Sequential servers with sockets API

Support Materials

- On-line man pages: `socket(2)`, `socket(7)`, `send(2)`, `recv(2)`, `read(2)`, `write(2)`, `setsockopt(2)`, `fcntl(2)`, `select(2)`, `tcp(7)`, `ip(7)`.
- Guide to using sockets by Brian "Beej" Hall
- Tcpdump manual
- Chapters 6, 7 y 8 of "Linux Socket Programming" by Sean Walton, Sams Publishing Co. 2001
- Cheat Sheet file within this project

Problem statement

The sockets assignment is divided into the following three parts:

1. Sequential servers (echo client and server, socket options, analysis with tcpdump, file servers)
2. Concurrent servers (processes, threads).
3. Input/output (signal handlers, poll, select)

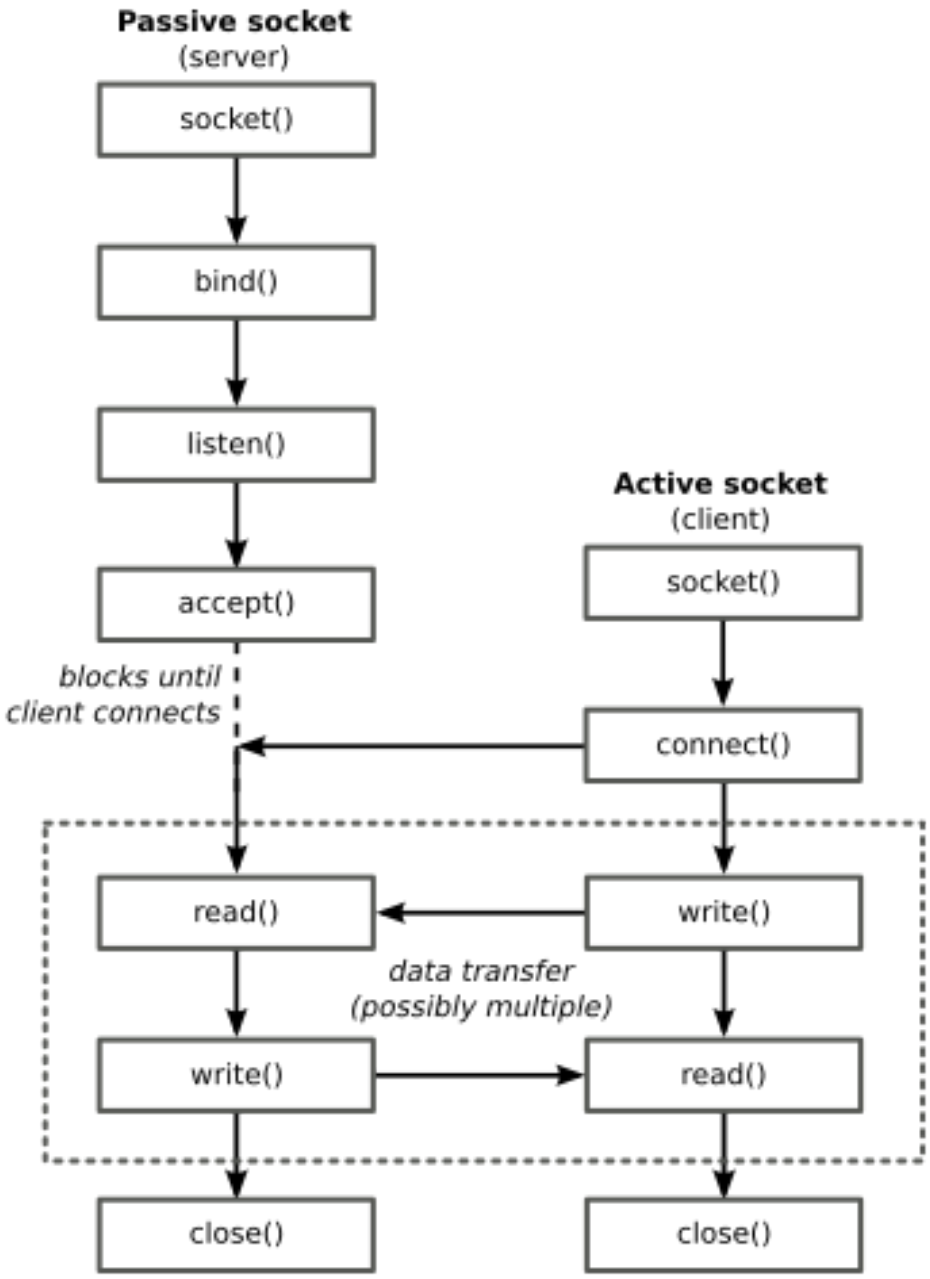
The **objective** of these labs is to understand and solve the questions that are presented below. It is not necessary to submit the answers unless you are interested in receiving feedback from the professor (via email).

Servidores secuenciales

Sequential servers are the simplest type of servers. They are generally used in services offered vía UDP (with sockets of type `SOCK_DGRAM`), but can also be found in TCP-based services (with sockets of type `SOCK_STREAM`).

You can review the difference between passive and active sockets as well as the API calls here: Intro-sockets. The life-cyle of an application using a passive socket is as follows:

1. allocate the socket (`socket`)
2. configure the socket (fill `sockaddr_in` structure)
3. create the socket (`open`),
4. associate the socket to a port (`bind`),
5. put the socket in passive mode to prepare for incoming connections (`listen`),
6. place the socket in the "waiting to handle incoming connections" state (`accept`), upon connection arrival, an active socket is generated
7. read/write (`read`/`write`) or receive/send (`recv`/`send`) on the active socket
8. close the active socket and free the connection (`close`),
9. return to the waiting for connections state.



In this practice, you have all the necessary files to test a sequential server (the code for a sequential echo server and an echo client is available). To use it, download the code with the following command:

```
git clone https://gitlab.gast.it.uc3m.es/aptel/sockets1_sequential_servers.git
```

There are two main files of code in the folder. The first is `EchoSever_seq.c`, that implements the server. The other is `EchoClient.c` that implements the client. The protocol they use is an ECHO service over TCP/IP in which the echo client connects to an echo server and whatever the client sends to the server will be sent back to the client.

Activities

1. Compile and execute the examples

> In the explanation of this practical assignment, we write port 8xxx to denote the result of adding 8000 to the last three digits of the IP address of the machine on which the server executes; this procedure ensures that there is no interference between different student groups.

Inside the downloaded folder, navigate to sockets1_sequential_servers/psockets1 using the following commands:

```
cd sockets1_sequential_servers
cd psockets1
```

Compile:

```
make clean
make
```

Then execute the server on port 8xxx:

```
./EchoServer_seq 8xxx
```

and in another window, start the client:

```
./EchoClient serverhost 8xxx
```

and see how it behaves.

> The `server host` parameter is either the hostname or the IP address of the server. Hostnames are human-readable nicknames that correspond to the address of a device connected to a network. This kind of hostname is translated into an IP address via the local hosts file, or the Domain Name System (DNS) resolver. Look to the label of your computer. You will find something like it001.lab.it.uc3m.es and an IP address. The first is the hostname, the second is the IP assigned to that hostname. You can find the IP of any machine using dig, for instance:

```
dig it001.lab.it.uc3m.es
;; ANSWER SECTION:
it001.lab.it.uc3m.es. 60 IN A 163.117.144.201
```

> Note every machine has at least two network interfaces: -IPv4 network standards reserve the entire 127.0.0.0/8 address block for loopback purposes. That means any packet sent to one of those 16,777,214 addresses (127.0.0.1 through 127.255.255.254) will be looped back. The device host file may have a line like: 127.0.0.1 localhost ::1 localhost -The real IP (in this case within the uc3m network) that starts with 163.117 (uc3m) so will be something like: 163.117.XXX.XXX

Have a look to the server code (`EchoServer_seq.c`) and the comments and question yourself about the following:

- which variable stores the passive socket?
- which stores the active socket?
- when do you get the active socket?

> To verify if your server is listening appropriately, execute your server and use netstat in the following way (example-look at red line-an * in address means all the addresses):

```
./EchoServer_seq 8765
Waiting for incomming connections at port 8765
Incomming connection from 127.0.0.1 remote port 52589
```

```
netstat -putan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 *:sunrpc *:* LISTEN
tcp 0 0 *:28017 *:* LISTEN
tcp 0 0 *:ssh *:* LISTEN
tcp 0 0 localhost:ipp *:* LISTEN
tcp 0 0 localhost:smtp *:* LISTEN
tcp 0 0 *:8765 *:* LISTEN
tcp 0 0 *:40645 *:* LISTEN
tcp 0 0 *:27017 *:* LISTEN
tcp 0 0 localhost:mysql *:* LISTEN
tcp 0 0 localhost:52589 localhost:8765 ESTABLISHED
tcp 0 0 localhost:8765 localhost:52589 ESTABLISHED
tcp6 0 0 [::]:sunrpc [::]:* LISTEN
tcp6 0 0 [::]:http [::]:* LISTEN
tcp6 0 0 [::]:ssh [::]:* LISTEN
tcp6 0 0 ip6-localhost:ipp [::]:* LISTEN
tcp6 0 0 ip6-localhost:smtp [::]:* LISTEN
tcp6 0 0 [::]:32769 [::]:* LISTEN
```

We are now going to observe the TCP connections with the tcpdump tool. We instruct the tool to report on the entire traffic passing via the local-loop interface (loopback) whose origin or destination port is 8xxx.

> You should understand that each machine has at least two network interfaces:
>
> - Network standards for IPv4 reserve the 127.0.0.0/8 segment for loopback. This means that any packet sent to one of these addresses (from 127.0.0.1 to 127.255.255.254) will be looped back to the source.
> - The actual IP address (in this case, within the university network) starting with 163.117 (uc3m) should look something like: 163.117.XXX.XXX.

2. Execute the following command in another window:

```
tcpdump -i lo port 8xxx
```

Press some keys at the client to see the echo.

```
tcpdump -i lo port 8888
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
16:28:57.553866 IP localhost.39363 > localhost.8888: Flags [P.], seq
470686013:470686018, ack 1031632982, win 513, options [nop,nop,TS val 2951382 ecr
2948837], length 5
16:28:57.554210 IP localhost.8888 > localhost.39363: Flags [.], ack 5, win 512,
options [nop,nop,TS val 2951382 ecr 2951382], length 0
16:28:57.554620 IP localhost.8888 > localhost.39363: Flags [P.], seq 1:6, ack 5,
win 512, options [nop,nop,TS val 2951382 ecr 2951382], length 5
```

Re-start the client and observe the exchange of frames with tcpdump (connection establishment, sending of data in both directions, acknowledgements and end of connection).

3. Start the client a couple of times and terminate it with CTRL-D or CTRL-C. Now kill the server with CTRL-C. What do you observe with tcpdump? Why?

Solve the problem. Do not proceed without resolving it (you will need to edit the source code).

> You may find it useful to observe the status of the client's network connections using the command: `netstat -tn` o `netstat -putan`

```
netstat -tn
THREEWAY HANDSHAKE
18:06:09.979299 IP monitor01.34998 > monitor01.8888: Flags [S],
18:06:09.979313 IP monitor01.8888 > monitor01.34998: Flags [S.]
18:06:09.979323 IP monitor01.34998 > monitor01.8888: Flags [.]
CLIENT SENDS FIN, SERVER ACKNOWLEDGE BUT DOES NOT CLOSE => NOT FIN
18:06:12.876977 IP monitor01.34998 > monitor01.8888: Flags [F.]
18:06:12.880514 IP monitor01.8888 > monitor01.34998: Flags [.]
```

> Server not executing close(). We add close(fd) at the end of TCPechod

4. Restart the (modified) server, start the client and kill the server without killing the client. What happens on re-starting the server? Why?

> In the netstat there is a FIN_WAIT(client stuck for a few seconds). Port 8888 used by that tcp connection so server cannot connect on reset

5. Kill the client, wait a few seconds then re-start the server. Now start two echo clients (on two different hosts) that connect to the same server. What happens if after starting the first

client but before asking for an echo, we start the second? Both clients send information. What happens?

> Both send the data, and get an ack from the server. But only one gets the echo back.

#The server can handle only ONE client at a time.

6. Now kill the second client and then kill the first. You should observe a RESET frame. Why?

> Because when first client is killed, server process the next connection in queue(second client), but second client is killed so the echoed fail and the server RST

7. Change the size of the connection queue (backlog of the socket) of the server, defined by the constant BACKLOG in the file EchoServer_seq, to 1. Now compile and re-start four echo clients from another host. What do you notice in tcpdump? Kill the last client to be started. What is the effect of doing so? What explanation can you give for this?

> Queue gets full with the third client so the fourth is deprecated

---

We are now going to use setsockopt to set socket options offered to the client. One such option is the choice of whether or not to use the Nagle algorithm. However, with the Linux kernel installed in the labs, the effect of setting this option is not visible. For this reason we will not experiment with it but will instead check the effect of setting two of the other possible options. For more information about the other socket options that can be set, consult the man pages `setsockopt(2)`, `socket(7)`, `ip(7)`, and `tcp(7)`.

8. Now allow to bind to a port that is already in use -by setting `SO_REUSEADDR` option (as long as there is no active listening socket already bound to it) with `setsockopt` function-. For that, uncomment the code lines in the `EchoServer_seq.c` y recompile the code. Observe the result of setting this option by revisiting question 4.

> Now we do not need to wait for client to definetly close, we can rebind to the same server port

```
int yes=1;
if (setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes, sizeof(int)) != 0) {
    perror("setsockopt");
    exit(1);
}
```

9. Modify the size of the segments sent by the client - by setting the option TCP_MAXSEG (which is defined in `<netinet/tcp.h>`)

> Cuando supera los 88 caracteres elegidos como máximo segment size, se divide el mensaje. Cuando el TCP_MAXSEG se baja a cierto umbral deja de funcionar la conexión y pone: client: failed to connect. Pasó con 58, 68, 78, 87 es el más cercano. En 88 está el threshold, no puede más bajo que ese para tener de máximo segmento que dividir.

```
int maxseg = 88;
if (setsockopt(sockfd, SOL_TCP, TCP_MAXSEG, &maxseg,sizeof(maxseg))==-1)
{…}
```

> It may happen, as it depends on the linux kernel version, that the minimum value of TCP_MAXSEG is different. It may also happen changing that parameter has no effect on the the loopbak interface (localhost) but it has for physical interfaces (eth0, eth1...)

Recompile and check the effect in a client writing texts longer than the specified length. **Is it possible to specify an MSS smaller than 88? If not, why not?**

Note: It is recommended to run the client on a docXXX machine, together with tcpdump on the same machine.