Padua University

Engineering Course

*Master of Computer Engineering*

# Computer Networks

Raffaele Di Nardo Di Maio

Accademic Year 2019-2020

# Contents

Chapter 1

# C programming

The C is the most powerful language and also can be considered as the language nearest to Assembly language. Its power is the speed of execution and the easy interpretation of the memory.
C can be considered very important in Computer Networks because it doesn't hide the use of system calls. Other languages made the same thing, but hiding all the needs and evolution of Computer Network systems.

## 1.1 Organization of data

Data are stored in the memory in two possible ways, related to the order of bytes that compose it. There are two main ways, called Big Endian and Little Endian.

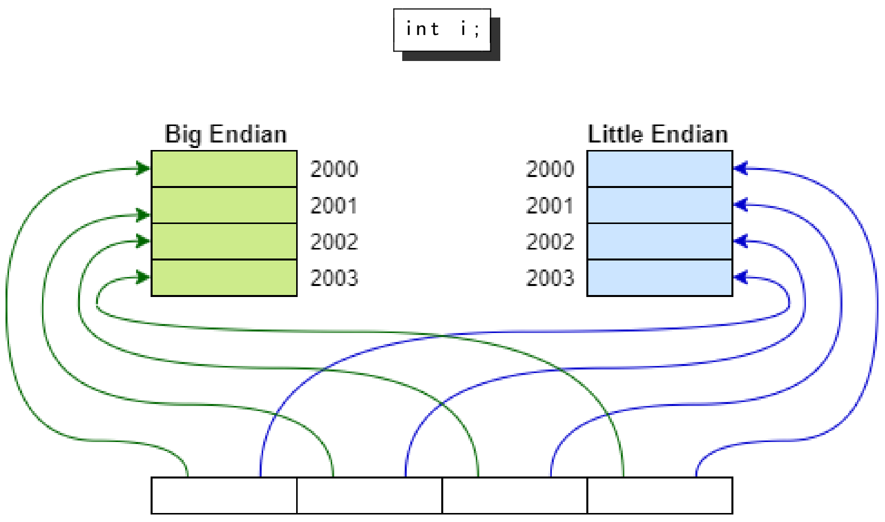

Figure 1.1: Little Endian and Big Endian.

The order of bytes in packets, sent through the network, is Big Endian.
The size of **int, float, char, ...** types depends on the architecture used. The max size of possible types depends on the architecture (E.g. in 64bits architecture, in one istruction, 8 bytes can be written and read in parallel).

| **signed** | unsigned |
|---|---|
| int8_t | uint8_t |
| int16_t | uint16_t |
| int32_t | uint32_t |
| int64_t | uint64_t |

Table 1.1: <stdint.h>

## 1.2 Struct organization of memory

The size of a structure depends on the order of fields and the architecture. This is caused by alignment that depends on the number of memory banks, number of bytes read in parallel. For example the size is 4 bytes for 32 bits architecture, composed by 4 banks (Figure 1.2). The Network Packet Representation is made by a stream of 4 Bytes packets as we're using 32 bits architecture.

```
struct example1                struct example2
{                              {
  char c;                        int x;
  int x;                         char c;
}                              }
```

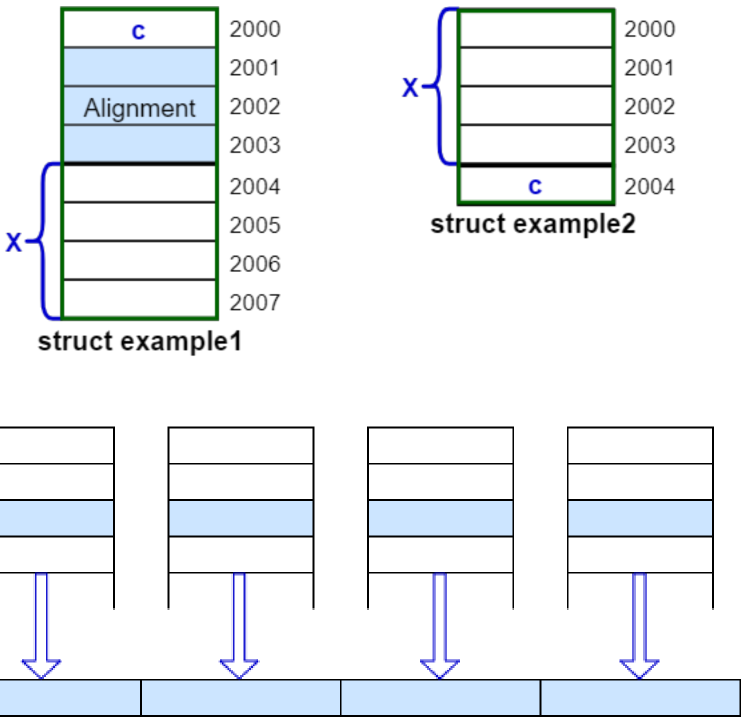

Figure 1.2: Parallel reading in one istruction in 32 bits architecture.

## 1.3 Structure of C program

The program stores the variable in different section (Figure 1.3):

- **Static area**
  where global variables and static library are stored, it's initialized immediately at the creation of the program. Inside this area, a variable doesn't need to be initialized by the programmer because it's done automatically at the creation of the program with all zeroes.
- **Stack**
  allocation of variables, return and parameters of functions
- **Heap**
  dinamic allocation

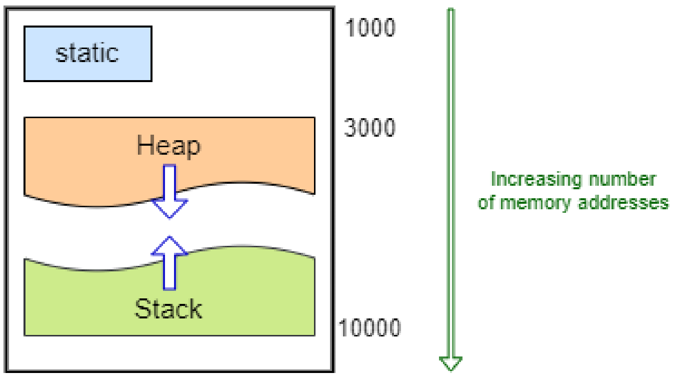

Figure 1.3: Structure of the program.

Chapter 2

# Network services in C

## 2.1 Application layer

We need IP protocol to use Internet. In this protocol, level 5 and 6 are hidden in Application Layer.
In this case, Application Layer needs to interact with Transport Layer, that is implemented in OS Kernel (Figure 2.1). Hence Application and Transport can talk each other with System Calls.

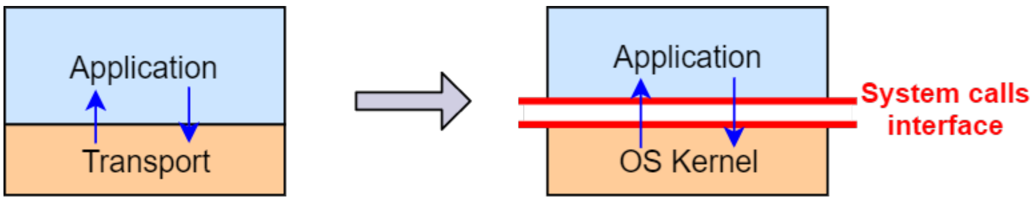

Figure 2.1: System calls interface.

## 2.2 socket

Entry-point (system call) that allow us to use the network services. It also allows application layer to access to level 4 of IP protocol.

```
#include <sys/types.h>
#include <sys/socket.h>

 int socket(int domain, int type, int protocol);\\
```

**RETURN VALUE** *File Descriptor (FD) of the socket*

*-1* if some error occurs and errno is set appropriately

(You can check value of errno including <errno.h>).

**domain** = *Communication domain*

protocol family which will be used for communication.

| | |
|---|---|
| **AF_INET:** | IPv4 Internet Protocol |
| **AF_INET6:** | IPv6 Internet Protocol |
| **AF_PACKET:** | Low level packet interface |

**type** = *Communication semantics*

| | |
|---|---|
| **SOCK_STREAM:** | Provides sequenced, reliable, two-way, connection-based bytes stream. An OUT-OF-BAND data mechanism may be supported. |
| **SOCK_DGRAM** | Supports datagrams (connectionless, unreliable messages of a fixed maximum length). |

**protocol** = *Particular protocol to be used within the socket*

Normally there is only a protocol for each socket type and protocol family (protocol=0), otherwise ID of the protocol you want to use

## 2.3 TCP connection

In TCP connection, defined by type **SOCK_STREAM** as written in the Section 2.2, there is a client that connects to a server. It uses three primitives (related to File System primitives for management of files on disk) that do these logic actions:

1. start (open bytes stream)
2. add/remove bytes from stream
3. finish (clos bytes stream)

TCP is used transfering big files on the network and for example with HTTP, that supports parallel download and upload (FULL-DUPLEX). The length of the stream is defined only at closure of the stream.

### 2.3.1 Client

#### 2.3.1.1 connect

The client calls **connect()** function, after **socket()** function of Section 2.2. This function is a system call that client can use to define what is the remote terminal to which he wants to connect.

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

**RETURN VALUE** *0* if connection succeds

*-1* if some error occurs and errno is set appropriately

**sockfd** = *Socket File Descriptor* returned by socket().

**addr** = *Reference to struct sockaddr*

sockaddr is a general structure that defines the concept of address.

In practice it's a union of all the possible specific structures of each protocol.

This approach is used to leave the function written in a generic way.

**addr** = *Length of specific data structure used.*

In the following there is the description of struct **sockaddr_in**, that is the specific sockaddr structure implemented for family of protocls **AF_INET**:

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};\\
```

The two addresses, needed to define a connection, are (see Figure 2.2):

- **IP address** (*$sin_addr$* in *sockaddr_in struct*)
  identifies a virtual interface in the network. It can be considered the entry-point for data arriving to the computer. *It's unique in the world.*
- **Port number** (*$sin_port$* in *sockaddr_in struct*)
  identifies to which application data are going to be sent. The port so must be open for that stream of data and it can be considered a service identifier. There are well known port numbers, related to standard services and others that are free to be used by the programmer for its applications (see Section 4.2 to find which file contains well known port numbers). *It's unique in the system.*

As mentioned in Section 1.1, network data are organized as Big Endian, so in this case we need to insert the IP address according to this protocol. It can be done as in previous example or with the follow function:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
```

The port number is written according to Big Endian architecture, through the next function:

```
#include <arpa/inet.h>

uint16_t htons(uint16_t hostshort);
```

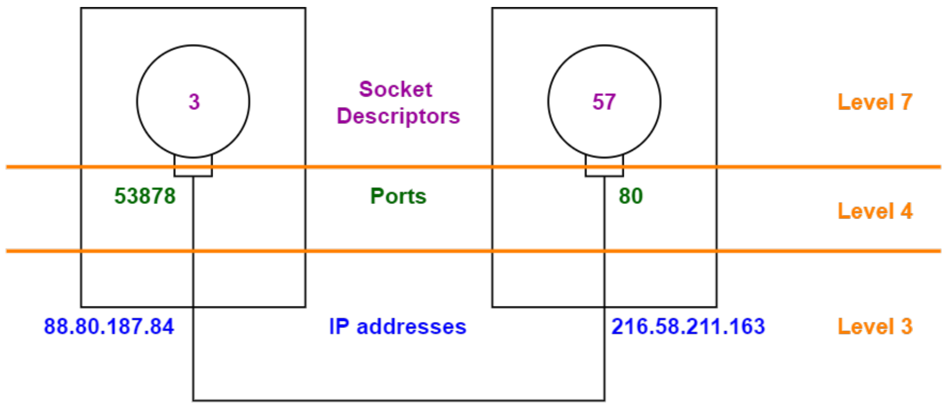

Figure 2.2: After successful connection.

### 2.3.1.2 write()

Application protocol uses a readable string, to excange readable information (as in HTTP). This tecnique is called simple protocol and commands, sent by the protocol, are standardized and readable strings.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

| | |
|---|---|
| **RETURN VALUE** | *Number of bytes written* on success |
| | *-1* if some error occurs and errno is set appropriately |
| **fd** = | *Socket File Descriptor* returned by socket(). |
| **buf** = | *Buffer of characters to write* |
| **count** = | *Max number of bytes to write* in the file (stream). |

The write buffer is usually a string but we don't consider the null value (\0 character), that determine the end of the string, in the evaluation of count (**strlen(buf)-1**). This convention is used because \0 can be part of characters stream.

### 2.3.1.3 read()

The client uses this blocking function to wait and obtain response from the remote server. Not all the request are completed immediat from the server, for the meaning of stream type of protocol. Infact in this protocol, there is a flow for which the complete sequence is defined only at the closure of it2.2.

**read()** is consuming bytes fom the stream asking to level 4 a portion of them, because it cannot access directly to bytes in Kernel buffer. Lower layer controls the stream of information that comes from the same layer of remove system.

```
#include <unistd.h>

        ssize_t read(int fd, void *buf, size_t count);
```

| | |
|---|---|
| **RETURN VALUE** | *Number of bytes read* on success |
| | *0* if EOF is reached (end of the stream) |
| | *-1* if some error occurs and errno is set appropriately |
| **fd** = | *Socket File Descriptor* returned by socket(). |
| **buf** = | *Buffer of characters in which it reads and stores info* |
| **count** = | *Max number of bytes to read* from the file (stream). |

So if **read()** doesn't return, this means that the stream isn't ended but the system buffer is empty.
If **read=0**, the function met EOF and the local system buffer is now empty. This helps client to understand that server ended before the connection.

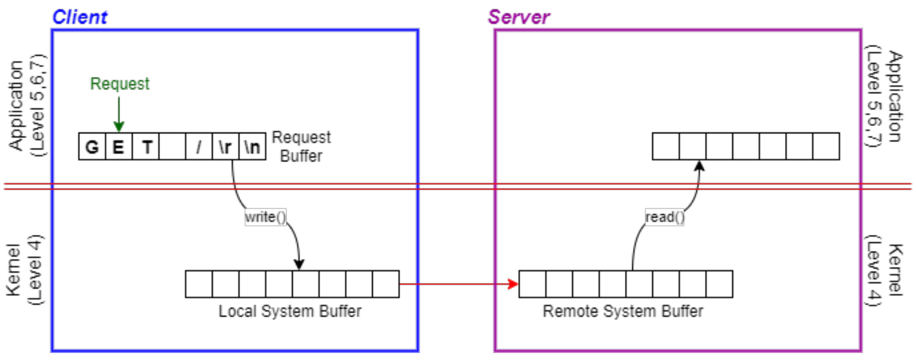

Figure 2.3: Request by the client.

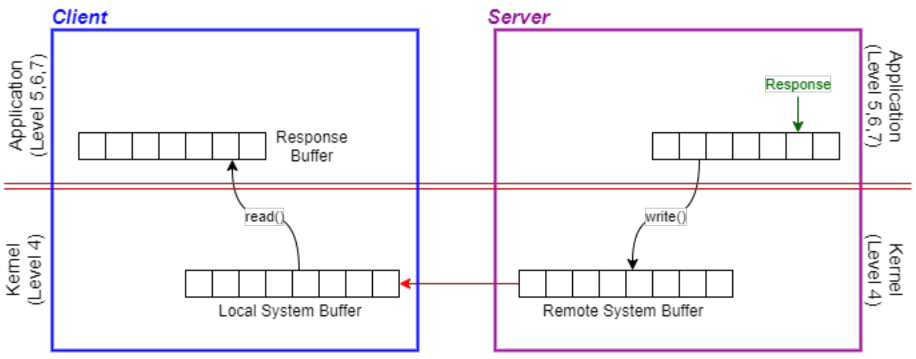

Figure 2.4: Response from the server.

## 2.4 UDP connection

UDP connection is defined by type **SOCK_DGRAM** as specified in Section 2.2. It's used for application in which we use small packets and we want immediate feedback directly from application. It isn't reliable because it doesn't need confirmation in transport layer. It's used in Twitter application and in video streaming.

# Chapter 3

# HTTP protocol

HTTP protocol was presented for the first time in the RFC 1945 (Request for Comment).
The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands).
It's not the first Hypertext protocol in history because there was Hypertalk, made by Apple before.
A feature of HTTP is the typing of data representation, allowing systems to be built independently f the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990.

## 3.1 Terminology

- **connection**
  a transport layer virtual circuit established between two application programs for the purpose of communication.
- **message**
  the basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in Section 4 and transmitted via the connection.
- **request**
  an HTTP request message.
- **response**
  an HTTP response message.
- **resource**
  a network data object or service which can be identified by a URI.
- **entity**
  a particular representation or rendition of a data resource, or reply from a service resource, that may be enclosed within a request or response message. An entity consists of metainformation in the form of entity headers and content in the form of an entity body.
- **client**
  an application program that establishes connections for the purpose of sending requests.
- **user agent**
  the client which initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.
- **server**
  an application program that accepts connections in order to service requests by sending back responses.

- **origin server**
  the server on which a given resource resides or is to be created.

- **proxy**
  an intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy must interpret and, if necessary, rewrite a request message before forwarding it. Proxies are often used as client-side portals through network firewalls and as helper applications for handling requests via protocols not implemented by the user agent.

- **gateway**
  a server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.
  Gateways are often used as server-side portals through network firewalls and as protocol translators for access to resources stored on non-HTTP systems.

- **tunnel**
  a tunnel is an intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed.
  Tunnels are used when a portal is necessary and the intermediary cannot, or should not, interpret the relayed communication.

- **cache**
  a program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cachable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server may include a cache, though a cache cannot be used by a server while it is acting as a tunnel.

Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

## 3.2 Basic rules

The following rules are used throughout are used to describe the grammar used in the RFC 1945.

| | | |
|---|---|---|
| **OCTET** | = | <any 8-bit sequence of data> |
| **CHAR** | = | <any US-ASCII character (octets 0 - 127)> |
| **UPALPHA** | = | <any US-ASCII uppercase letter "A".."Z"> |
| **LOALPHA** | = | <any US-ASCII lowercase letter "a".."z"> |
| **ALPHA** | = | UPALPHA \| LOALPHA |
| **DIGIT** | = | <any US-ASCII digit "0".."9"> |
| **CTL** | = | <any US-ASCII control character (octets 0 - 31) and DEL (127)> |
| **CR** | = | <US-ASCII CR, carriage return (13)> |
| **LF** | = | <US-ASCII LF, linefeed (10)> |
| **SP** | = | <US-ASCII SP, space (32)> |
| **HT** | = | <US-ASCII HT, horizontal-tab (9)> |
| <"> | = | <US-ASCII double-quote mark (34)> |

## 3.3 Messages

### 3.3.1 Different versions of HTTP protocol

- **HTTP/0.9 Messages**
  Simple-Request and Simple-Response do not allow the use of any header information and are limited to a single request method (GET).
  Use of the Simple-Request format is discouraged because it prevents the server from identifying the media type of the returned entity.

```
HTTP-message = Simple-Request | Simple-Response
```

```
Simple-Request  = "GET" SP Request-URI CRLF


Simple-Response = [ Entity-Body ]
```

- **HTTP/1.0 Messages**
  Full-Request and Full-Response use the generic message format of RFC 822 for transferring entities. Both messages may include optional header fields (also known as "headers") and an entity body. The entity body is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF).

```
HTTP-message = Full-Request | Full-Response
```

```
Full-Request = Request-Line
               *(General-Header | Request-Header | Entity-Header)
               CRLF
               [Entity-Body]


Full-Response = Status-Line
                *(General-Header | Request-Header | Entity-Header)
                CRLF
                [Entity-Body]
```

### 3.3.2 Headers

The order in which header fields are received is not significant. However, it is "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Entity-Header fields.
Multiple HTTP-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list.

```
HTTP-header = field-name ":" [ field-value ] CRLF
```

### 3.3.3 Request-Line

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Method          = "GET" | "HEAD" | "POST" | extension-method

extension-method = token
```

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource.
Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

### 3.3.4 Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

```
Request-URI = absoluteURI | abs_path
```

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Expires header field.
Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address.

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted.

### 3.3.5 Request Header

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server.
These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method (procedure) invocation.

```
Request-Header = Authorization | From | If-Modified-Since | Referer | User-Agent
```

### 3.3.6 Status line

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

**General Status code**

| | |
|---|---|
| **1xx: Informational** | Not used, but reserved for future use |
| **2xx: Success** | The action was successfully received, |
| | understood, and accepted. |
| **3xx: Redirection** | Further action must be taken in order to complete the request |
| **4xx: Client Error** | The request contains bad syntax or cannot be fulfilled |
| **5xx: Server Error** | The server failed to fulfill an apparently valid request |

**Known service code**

| | |
|---|---|
| **200** | OK |
| **201** | Created |
| **202** | Accepted |
| **204** | No Content |
| **301** | Moved Permanently |
| **302** | Moved Temporarily |
| **304** | Not Modified |
| **400** | Bad Request |
| **401** | Unauthorized |
| **403** | Forbidden |
| **404** | Not Found |
| **500** | Internal Server Error |
| **501** | Not Implemented |
| **502** | Bad Gateway |
| **503** | Service Unavailable |

## 3.4 Examples

The following pieces of code are examples of TCP client connection to **www.google.it**, using functions explained in Chapter 2.

#### 3.4.0.1 HTTP 0.9

The following piece of code define a structure, used to connect to Google server.

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

//Identifies the address we want to connect to
struct sockaddr_in server;

int main()
{
    /*
     * Creation of socket = file descriptor for the Socket
     *                      (number of index in ufdt)
     */
    int size;
    int s = socket(AF_INET, SOCK_STREAM, 0);
    char request[100], response[1000000];

    if(s == -1)
    {
        perror("Socket Failed\n");
        return 1;
    }

    /*
     * Extablish the connection to www.google.it
     */

    //Family of addresses (IPv4 addresses)
    server.sin_family = AF_INET;

    //http service port = 80
    server.sin_port = htons(80);

    //Definition of IP address of google
    unsigned char ip_addr[4] = {216, 58, 208, 163};
    server.sin_addr.s_addr = *(unsigned int*) ip_addr;
    int t = connect(s, (struct sockaddr *) &server, sizeof(server));

    if(t==-1)
    {
        perror("Connection error\n");
        return 1;
    }

    /*
     * Send a Request (Application Layer = HTTP)
     */
    sprintf(request, "GET / HTTP/1.0\r\n\r\n");

    for(size=0; request[size]; size++);
    t = write(s, request, size);

    if(t==-1)
    {
        perror("Write failed\n");
        return 1;
    }

    /*
     * Receive the response (HTML page)
     * 1000000=MAX length
     * 1000000-size ---> guarantees that the max amount of characters read is 1000000
     */
    for(size=0; (t=read(s, &response[size], 1000000-size))>0; size=size+t);

    //Print the value of the response message
    int i;
    for(i=0; i<size; i++)
        printf("%c", response[i]);
```

Listing 3.1: web_client.c

The most important thing is that **socket()** is entry-point for level 4, but also **connect()** is the request to Kernel to extablish the connection.
**read()** and **write()** are system calls used respectively to obtain result(response) of a request and to generate request.
These function permit us to ask to lower level to do this things, without knowing content of system buffers (stream). The second part is only used to read the input.

#### 3.4.0.2 HTTP 1.0

#### 3.4.0.3 HTTP 1.1

# Chapter 4

# Shell

## 4.1 Commands

| | | |
|---|---|---|
| **man** man | | Shows info about man command and lists all the sections of the manual. |
| **strace** objFile | | Lists all the system calls used in the program. |
| **gcc** -o objFile source **-v** | | Lists all the path of libraries and headers used in creation of objFile. |
| **netstat** | -t | Lists all the active TCP connections showing domain names. |
| | -u | Lists all the active UDP connections showing domain names. |
| | -n | Lists all the active, showing IP and port numbers. |
| **nslookup** domain | | Shows the IP address related to the domain (E.g. IP of www.google.it) |
| **wc** [file] | | Prints in order newline, word, and byte counts for file if file not specified or equal to -, counts from stdin. |

## 4.2 Files

| | |
|---|---|
| **/etc/services** | List all the applications with their port and type of protocol (TCP/UDP). |
| **/usr/include/x86_64-linux-gnu/bits/socket.h** | List all the protocol type possible for socket. |
| **/usr/include/x86_64-linux-gnu/sys/socket.h** | Definition of struct sockaddr and specific ones. |

## 4.3 vim

### 4.3.1 .vimrc

In this section there will be shown the file **.vimrc** that can be put in the user home (∼ or **$HOME** or –) or in the path **/usr/share/vim/** to change main settings of the program.

```
syntax on
set number
filetype plugin indent on
set tabstop=4
set shiftwidth=4
set expandtab
set t_Co=256
```

Listing 4.1: .vimrc

### 4.3.2 Shortcuts

**Main**

| | |
|---|---|
| **Esc** | Gets out of the current mode into the “command mode”.<br>All keys are bound of commands |
| **i** | “Insert mode”<br>for inserting text. |
| **:** | “Last-line mode”<br>where Vim expects you to enter a command. |

**Navigation keys**

| | |
|---|---|
| **h** | moves the cursor one character to the left. |
| **j** or **Ctrl** + **J** | moves the cursor down one line. |
| **k** or **Ctrl** + **P** | moves the cursor up one line. |
| **l** | moves the cursor one character to the right. |
| **0** | moves the cursor to the beginning of the line. |
| **$** | moves the cursor to the end of the line. |
| **ˆ** | moves the cursor to the first non-empty character of the line |
| **w** | move forward one word (next alphanumeric word) |
| **W** | move forward one word (delimited by a white space) |
| **5w** | move forward five words |
| **b** | move backward one word (previous alphanumeric word) |
| **B** | move backward one word (delimited by a white space) |
| **5b** | move backward five words |
| **G** | move to the end of the file |
| **gg** | move to the beginning of the file. |

**Navigate around the document**

| | |
|---|---|
| **h** | moves the cursor one character to the left. |
| ( | jumps to the previous sentence |
| ) | jumps to the next sentence |
| { | jumps to the previous paragraph |
| } | jumps to the next paragraph |
| [[ | jumps to the previous section |
| ]] | jumps to the next section |
| [] | jump to the end of the previous section |
| ][ | jump to the end of the next section |

**Insert text**

| | |
|---|---|
| **h** | moves the cursor one character to the left. |
| **a** | Insert text after the cursor |
| **A** | Insert text at the end of the line |
| **i** | Insert text before the cursor |
| **o** | Begin a new line below the cursor |
| **O** | Begin a new line above the cursor |

**Special inserts**

| | |
|---|---|
| **:r** [filename] | Insert the file [filename] below the cursor |
| **:r !**[command] | Execute [command] and insert its output below the cursor |

**Delete text**

| | |
|---|---|
| **x** | delete character at cursor |
| **dw** | delete a word. |
| **d0** | delete to the beginning of a line. |
| **d$** | delete to the end of a line. |
| **d)** | delete to the end of sentence. |
| **dgg** | delete to the beginning of the file. |
| **dG** | delete to the end of the file. |
| **dd** | delete line |
| **3dd** | delete three lines |

**Simple replace text**

| | |
|---|---|
| **r**}text} | Replace the character under the cursor with {text} |
| **R** | Replace characters instead of inserting them |

**Copy/Paste text**

| | |
|---|---|
| **yy** | copy current line into storage buffer |
| **["x]yy** | Copy the current lines into register x |
| **p** | paste storage buffer after current line |
| **P** | paste storage buffer before current line |
| **["x]p** | paste from register x after current line |
| **["x]P** | paste from register x before current line |

**Undo/Redo operation**

| | |
|---|---|
| **u** | undo the last operation. |
| **Ctrl+r** | redo the last undo. |

**Search and Replace keys**

| | |
|---|---|
| **/search_text** | search document for search_text going forward |
| **?search_text** | search document for search_text going backward |
| **n** | move to the next instance of the result from the search |
| **N** | move to the previous instance of the result |
| **:%s/original/replacement** | Search for the first occurrence of the string “original” and replace it with “replacement” |
| **:%s/original/replacement/g** | Search and replace all occurrences of the string “original” with “replacement” |
| **:%s/original/replacement/gc** | Search for all occurrences of the string “original” but ask for confirmation before replacing them with “replacement” |

**Bookmarks**

| | |
|---|---|
| **m {a-z A-Z}** | Set bookmark {a-z A-Z} at the current cursor position |
| **:marks** | List all bookmarks |
| **'{a-z A-Z}** | Jumps to the bookmark {a-z A-Z} |

**Select text**

| | |
|---|---|
| **v** | Enter visual mode per character |
| **V** | Enter visual mode per line |
| **Esc** | Exit visual mode |

**Modify selected text**

| | |
|---|---|
| | Switch case |
| **d** | delete a word. |
| **c** | change |
| **y** | yank |
| > | shift right |
| < | shift left |
| **!** | filter through an external command |

**Save and quit**

| | |
|---|---|
| **:q** | Quits Vim but fails when file has been changed |
| **:w** | Save the file |
| **:w** new_name | Save the file with the new_name filename |
| **:wq** | Save the file and quit Vim. |
| **:q!** | Quit Vim without saving the changes to the file. |
| **ZZ** | Write file, if modified, and quit Vim |
| **ZQ** | Same as :q! Quits Vim without writing changes |

### 4.3.3 Multiple files

- **Opening many files in the buffer**

```
vim file1 file2
```

Launching this command, you can see only one file at the same time. To jump between the files you can use the following vim commands:

| | |
|---|---|
| **n(ext)** | jumps to the next file |
| **prev** | jumps to the previous file |

- **Opening many files in several tabs**

```
vim -p file1 file2 file3
```

All files will be opened in tabs instead of hidden buffers. The tab bar is displayed on the top of the editor. You can also open a new tab with file *filename* when you're already in Vim in the normal mode with command:

```
:tabe filename
```

To manage tabs you can use the following vim commands:

| | |
|---|---|
| **:tabn[ext]** (command-line command) | Jumps to the next tab |
| **gt** (normal mode command) | |
| **:tabp[revious] (command-line command)** | Jumps to the previous tab |
| **gT** (normal mode command) | |
| **ngT** (normal mode command) | Jumps to a specific tab index<br>n= index of tab (starting by 1) |
| **:tabc[lose]** (command-line command) | Closes the current tab |

- **Open multiple files splitting the window**
  *splits the window horizontally*

```
vim -o file1 file2
```

You can also split the window horizontally, opening the file *filename*, when you're already in Vim in the normal mode with command:

```
:sp[lit] filename
```

*splits the window vertically*

```
vim -O file1 file2
```

You can also split the window vertically, opening the file *filename*, when you're already in Vim in the normal mode with command:

```
:vs[plit] filename
```

Management of the windows can be done, staying in the normal mode of Vim, using the following commands:

| | |
|---|---|
| **Ctrl+w <cursor-keys>** | Jumps between windows |
| **Ctrl+w [hjkl]** | |
| **Ctrl+w Ctrl+[hjkl]** | |
| **Ctrl+w w** | Jumps to the next window |
| **Ctrl+w Ctrl+w** | |
| **Ctrl+w W** | Jumps to the previous window |
| **Ctrl+w p** | Jumps to the last accessed window |
| **Ctrl+w Ctrl+p** | |
| **Ctrl+w c** | Closes the current window |
| **:clo[se]** | |
| **Ctrl+w o** | Makes the current window the only one and closes all other ones |
| **:on[ly]** | |