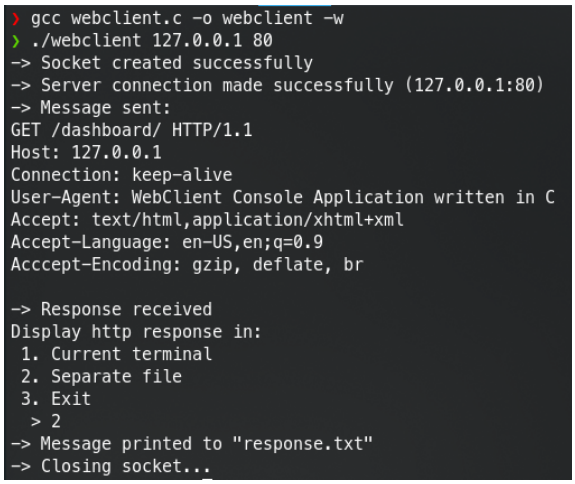

Figura 8: Compilação e execução do programa em Linux

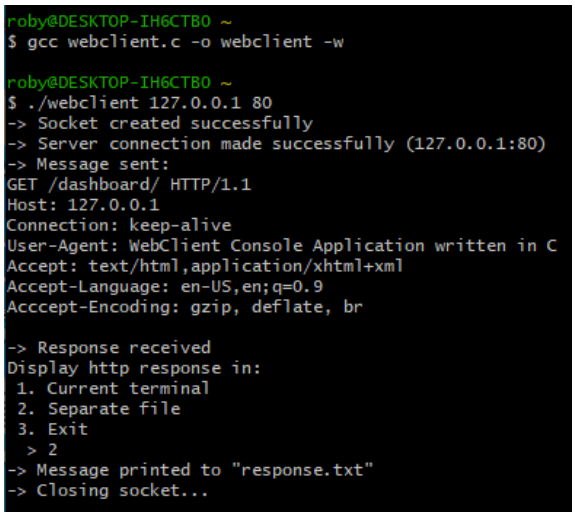

Figura 9: Compilação e execução do programa no Cygwin