Initiation Réseaux
TP4 - Network Socket

1 Objective

1. Practice network socket.
2. Review client/server approach.

Evaluation: When you finish the subject or certain steps, ask me to check your work in order to evaluate your participation. You do **NOT** have a report to submit for this session.

2 Web server

In this lab, you will write a very simple Web server in C language. The connection between your server and a web browser will be made by a network socket over the TCP/IP protocol.

2.1 Skeleton

You can start with any text editor (prefer the one you are able to use) by coding a main function:

```c
#include <stdio.h>
#include <stdlib.h>

#define handle_error(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)

int main(int argc, char **argv)
{
  return EXIT_SUCCESS;
}
```

You could call your file `websrv.c` as an example. In this case, you could compile it by just typing:

```
make
```

In order to print all compilation warnings, produce standard and optimized code, you could write the following line in a file called `Makefile` in the current directory:

```
CFLAGS=-Wall -pedantic -O2

default: websrv

.PHONY: clean
clean:
  rm -rf websrv
```

Be careful, the recipes (*i.e.* `rm -rf websrv`) of a rule has to start with a tabulation (and not spaces).

Now, you can build your program just typing `make` (without any argument) since the default rule corresponds to the binary `websrv`.

2.2 Socket creation

The first step is to create a server socket using the `socket` function. The command `man 2 socket` gives you the following informations:

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

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

The `domain` corresponds to the network layer protocol (layer 3) and can take the values `AF_UNIX` for local communications, `AF_INET` for IPv4 protocols and `AF_INET6` for IPv6 protocols, among others. You use the _____ domain to create this socket.

The `type` corresponds to the type of transport layer protocol (layer 4) and can take the values `SOCK_STREAM` for stream protocol (*e.g.* TCP) and `SOCK_DGRAM` for datagram protocol (*e.g.* UDP), among others. You use the _____ type to create this socket.

The `protocol` corresponds to the protocol for the type previously selected. In your case, there is only one protocol per protocol type. Therefore this parameter can be safely set to `0`.

For this first function, consider the following code example:

```
1  int sockfd;
2
3  sockfd = socket(AF_???, SOCK_???, 0);
4  if (sockfd == -1)
5    handle_error("socket");
```

The variable `sockfd` represents the socket file descriptor you will use for this entire exercise. All the functions of this exercise return `-1` when they fail. You must test this return value to avoid to write a quiet program that silently crashes. In order to make that test, we define for you a macro in Section 2.1. This macro acts as a function and you can give it the name of the faulty socket function as a parameter.

2.3 Socket setting

The 1.5 step is to set an option to your socket to make it reusable. It is not mandatory but if you bypass this step, you will have problem when you will start your program twice ("already in use" for the second start). The command `man 2 setsockopt` gives you the following informations:

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

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t
optlen);
```

The parameter `sockfd` must be (as for all the next functions) the `sockfd` int variable that you previously declared.

The parameter `level` (respectively `optname`) must be set to `SOL_SOCKET` (respectively `SO_REUSEADDR`).

For the paremeter `optval`, you have to declare an `int` variable (*e.g.* `val`) and set it to `1`. As you have noticed, `optval` is a parameter of type `void *optval` (a generic pointer). Therefore, you have to pass the address of the variable `val` with the following syntax:

```
1  setsockopt(..., ..., ..., &val, ...);
```

The last parameter is the size this previous parameter. You have to pass `sizeof(int)` since your option has been declared of type `int`.

2.4 Socket address creation

The second step is to create a socket address to represent an IP address. This is the address of the machine on which the server will run. The command `man 7 ip` gives you the following informations:

```
#include <arpa/inet.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 */
}

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

You have to declare a variable (*e.g.* `addr`) of type `struct sockaddr_in` and set this 3 fields. If you don't remember how to deal with the structures, consider the following example:

```
1  /* EXAMPLE: how to use struct */
2  struct xyz /* structure definition */
3  {
4    int foo;
5    int bar;
6  };
7
8  struct xyz w; /* variable w of type 'struct xyz' */
9
10 w.foo = 1;
11 w.bar = 2;
```

The field `sin_family` corresponds to the family of address. As say in the comment of the manpage, it can be set to `AF_INET`.

The field `sin_port` corresponds to the port of the service that your program will serve. The default port for the Web service is _____. Be careful, the port must be passed in network byte order. This can be done using the function `htons` that simply takes the port number as a parameter and returns it in network byte order.

The field `sin_addr` is a structure itself. But this structure has just one field and you can set it to `INADDR_ANY`.

2.5 Socket binding

The third step is to bind the socket to an IP address in order to make it reachable from the IP network.

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

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

The parameter `sockfd` is always your very same socket file descriptor.

The parameter `addr` is a pointer on the address you previously declared. You can notice the type of the parameter is not exactly the same as the one used in the Section 2.4. Actually, it allows to make generic code (like with inheritance in Java). You can safely cast this parameter:

```
1  (struct sockaddr *) &addr
```

The parameter `addrlen` is just the size of the structure `sockaddr` actually used. You can pass `sizeof(struct sockaddr_in)`.

2.6 Socket listening

The fourth step is to make the socket listen to a new input connection. The command `man 2 listen` gives you the following informations:

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

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

The parameter `backlog` is useless in your case and you can safely set it to `0`.

2.7 Socket acceptance

The fifth step is to make the socket accept new connections. The command `man 2 accept` gives you the following informations:

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

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

The fields addr and addrlen are for the address of the client socket (from the distant machine) accepted for this connection. At this step, you don't need of this address and you can safely pass `NULL` to these two parameters.

The function `accept` is a system call that blocks your program until a new connection intends to establish.

2.8 Connection test

When you try to connect to your server program through the network (from another machine) using a web browser, your program quits. Actually, the connection has been accepted and the function `accept` returns the client socket file descriptor. Your server can use it to send a message to the client. Therefore you must store it in a new variable (*e.g.* `clifd`).

Now, you will surround your accept function call by an infinite `while` loop (a server program should never quit until it is interrupted). Add the following code to make your server program able to serve an "index.html" file through the HTTP protocol. :

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int clifd;
5  int file;
6  char buf[BUFSIZ];
7  int siz;
8
9  while (1)
10 {
11   /* accept() code here */
12
13   /* read client request, store it in 'buf' and return number of read char */
14   siz = read(clifd, buf, BUFSIZ);
15   /* DEBUG: write client request (stored in 'buf') to stdout(1) */
16   write(1, buf, siz);
17   /* open 'index.html' file in read-only mode*/
18   if ((file = open("index.html", O_RDONLY)) == -1)
19     handle_error("open");
20   /* print HTTP header response at start of buffer */
21   siz = sprintf(buf, "HTTP/1.1 200 OK\n\n");
22   /* read file, store it in 'buf' after the header */
23   siz += read(file, buf + siz, BUFSIZ);
24   /* DEBUG: write server response to stdout */
25   write(1, buf, siz);
26   /* write server response (header + content of file) to client socket */
27   write(clifd, buf, siz);
28   /* close client socket and 'index.html' file */
29   close(clifd);
30   close(file);
31 }
```