Sockets API in C

Before next lab session, you can optionally deliver (in groups of two maximum) the printed code of the client and server where you must point out the system calls of this code, what they do and what relation they have with Sockets.

TCP/IP defines an end-to-end protocol, but it does not define an API with system calls. The use of TCP/IP depends on both the operating systems and the hardware (e.g. network card).

The sockets API (BSD Linux) was created to facilitate the programming and become a facto standard. This API uses system calls such as open, read, write to send streams similar to the file management.

The communication is established through sockets. When a socket is opened, an integer number (int) is returned. This number is used as reference to such socket like a file descriptor.

Network Byte sorting and conversions

Not all computers store the bytes in the same order, there are two ways: big-end first (Big endian) and little-end (Little endian) first. That is:

- Big Endian high-order byte is stored first, so the first byte is the biggest.
- Little Endian> low-order byte is stored first, so the first byte is the smallest.

To allow machines with different byte order conventions communicate with each other, the Internet protocols specify a canonical byte order convention for data transmitted over the network. So it is needed to convert two data types: short and long. For that, there are functions that perform such conversions:

- htons -> Host to Network Short;
- htonl() -> Host to Network Long,
- ntohs() -> Network to Host Short,
- ntohl() -> Network to Host Long

The IP addresses (`sin_addr`) and ports (`sin_port`) in the sockaddr_in structure must be "Network Bytes Orders", therefore, these functions must be used.

Structures

A structure sockaddr_in is used to describe a socket:

```
struct sockaddr_in {
 u_char sin_len; /* total length */
 u_short sin_family; /* AF_INET: address family */
 u_short sin_port; /* port number */
 struct in_addr sin_addr; /* IP address(u_long) */
 char sin_zero; /* not used (0) */
}
```

Where `sin_len` is the size of the structure when filled, `sin_family` indicates the address family to use (because sockets were created when different network protocols existed so it is required to specify the address family); in our case it is IP (AF_INET). `sin_port` indicates the port associated to that end. If it is a server (passive sock), the port indicates the listening port for the passive sock. If it is a client (active sock) then it indicates the port to which it connects. `sin_addr` is the IP address associated to the socket and `sin_zero` is not used.

To initialize this structure (client or server) we will use the following code segment, supposing that we have two variables `host` (host name or IP address) and `port` (port number):

```
char    *host = "163.117.141.197";
char    *port = "80";
struct sockaddr_in sin; /* an Internet endpoint address     */
struct hostent  *phe;
```

You can see more information about `sockaddr_in` structure in the Linux manual page ip(7) and about the 'hostent' structure in the Linux manual page gethostbyname.

The first step is to erase the structure with zeros. Afterwards, the structure is filled as is shown next:

```
memset(&sin, 0, sizeof(sin)); //erase with zeros
/* AF_INET is the address family that is used for the socket we are creating
 * (in this case an Internet Protocol address) */
sin.sin_family = AF_INET;
/* Assign port number to connect to (client) or bind (server) */
sin.sin_port = htons((u_short)atoi(port));
/* Map host name to IP address, allowing for dotted decimal
 * using this way to assign the IP address, we allow to use
 * either the IP address or the host name. gethostbyname will
 * make a DNS request if necessary to find out the IP address */
if ( phe = gethostbyname(host) )
  memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
  errexit("can't get \"%s\" host entry\n", host);
```

In the code, you can see that after erasing the structure, `AF_INET` is defined for the address family and the port is defined in network format. atoi is a function to convert ascii to integer (ascii to integer -> a to i), that is, string "80" to number 80.

Then, `gethostbyname(host)` is called. This function transforms the text "163.117.141.197" in a sequence of four integer numbers `163 117 141 197`, which is a IP address. If a host name was provided, then a DNS request (it will be seen later) will be performed in order to find the IP address. For instance, if the host name was "pervasive.it.uc3m.es", a DNS request would be performed to obtain the record A:

```
dig A pervasive.it.uc3m.es
```

In addition, it could be initialized in the following way (`inet_addr()` function converts an IP address into `unsigned long int`):

```
dest.sin_addr.s_addr = inet_addr("195.65.36.12");
```

Or it could be converted in opposite sense using `inet_ntoa()` that converts an IP address structure into `char *`:

```
char *ip;
ip=inet_ntoa(dest.sin_addr);
printf("La dirección es: %s\n",ip);
```

Differences between client and server in the sockets API

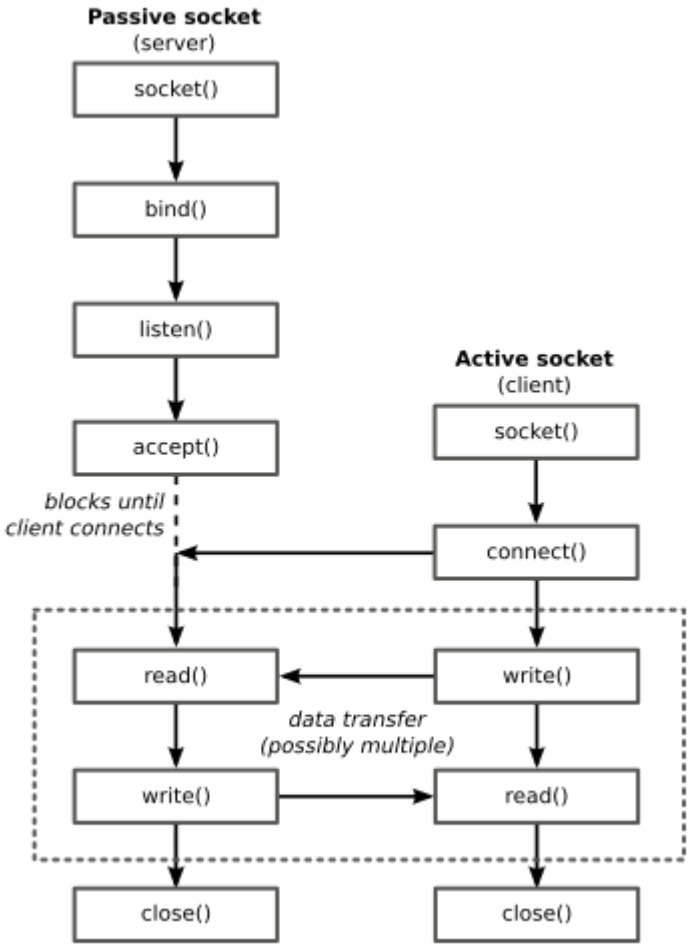
The client sockets or active sockets allow to connect to a server and use the `read()` and `write()` methods to read and write from the connection.

The server sockets or passive sockets allow to bind an address and to wait incoming connections (listen). Depending on the server criteria, the connections could be accepted or not.

If the incoming connection is accepted on the server, `accept()` function is called and it will return an active socket that allow write or read from it. Meanwhile, the server returns to listen/accept more connections. This is the purpose of a server: serving many clients (think of a web server like www.facebook.es that allows many people to connect simultaneously). So:

- A passive socket allows only listening to and accepting incoming connections.
- Read and write operations cannot be performed in a passive socket.
- When a connection is accepted, an active socket is created that represents the client connection in the server:
    - there is one passive socket in the server
    - there will be as many active sockets as connected clients
    - if the server has to read or write data to/from client, it will use the active socket associated to such client

In the following image you can see the calls required to create a passive socket and an active socket:



Active Sockets

Active sockets are the ones which initiate the TCP connection. Typically used by a client who knows the address and port of the other end (the passive socket typically at the server side). The syscall `socket()` creates a socket which will be specialized later into active or passive. When we invoke the syscall `connect()` on a socket, the socket becomes an active socket, and it will try to connect to the specified endpoint. TCP will take charge of initializing the sequence number as we saw in the theory class.

Once the connection is been created, the socket can be used to send/receive data to/from the other end, using the syscalls `read()`/`recv()` and `write`/`send()`

Passive Sockets

Passive sockets are typically server sockets. They are used to wait for incoming connections and to create a new connected socket if the connection is accepted.

Examples of clients and servers

In this repository you will find an example of a client and a server. You can find them in the files helloClient.c and helloServer.c

Download the code

Lets start by downloading the example code. The URL of the repository is https://gitlab.gast.it.uc3m.es/aptel/intro-sockets.git

To download the source code, open a command interpreter (terminal or similar) and execute the following command:

```
git clone https://gitlab.gast.it.uc3m.es/aptel/intro-sockets.git
```

If git complains about a SSL or TLS certificate problem just try the following:

```
git -c http.sslVerify=false clone https://gitlab.gast.it.uc3m.es/aptel/intro-sockets.git
```

That will create the following directory structure at the directory where you executed the command:

```
└── intro-sockets
    ├── README.md
    └── src
        ├── helloClient.c
        └── helloServer.c
```

Inside the `src` folder are the files with the code of the client and the server.

Socket System calls

Now we show the main syscalls of the sockets API so you can use them as reference from now on. All of them show the header files (.h) that should be included for the compiler to correctly assign types and resolve dependencies.

socket()

This syscall creates a socket which is not yet specialized into active or passive.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain,int type,int protocol);
```

- domain: we use AF_INET (for using ARPA protocols, i.e. Internet). AF_UNIX (defines sockets for internal communication within the host). Those are the most used. IPv6 can be used by specifying the domain AF_INET6.
- type: is the class of sockets we want to use (STREAM=flow-oriented or DGRAM=datagram sockets)

Property	Stream (TCP)	Datagram (UDP)
Reliable delivery	Yes	No
Message boundary	No (stream based, as a pipe)	Yes
Connection oriented	Yes	No

- protocol: set to 0 (the system will select the most appropriate (Stream=TCP, Datagram=UDP)

Socket returns a socket descriptor, an integer similar to the file descriptor we get with open(). The socket descriptor will be used for sending and receiving data. If the descriptor is -1 that signals an error.

bind()

Used by passive sockets to indicate the address (local network interface) where incoming connections are to be received.

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

int bind(int fd, struct sockaddr *my_addr, int addrlen);
```

- `fd`: socket descriptor returned by socket().
- `my_addr`: pointer to a sockaddr indicating the local address. INADDR_ANY allows to indicate all possible address.
- `addrlen`: length of the sockaddr structure pointed be my_addr. Set us sizeof(struct sockaddr).

bind() is used to specify a local address and port at the server where connections will be received. -1 is returned in case of errors (typically if the port is in use).

See the following example:

```
server.sin_port = 80;                 /* bind() uses the port 80 */
server.sin_addr.s_addr = INADDR_ANY; /*  server IP automatically  */
```

Note: port numbers less than 1024 are privileged and can only be used by the root (or administrator) user. Non-privileged users can use free ports between 1024 and 65535 (free means not already used by an existing socket).

connect()

Used by active sockets. It tries to establish a connection with a remote socket (server).

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

```
int connect(int fd, struct sockaddr *serv_addr, int addrlen);
```

- `fd`: socket descriptor returned by socket().
- `serv_addr`: pointer to a sockaddr indicating the other end (server) address and port.
- `addrlen`: sizeof(struct sockaddr).

-1 is returned in case of error, for instance no server listening at that address/port.

listen()

For passive sockets. Allows to indicate we want to receive incoming connections at the address/port specified in bind(). At the point we execute this syscall, incoming connections will be attended by the system.

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

int listen(int fd,int backlog);
```

- fd: socket descriptor returned by socket().
- backlog: place a limit to the number of incoming connections that the system is able to queue for accept.

After listen() we shall invoke accept(), that will block the execution flow of the process until an incoming connection. The previous figure shows the ordering of syscalls for passive sockets: socket() -> bind() -> listen() -> accept(). As in the other case it returns -1 in case of error.

accept()

For passive sockets. It accepts a connection on a socket. If no pending connections are present on the queue, and the socket is not marked as nonblocking, **accept**() blocks the caller until a connection is present.

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

int accept(int fd, void *addr, int *addrlen);
```

- `fd`: socket descriptor returned by socket().
- `addr`: pointer to a *sockaddr_in* structure that includes the connected client data.
- `addrlen`: length of the addr structure pointed by *addr. Set as sizeof(struct sockaddr_in).

It returns an active socket (socket descriptor) that represents the client connection. This is used to send and receive data to/from the client.

Remember that a passive socket only allows listen to and accept incoming connections.

Example:

```
sin_size=sizeof(struct sockaddr_in);

if ((asocket = accept(psocket,(struct sockaddr *)&client,&sin_size))==-1){
printf("accept() error\n");
exit(-1);
}
```

send() / write()

For active sockets (clients or sockets returned by the accept() function in the server)

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

int send(int fd, const void *msg, int len, int flags);
```

- `fd`: socket descriptor to send data.
- `msg`: pointer to the buffer that will be sent.
- `len`: data length (en bytes).
- `flags`: setup to 0.

Alternatively, you can use `read()` as if it were a file:

```
int read(socket, buffer, length)
```

`recv()` and `read()` return the number of bytes read into the buffer, or -1 if an error occurred.

Server

Open and read the server code (helloServer.c) in a text editor. Make sure that you understand all its lines. After reading, compile the code using `gcc`:

```
gcc -o helloServer helloServer.c
```

A `helloServer` file is created as result. Start the server:

```
./helloServer
Waiting for incomming connections at port 2222
Exit the program with Ctrl-C
```

Check the server is running using the command ps. Open another console and execute:

```
    ps aux
```

All the user processes in the machine are shown. To filter the result, pipeline the output of `ps aux` to `grep` and see the output of the command:

```
    ps aux | grep helloServer
    dds      31891  0.0  0.0   4080   656 pts/0    S+   13:50   0:00 ./helloServer
    dds      32224  0.0  0.0  12728  2108 pts/2    S+   14:36   0:00 grep
helloServer
```

The first row shows the process in the server. The first column corresponds to the user running the process. The second column is the process identifier and the end column shows the process name.

If you want to kill the process, use 'kill' and execute again 'ps' to check the server stops running:

```
    kill 31891
    ps aux | grep helloServer
    dds      32291  0.0  0.0  12728  2108 pts/2    S+   14:39   0:00 grep
helloServer
```

Start again the server. Check the process is running using `ps`.

Make sure that the server is listening to in the port 2222 using the command `netstat`:

```
    netstat -putan | grep helloServer
    (Not all processes could be identified, non-owned process info
     will not be shown, you would have to be root to see it all.)
    tcp   0   0 0.0.0.0:2222      0.0.0.0:*   LISTEN      32313/helloServer
```

Netstat allows you to query network statistics. As you can see, it uses TCP (-t), is associated with all network addresses of the server (0.0.0.0), and port 2222. The socket is passive since it is in LISTEN state (only passive sockets can be). Additionally, we see the process identifier 32313 and the process name (helloServer) (-p).

You can check the available network interfaces in the machine using the command `ifconfig`:

```
    eth0      Link encap:Ethernet  HWaddr 90:e6:ba:f4:b2:7a
              inet addr:163.117.141.197  Bcast:163.117.141.255  Mask:255.255.255.0
              inet6 addr: fe80::92e6:baff:fef4:b27a/64 Scope:Link
              inet6 addr: 2001:720:410:1030:92e6:baff:fef4:b27a/64 Scope:Global
              UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
              RX packets:2502174 errors:0 dropped:0 overruns:0 frame:0
              TX packets:911063 errors:0 dropped:0 overruns:0 carrier:0
```

```
              collisions:0 txqueuelen:1000
              RX bytes:2107719829 (1.9 GiB)  TX bytes:331277365 (315.9 MiB)
              Memory:fbe20000-fbe3ffff

eth0:0    Link encap:Ethernet  HWaddr 90:e6:ba:f4:b2:7a
              inet addr:163.117.141.200  Bcast:163.117.141.255  Mask:255.255.255.0
              UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
              Memory:fbe20000-fbe3ffff

lo        Link encap:Local Loopback
              inet addr:127.0.0.1  Mask:255.0.0.0
              inet6 addr: ::1/128 Scope:Host
              UP LOOPBACK RUNNING  MTU:65536  Metric:1
              RX packets:1903000 errors:0 dropped:0 overruns:0 frame:0
              TX packets:1903000 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:0
              RX bytes:764861624 (729.4 MiB)  TX bytes:764861624 (729.4 MiB)
```

> The machine name (hostname) or the IP address of the machine you want to connect to. Machine names, or hostnames, are human-friendly aliases that correspond to the IP address of a device connected to the network (the address would be much more difficult to remember). These aliases are translated to IP addresses using the Domain Name System (DNS) resolver service. Look at the label on your computer. You will find something like a name, let's say 'it001' which means that the machine's name is 'it001.lab.it.uc3m.es.' Just below, you will see the IP address. If not, you can find it using `ifconfig` as we have shown.

Client

Open and read the client code in a text editor. Make sure that you understand all its lines. After reading, compile the code using gcc:

```
gcc -o helloClient helloClient.c
```

A helloClient file is created as result. Start the client and write your name. The server answers "Hello 'your name'":

```
./helloClient
connecting to localhost
Dani
Hello Dani
```

In the console (or terminal) where the server is running you can see that a message indicating the client connection:

```
    ./helloServer
    Waiting for incomming connections at port 2222
     Exit the program with Ctrl-C
    Incomming connection from 127.0.0.1 remote port 36757
    Dani
    Waiting for incomming connections at port 2222
     Exit the program with Ctrl-C
```

TCP Traffic

In order to see the TCP traffic between client and server, open another terminal and use the command tcpdump:

```
    tcpdump -i lo port 2222
```

the parameter -i indicates the network interface (lo - localhost/loopback). port is used to filter only the traffic in the port 2222.

Start the server and another terminal execute the client. Don't write any message. Use 'netstat' to check the client and server states:

```
    netstat -putan | grep helloServer
    (Not all processes could be identified, non-owned process info
     will not be shown, you would have to be root to see it all.)
    tcp        0      0 0.0.0.0:2222            0.0.0.0:*               LISTEN
32313/helloServer
    tcp        0      0 127.0.0.1:2222          127.0.0.1:36761
ESTABLISHED 32313/helloServer
```

Now there are two sockets in the server: a passive socket (LISTEN) and another active socket (connected client) with state ESTABLISHED.

```
    netstat -putan | grep helloClient
    (Not all processes could be identified, non-owned process info
     will not be shown, you would have to be root to see it all.)
    tcp        0      0 127.0.0.1:36761         127.0.0.1:2222
ESTABLISHED 327/helloClient
```

The client, in its turn, has an active socket connected to the server. The ports must be consistent.

Write your name in the client and see the traffic in 'tcpdump'. Identify the connection establishment (*three-way handshake*), data exchange and connection termination:

```
    tcpdump -i lo port 2222
```

Connection establishment:

```
    14:52:49.367564 IP localhost.36760 > localhost.2222: Flags [S], seq
3680637885, win 43690, options [mss 65495,sackOK,TS val 148776448 ecr 0,nop,wscale
7], length 0
    14:52:49.367589 IP localhost.2222 > localhost.36760: Flags [S.], seq
1378612534, ack 3680637886, win 43690, options [mss 65495,sackOK,TS val 148776448
ecr 148776448,nop,wscale 7], length 0
    14:52:49.367611 IP localhost.36760 > localhost.2222: Flags [.], ack 1, win
342, options [nop,nop,TS val 148776448 ecr 148776448], length 0
```

Data exchange and termination:

```
    14:52:59.006007 IP localhost.36760 > localhost.2222: Flags [P.], seq 1:6, ack
1, win 342, options [nop,nop,TS val 148778858 ecr 148776448], length 5
    14:52:59.006137 IP localhost.2222 > localhost.36760: Flags [.], ack 6, win
342, options [nop,nop,TS val 148778858 ecr 148778858], length 0
    14:52:59.006171 IP localhost.2222 > localhost.36760: Flags [P.], seq 1:7, ack
6, win 342, options [nop,nop,TS val 148778858 ecr 148778858], length 6
    14:52:59.006195 IP localhost.2222 > localhost.36760: Flags [FP.], seq 7:12,
ack 6, win 342, options [nop,nop,TS val 148778858 ecr 148778858], length 5
    14:52:59.006261 IP localhost.36760 > localhost.2222: Flags [.], ack 7, win
342, options [nop,nop,TS val 148778858 ecr 148778858], length 0
    14:52:59.006381 IP localhost.36760 > localhost.2222: Flags [F.], seq 6, ack
13, win 342, options [nop,nop,TS val 148778858 ecr 148778858], length 0
    14:52:59.006395 IP localhost.2222 > localhost.36760: Flags [.], ack 7, win
342, options [nop,nop,TS val 148778858 ecr 148778858], length 0
```

Note: We recommend consulting the manual pages (`man`) of the commands used, `ps`, `netstat`, `tcpdump`, to learn more about the commands and the options used.