Complementary Information on the Project

Sophie Coudert
Télécom Paris

November 14, 2024

VERSION 0.

1 Client/Server and Sockets

Warning: this presentation assumes an IP address references a single machine (which could be a virtual machine). In systems, such as cloud systems, an IP address can reference multiple executions machines (also called *nodes*).

Roughly speaking, when a client wants to request a service from a server, it sends a message to an address `IP:port`, where `IP` is an "Internet Protocol" address (IPV4 or IPV6) that identifies the machine in which the service is located on the network, and `port` (a number) references an address of a service inside a machine. When a server starts on a machine at address `IP`, it asks the operating system to use a given port `p`. Then, when a message for port `p` reaches the operating system using address `IP:port`, the operating system forwards the message to the server software.

Clients also need an `IP:port` couple reserved in the computer it runs: this address allows a client to send messages to servers, and to wait for servers' answers.

Clients and Servers connect to ports using a syscall called *socket*. There are two main kind of sockets:

- connection/listening sockets associated to an address to which clients send connection requests. This connection mechanism is used by all kind of servers.
- communication/data sockets which offer full duplex communication and are used to exchange data between the client and the server. The exchanged data depends of the offered service.

The usual way to establish a connection is:

1. the servers "opens" (with a syscall) a listening socket at an `IP:port` address. It then waits for clients to connect at this address.
2. a client sends a connection request at the server address.
3. if the server accepts the connection (syscall *accept*), a (virtual) communication channel is created by the operating systems at both ends. This commucation channel consists in two bound communication sockets:
   - a client data socket on the client's machine (at some `IP_c:port_c` address you don't need to know). This is the socket a client can use to *read* (receiving data from the server) and *write* (sending data to the server).
   - a dual server data socket on the server's machine (at some `IP_s:port_s` address you don't need to know), which works similarly to the one of the client.

4. client and server exchange data related to the precise service the server offers, generally following some communication conventions/protocol associated to this service.

5. The client or the server closes the connection, using the *close* syscall.

In this project, you won't have to worry about points 1-3 above as we provide the client and server code for this. Your job is to implement point 4, at client and server side, in order to implement the Eurecom drive.

There are different kinds of sockets (not detailed), but all you need to know is that as the used sockets in this project are "stream" sockets (also known as *TCP* sockets), you can use them as ordinary files, i.e., you can use the *read* and *write* syscalls, just like you would do for files. Indeed, these sockets are considered as file descriptors (i.e., int values), similar to the ones you obtain when you use *open* to open a file on your system. Yet, there are three small differences:

- a return value equal to 0 while reading means that the connection has been closed (for files, the meaning is different...). Note that reading from a closed socket may also return -1 with error variable `errno` containing `ECONNRESET` (for example if the peer did not cleanly closed the socket).

- if the amount of read data is smaller than the expected one, it does not mean that the end of file has been reached but that not enough data has currenltly reached the client / server.

- writing to a closed socket causes a `SIGPIPE` signal, which (by default) makes the program exit. The instruction `signal(SIGPIPE, SIG_IGN);` inhibits this signal. After using this instruction, a `write` to a closed socket does not lead to an exit anymore, but it returns -1 (i.e., an error), and the error variable `errno` contains `EPIPE`.

Warning: the default behaviour of `read` is to block until at least one byte is available. So, if meanwhile the sender has crashed, the receiver is blocked forever.

As a summary, you have to:

- At server side, implement function:
  `void student_server(int channel, int argc, char *argv[])`
  where `channel` is the server data socket used to communicate with the client, using `read` and `write` system calls. `argc` and `argv[]` are the usual command line arguments: they are given as input to `student_server` to allow you to use them.

- At client side, implement function:
  `int student_client(int channel, int argc, char *argv[])`
  where `channel` is the client data socket used to communicate with the server `read` and `write` system calls. As for previous function, `argc` and `argv[]` are the usual command line arguments.
  Returning 0 from this function makes the client exit. Returning any other value makes the client restart.

Both functions are called just after the dual way communiation channel has been established, before any data exchange apart from the necessary messages to establish this communication channel. When `student_server` returns (i.e., if no error occured when the connected client leaves), the server closes the server data socket and waits for the next client. When `student_client` returns, the client closes the client data socket (thus, from the server point of view, the client "leaves"). If the returned value is 0, the program exits. Otherwise the client tries to reconnect and then calls `student_client` again.

2 Project Structure

The client and server code we provide (so you don't have to implement: it's done, please, give us a good grade!) look like:

student_server.h:
```
int student_server(int channel, int argc, char *argv[]);
```

student_client.h:
```
int student_client(int channel, int argc, char *argv[]);
```

server.c:
```
#include "lib/student_server.h"
main:
  create listening socket
  loop:
     print connection IP:port (stdout)
     wait for client connection
     ! client connects !
     -> establish channel
(*)  student_server(channel,...)
     close channel
```

client.c: (connection IP:port in command line)
```
#include "lib/student_client.h"
main:
   1. connect to server,
         i.e. obtain channel
(*)    x = student_client(channel,...)
       close channel
       if x is 0 then exit
       otherwise goto 1
```
Note: also exits if connection fails.

The two lines with an asterisk `(*)` are the ones where the functions that you have to implement are called. As you do not need to know the precise C code of the client and the server, they are not provided as source code but as two library `server.o` and `client.o`. Both have been placed in the `lib` directory. `student_server.h` and `student_client.h` are also provided in the `include` directory. Your job is to therefore to complete the `student_server.c` and `student_client.c` files in the `usrc` directory with your implementation of the service specified in the webpage of the course. Your code should look like:

usrc/student_server.c:
```
#include "../include/student_server.h"
... your code ...
void student_server(int channel, int argc, char *argv[]) {
   ... your code ...
}
```

usrc/student_client.c:
```
#include "../include/student_client.h"
... your code ...
int student_client(int channel, int argc, char *argv[]) {
   ... your code ...
}
```

The `Makefile` we provide is ready to compile your code: it handles dependencies in order to combine all components to produce the two executable files: `server` and `client`. This files are produced in directory `bin` (and linked in working directories `bin/EDserver` and `bin/EDclient`). Of course, if your code were to grow quite consequently, you could decompose your C file into several files/libraries. To know how to do this, you can read the provided short introduction to how to structure a C program (in `doc/Cstructuring.pdf`).

3 Provided Library, Example and Executable

3.1 Provided Functions

To help you, a small and easy-to-understand library provides functions for parsing client's commands, for printing packets headers (debug for both client and server), and for sorting directory content in string describing it.
Its header is `include/utilities.h` (do open this file and look at it). `lib/utilities.o` is automatically added to the client's executable file when running `make`).

The two following functions may help you:

- `int parse_commandline(...);`
  You can directly use this function in your code to parse the command line after getting it from the user. It checks whether commands are correct and then fills a structure that contains the command's arguments as strings. It returns 0 in case of error and a non zero value otherwise.
- `char *pkt_string(...);`
  This function is intended for debugging purpose as observing packet headers content may help you resolve bugs. The function returns a human readable string which describes the content of the header of the packet passed as parameter. Warning: any call to this function erases the content of the previous call.
- `int sort_dir(...);`
  You can directly use this function in your code to sort files w.r.t alphabetical order in a string describing a directory content. This may be useful to build the string returned by the server when responding to an ls command. This function returns 0 in case of error and a non zero value otherwise.

3.2 Provided Example

The skeletons for starting with `student_server.c` and `student_client.c` are given in the two boxes just before. Yet, `student_server.c` and `student_client.c` files we provide (in `usrc/`) contain an example that should help you understanding how to get some tasks done. Of course this example does not solve your project, but it shows how to handle packets: formatting them, sending and receiving them. In particular, it shows how to use the `parse_commandline` function.

Basically, in our example, the client parses a command line provided by the user and sends a corresponding packet to the server, which in turn prints them in its terminal. The format of packets is given as documentation in the source C files of this example. But beware, in this example, the format of packets is different from the one of your project.

To run this example:

- in the root directory of the project, type `make` to produce `client` and `server` in directory `bin`. Directories `bin/EDclient` and `bin/EDserver` can be used for testing and debugging. They contain a link to the executable binaries and some example files.
- in a first terminal, run the server: `./server` (in directory `bin/EDserver`)
- look at the IP:port information printed by the server
- in a second terminal, run the client: `./client IP port` (in directory `bin/EDclient`)
- enter commands in the client terminal and look at the corresponding information printed by the server in its terminal.

The way to compile and start your server and client shall be quite the same. Also, obvisouly, if you were to decompose your code into different source files, which is probably a good idea, you will have to accordingly update the `Makefile`.

3.3 Provided Client and Server Executables

Once you have replaced the example by your code, `make` should produce both an executable client and an executable server respecting the project specifications. Actually, we provide our executable files we have produced form our source code: they are located in `tools` directory, but of course without the source code... This may be very useful for debugging your code. Indeed, to debug your client, it is convenient to have a reliable server (our server). Reciprocally it is convenient to have a reliable client to debug your server.

Moreover, these two executable files have additional options and features, i.e., it contains features not specified in the project specification: Look at the `README` file in the `tools` directory. In particular, you can trigger the printing of the headers of the exchanged packets (printed in terminals). You can also start our client and server in a buggy mode to observe how errors are handled at the other side by your client or server.

4 Advices and Remarks

Messages and error messages
The client and server we provide print many messages for debug purpose, in particular detailed error message. This may be very useful to understand what happens in particular in case of error. You don't have to produce so many messages with so detailed information. Only the errors specified in the project specification's web page are required. However it is strongly recommended that you print other significant error messages when relevant as it may help you to debug your own code.

Blocking programs, and more...
One common reason for a program to block is to enter into an infinite loop. In the provided environment, the `read` syscall on sockets blocks when no data is available in this socket[1], thus waiting for data to arrive. So, if data never arrives, `read` blocks foerever ...For this reason you have to know which amount of data is to be read. The packet header is of fixed size, so no problem for the header. When the packet contains data of variable size, the total size if data is given in the header. So, after having read this header, you can deduce the amount of data to be read in the data field of packets. Last, if you have made an error when computing data size of previous packet, this error could impact next packets since what you read could be data of the previous packet.

Structuring the code
Since your code is likely to be much larger than the code you have produced during labs, we do suggest to plit your code in different C source files. For example, both sides (client and server) have to send and receive packets. Thus it could be a good idea to put the common communication code in a communication library which can be included by both sides. The provided example illustrates this practice. You can obvisouly develop any other library that could be necessary for your project. The document `doc/Cstructuring.pdf` should help you to better understand how to develop libraries. The provided project specifically contains directories for user libraries:

- `uinclude` (for "user include") to put your library headers (`.h` files).
- `usrc` (for "user source") to put your library source code (`.c` files).
- `ulib` (for "user libraries") to put your library objects files (`.o` files, generated by `make`).
- `bin` (for "binaries") to put binaries build by `make`).

[1] Other choices are possible but lead to more subtle and complex handling of communications.

`include` and `lib` are reserved for the libraries we provide. Last but not least, if you were to add libraries, you will of course have to modify the `Makefile`. For this you can rely on the provided example.

Syscall and LibC Functions that may be useful (this list is not exhaustive)

- `open` and `write` syscalls
- `lseek` (less probably `fseek`, `ftell`) to find file sizes.
- `perror` and `strerror` to print error messages associated to error codes `errno`. `errno` is set each time you call a function (syscall or function of the libC)
- `opendir`, `readdir` and `closedir` to handle directories and their content.
- `unlink` to remove a directory entry.
- `rename`...