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.

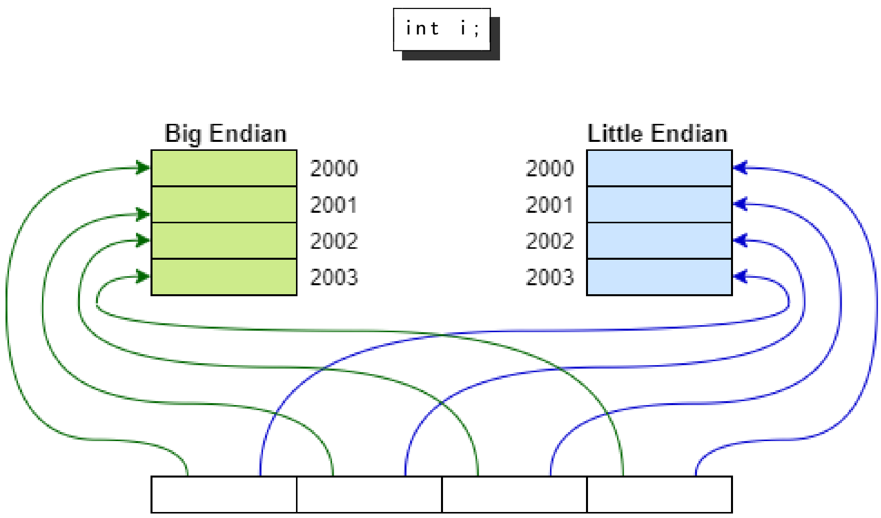


Figure 1.1: Little Endian and Big Endian.

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).

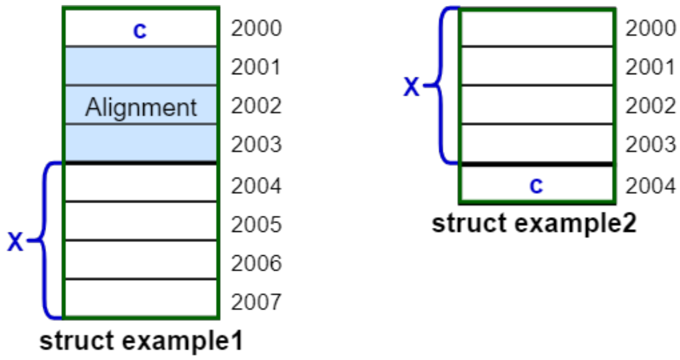
signed	unsigned
int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

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;
}                              }
```



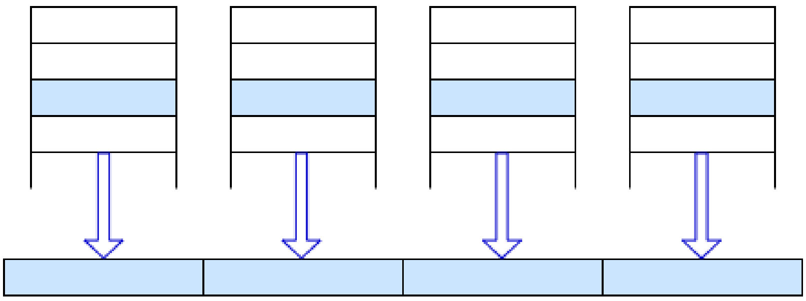


Figure 1.2: Parallel reading in one istruction in 32 bits architecture.

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

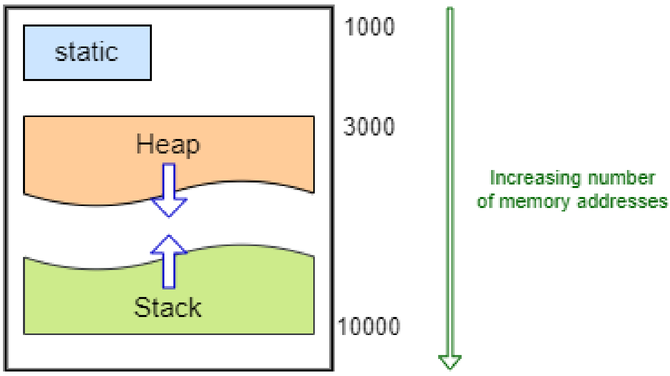


Figure 1.3: Structure of the program.

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.

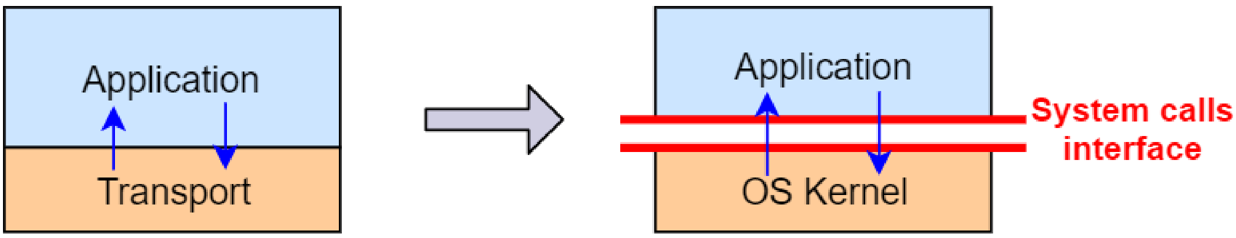


Figure 2.1: System calls interface.

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 */
};\\
```

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);
```

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.

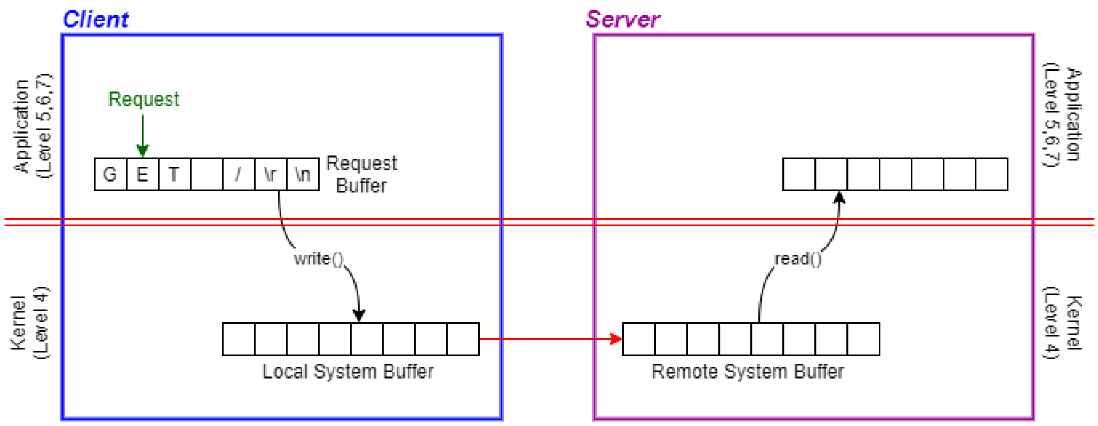


Figure 2.2: Request by the client.

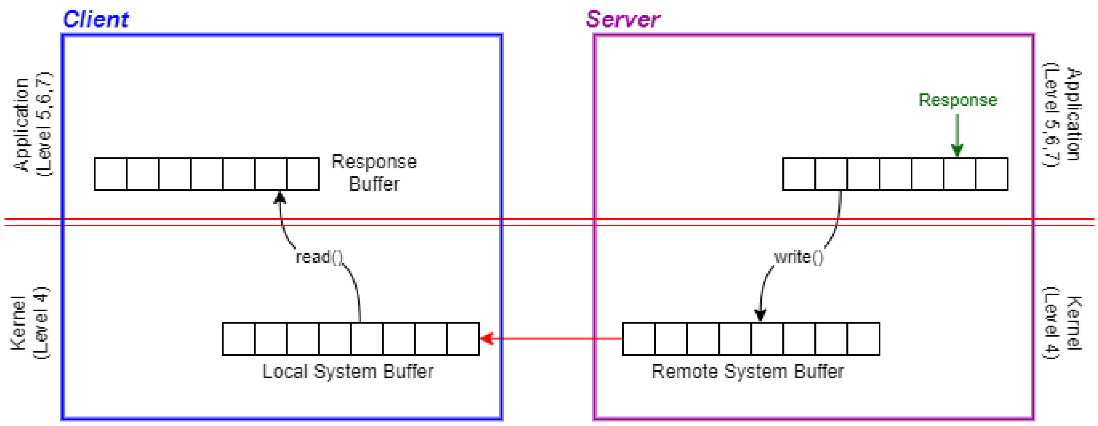


Figure 2.3: Response from the server.

2.3.1.4 Client connection to google

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 = HTTPS)
     */
    sprintf(request, "GET /\r\n");
    t = write(s, request, 7);

    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 2.1: 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).

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

Shell

3.1 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.

3.2 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.

3.3 vim

3.3.1 .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 3.1: .vimrc

3.3.2 Shortcuts

Main

Esc	Gets out of the current mode into the "command mode". All keys are bound of commands
i	"Insert mode" for inserting text.
:	"Last-line mode" 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.
^	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