FIP1A - RES110 Principes fondamentaux des réseaux
UE RES112 - Bases des réseaux, réseaux

TP Programmation Réseau

Printemps 2023

Enseignant et responsable du module: Jean-Marie BONNIN (jm.bonnin@imt-atlantique.fr)

Enseignante : Françoise SAILHAN (francoise.sailhan@imt-atlantique.fr)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

FIP1A - RES110 Principes fondamentaux des réseaux
UE RES112 - Bases des réseaux, réseaux
TP Programmation Réseau

1 Introduction

The goal of these exercises is to familiarize yourself with `Python`, with the `socket` interface and with the `fork` system call. The second exercise asks you to think about the constraints imposed by the socket interface and the multi-process server model. The third exercise asks you to overcome these constraints by using the `select` system call.

The most important elements of these exercises are summarized in the Conclusion section. Make sure you read that part and that you fully understand each point.

1.1 Using Python

First note that we are using Python3 (Python2 is deprecated since January 1, 2020).

Recall that there are two ways to run a python program (or *script*, the two terms are equivalent in the context of RES 112) :

1. You can use the command `python3 script.py` where `script.py` is the name of the script file.
2. You can put the following in the *first* line of your script : `#!/bin/env/python3` and then mark the file as executable by using the command `chmod u+x script.py`. Then you can launch the script by typing `./script.py` in a shell. Note that the working directory of the shell sometimes called the "current directory" has to be the one where the script is, hence the `./` "dot-slash" before the file name, indicating that the script is located in "this"directory (in Unix the dot represents the current directory).

1.2 Developing Networking Protocols

When developing communication protocols, a typical issue is to determine if bugs come from the sender or the receiver part. This is of course especially the case if you are the one who develop both the sender and the receiver software.

To cope with this problem during these exercises, you can use the `netcat` program to test your implementation. As mentioned in its documentation *"Netcat is a featured networking utility which reads and writes data across network connections, using the TCP/IP protocol"*.

Under Ubuntu/Debian Linux distributions, Netcat can be installed using the following command : `sudo apt install netcat`. You can then use Netcat through the `nc` command. Use `nc -h` or `man nc` for more information about how it works. For instance :

- `nc -l 1800` will start a TCP server listening on the port 1800 of the local machine
- `nc localhost 1800` will start a TCP connection to the port 1800 on the local machine (`localhost`)

Running these two commands into two different terminals, you should be able to send data back and forth between the client and the server. This way, you can test your server implementation using one or several netcat client(s). Conversely, you can use netcat in server mode to test your client implementation.

2 Exercise 1

As a first example of a simple program using the `socket` interface, you will implement a client-server pair, which we will call, for a lack of a better name, `echoserver` and `echoclient`. The server will simply print on the screen all the messages received from the client. Therefore, even though sockets can be used for bi-directional communication, in this first exercise, we will use them only in one direction. (As it will become clear later on, this greatly simplifies the programs.)

2.1 Part 1.A : Single Process Server

First, consider the simplest possible case : a server capable of handling only one client at a time. In this case, the server can use only one process.

As a starting point, you can use the skeleton files (in the `res-112-tp.tar.bz2` file). To extract the files you can use the command `tar xjf res-112-tp.tar.bz2`. The files `echo_client_socket_skeleton.py` and `echo_server_socket_skeleton.py` contain the code for the client and the server, respectively.

The server should wait for the client to open a connection (using the port number specified on the command line). Then it will simply display whatever it receives from the client.

The client should open a socket connecting it with the server, using the address and the port number given on the command line. Then it should wait for the user to input a string and send it to the server.

Hints/advice :

- The official Python socket documentation and its examples can be of a great help !
- It may be easier for you to develop the server first.
- Don't forget that you can use `netcat` to test your client and server separately (as discussed in the Introduction section). If needed, you can get the IP address of your machine with the following command : `ip address list`

Action Item 1 : First, make a copy of the `echo_client_socket_skeleton.py` file called `echo_client_socket.py` and a copy of the `echo_server_socket_skeleton.py` file called `echo_server_socket.py`. Then, modify the files in order to obtain the behavior described above.

When done, store the two files to prepare the final deliverable (1 per groupe) that will be associated with the report (1 per person). Do not forget to answer the following question.

1. Question 1A : What happens when a second client tries to connect while a first one is already connected ? Try to explain what you observe ?

The server refuses the connection because there is already a user connected to it.
The server socket can only handle 1 connection simultaneously.

2.2 Part 1.B : Multi-Process Server

Using the `fork` system call (from the `os` module), one can write a server capable of handling multiple clients. In this case, the server will start by listening for incoming connections on the port specified on the command line. Whenever a new connection request arrives, it will *spawn* a new *child* process, which will handle the connection (i.e., it will display what it receives from the client). Clearly you can use the same client as before.

Hints/advice :

- To do this part of the exercise, it is important to understand what a *process fork* is. Information about this can for instance be found on Wikipedia (in English or French)
- The `netstat` command enables to monitor opened connections on your system.
  - If not already done, `netstat` can be installed on Ubuntu/Debian with : `sudo apt install net-tools`
  - `netstat` can be combined with other GNU/Linux tools. For instance the command `watch -n1 "netstat -atn | grep 1800"` asks every second to netstat to list all TCP sockets and filter lines containing the value 1800. It may be useful to let this command run continuously in a separate terminal to monitor what happens.

Action Item 2 : First, make a copy of the `echo_server_socket.py` file called `echo_server_socket_multi_clients.py`. You can accomplish this by typing `cp echo_server_socket.py echo_server_socket_multi_clients.py` in the directory containing the `echo_server_socket.py` file. Then, modify the newly created file in order to obtain the behavior described above.

Make sure you test your server with multiple clients in parallel.

When done, store the two files to prepare the final deliverable (1 per groupe) that will be associated with the report (1 per person). Do not forget to answer the following question.

1. Question 1B-1 : What happens if you change the size of the receive buffer (that is the integer passed to the `recv` function) and you send fairly long messages from the client ?

The message will be splitted with the according size of the buffer, so that nothing is lost during the treatment of the server.

2. Question 1B-2 : Using the command `netstat -atn` you can list all the network connection on a machine. Find the ports associated to each connection between the server and the clients.

The server listens on port 1800 as expected, but the connection from the client uses port 52060

```
> tcp        0      0 127.0.0.1:52060         127.0.0.1:1800          ESTABLISHED
15,16d14
< tcp        0      0 10.129.160.42:41263     10.129.150.11:2049      TIME_WAIT
< tcp        0      0 10.129.160.42:59126     10.129.150.11:2049      TIME_WAIT
19c17
< tcp        0      0 10.129.160.42:59128     10.129.150.11:2049      TIME_WAIT
---
> tcp        0      0 127.0.0.1:1800          127.0.0.1:52060         ESTABLISHED
```

3. Question 1B-3 : As you might have noticed, in the sample code you used, the (server) socket was bound to an empty address (`host=''`). What would happen if we use `host='192.168.100.10'` instead ? Why ? (You can try to make this change and run the program to see what happens.)

The server's socket won't start because none of the network interfaces has this IP Address binded to it.

Nonetheless if we put one of those IP (in 10.0.0.0 subnet) and the same value on the client (instead of localhost), It works very well.

3 Exercise 2

In the previous exercise, we used all the connections in only one direction : the client(s) were writing and the server was reading. In this exercise we will use each connection in both directions. The goal is to make a first (baby) step toward a chat server. Now the second client to connect will display what is sent by the first client to connect. The server will simply forward the messages from the first client to the second.

The server starts by listening for the first connection on the port number given on the command line. After the first client makes a connection, the server sends a string with the number 1 to the client, so that it will know that it is the first client. Then the server waits for the second client to connect, when it does, the server will send a string with the number 2. At this point the server is ready to start forwarding messages between the two clients : it will wait for the incoming messages from the first client and forward them to the second one.

The client starts by connecting to the server, using the address and port number given on the command line. Once the connection is established, it receives (reads) one single character from the server. If the character is '1', the client asks the user to input the messages that will be sent to the server (and, therefore, to the other client). Basically, in this case, the client has the same behavior as in the previous exercise. If the character received from the server is '2', instead, the client will simply wait for messages from the server and it will display them.

Hints/advice :

- Keep Modest ! This exercise is a very first step toward a fully operational server. Fell free to "hard code" some parts of your program. For instance, the process of sequentially waiting for the first connection and then for the second one can be coded as such in your program.

Action item 3 : Do the following :

- Make copy of the `echo_server_socket.py` file called `forwarding_server_socket.py`, again, you can do this by typing `cp echo server socket.py forwarding server socket.py` in the directory containing `echo_server_socket.py`.
- Make a copy of the `echo_client_socket.py` file called `forwarding_client_socket.py`. As usual, you can do this by typing `cp echo_client_socket.py forwarding_client_socket.py`.
- Modify both newly created files in order to obtain the behavior described above.

When done, store the two files to prepare the final deliverable that will be associated with the report. Do not forget to answer the following question.

1. Question 2 : Do you think that it would be possible, using only the system calls you have utilized so far, to modify the server so that a third client can connect to the server and receive the messages from the first client as well ? (Hint : think about blocking calls and what they imply.)

4 Exercise 3

Using the `select` system call, which is available in Python, it is possible to overcome the limitations of the solution presented in the previous exercise (see the excerpt from the Python documentation below). `select()` allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., `read(2)`) without blocking. In our case, the server could handle multiple clients, forwarding the messages sent by the first client to all the others.

The following is an excerpt from the Python documentation :

`select` — **Waiting for I/O completion** This module provides access to the `select()` and `poll()` functions available in most operating systems, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets ; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following :

...

`select`(*rlist, wlist, xlist, [timeout]*)

This is a straightforward interface to the Unix `select()` system call. The first three arguments are sequences of 'waitable objects' : either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer :

- *rlist* : wait until ready for reading
- *wlist* : wait until ready for writing
- *xlist* : wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the timeout argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.
The return value is a triple of lists of objects that are ready : subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.
Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

4.1 The forwarding server

Action Item 4 : Make a copy of the `forwarding_server_socket.py` file called `forwarding_server_socket_select.py`. Modify the code of the server by using `select()`, so that it will handle any number of clients (more than two).

When done, store the file to prepare the final deliverable that will be associated with the report.

Hints/advice :

- In this part, you only have to work on the server code.
- But, for you tests, it would be convenient to have clients able to send and receive at the same time. Don't forget that you can use `netcat` for that purpose.

4.2 The full duplex client

In this part, you will implement a full duplex client that is able to wait at the same time for chat messages on the appropriate socket and for user input on `stdin` (just as `netcat` does).

Action Item 5 : Make a copy of the `echo_client_socket.py` file called `echo_client_full_duplex.py`. Modify the code of the client by using `select`, so that it will handle simultaneously incoming messages from the network and the command line.

When done, store the file to prepare the final deliverable that will be associated with the report.

Hints/advice :

- In this part, you only have to work on the client code. `select()` can be used to wait for any stream of the system (not only network sockets). For instance, `select()` can also wait for inputs on the standard input (`sys.stdin` in Python).

5 Exercise 4 (optional)

5.1 Make your server visible from the Internet (Bonus)

Up to now, you ran your client and server on the same machine. It is convenient for development and test purpose, but obviously not a valid solution for the real life. The goal of this exercise is to see how to run your server on a machine and try to connect to from a client running on another machine.

The issues you'll have to face clearly depend on your deployment environment. For instance, the situation may not be the same if

- your server run a virtual machine hosted on school's VDI infrastructure or
- your server runs directly on the school machine or
- if it runs on a computer at home (behind an Internet service Provider's "box")
- or even on a service provider like Amazon or OVH.

Different issues => Different solutions. Figure out what issues prevent your service from being accessible and try to solve them if possible.

Action Item 5 : Clearly, the second solution (ie. run the server directly on the school's computer, but not in the VM) should work. Implement it and test it with your teammates using two different computers.

1. Question 4A : Write a text to describe your configuration and what are the issues. If these issues can be overcame, tell how you would proceed. And if not, please explain why it not possible.

5.2 Use textual addresses (Bonus)

In the previous part, you used IP addresses to reach the server.

Action Item 7 : Try to deploy a configuration to let the client use a machine name or a FQDN to reach the server.

1. Question 4B : Write a text to describe your solution. would it be the same if your server was directly accessible from the Internet ?

6 To Go Further (optional)

6.1 Make your client and server compatible with IPv4 and IPv6

As you know the Internet is slowly moving toward IPv6. As of today most of the OS (Windows, Mac, Linux, Android, IOS...) integrate a dual IPv4/IPv6 communication stack.

Action Item 8 (Bonus) : Modify your client and server implementations to make them compatible with both versions of IP at the same time. Find out a way to test all combinations.

Hint/advice :

- This is not the only way, but you may use a network analyser such as Tcpdump or Wireshark to ensure that the traffic is conform to what you expect.

6.2 Multicast

One of the concern with our "chat" application is that each message has to be sent n times on the network (n is the number of client). It could become a concern when the number of clients increase. Using multicast communication at network layer enables to send a single message that will be delivered and processed by several machines. This could be a solution to implement a distributed live chat application.

Based on python sockets, try to implement such a chat system where one "sender" can send messages to several "receivers" at the same time using multicast. You can then run one sender and one receiver side by side on each machine. As you will have to use UDP for multicast communication, there is no acknowledgement, and you'll have implement it by yourself if you want to know who received your messages.

Action Item 9 (Bonus) : Modify your client and server implementations to make use of IP (v4 or v6) UDP multicast instead of unicast. Find out a way to test various combinations.

Hint/advice : You may need to use several (virtual) machines to test this set up. And these machines may have to be on the same network. To go step by step, you will use with a dissymmetric set up where you have two separate software : one for the sender part and another one for the receiver part. In a first step, do not consider the acknowledgment, and implement it in a second step.

6.3 Handle exceptions

Up to now you probably wrote your program as a sequence of instructions based on the assumption that all went as expected. But, in practice a lot of specific cases can happen. For example, a client can

disconnect from the server to which it was connected or a server can not be available when a client tries to connect to.

Action Item 10 (Bonus) : In this part, try to harden your code so that it is robust to the most probable use cases. Use function return codes and exceptions to gracefully handle theses cases.

7 Conclusion

At the end of these exercises you should fully understand (and remember) the following points :

- The `socket` interface allows processes to communicate using a TCP (or UDP) connection
- In the case of TCP connections :
  - The data sent with a single `write` (or `send`) function can be received with multiple `read` (or `recv`) calls (you have seen this in the first exercise (1.b) when you have changed the size of the receiver buffer).
  - A TCP connection offers a reliable "byte-stream" service : the bytes transmitted by the sender are reliably delivered in the correct order to the receiver.
- Some of the functions of the `socket` interface are *blocking* (e.g., `recv`) : when a program calls one of these functions, its execution is suspended until a certain event "unblocks" the call. For example, in the case of the `recv` function, the execution is blocked until it receives data on the connection.
- A process can be blocked only on one function on any given time. This implies that a single process can wait for only one event at any given time.
- The `fork` system call creates a clone of a process. Each clone (child process) can then block on a different event (e.g., receiving data from a specific client).
- Different processes do not share the same address space, therefore they cannot share data easily (This can be done, but it is outside the scope of RES 112).
- A process can use the `select` system call to wait for multiple events. This way a single server process can wait for data (and connection requests) from multiple clients.