UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

# Networking, Practical 1: Concurrent TCP Programming

---

The goal of this practical session is to familiarize yourself with the TCP API and concurrent, thread-based programming, in Java or Python. The different questions aim at showing what problems can occur when using concurrent program (several execution threads at the same time in a single program), and showing some possible solution to avoid these problems.

It is ok to design and test networking programs mostly on a single machine (the application layer connects two processes, they can indifferently run on the same or on different machine). Don't hesitate to try and explore things that are not in the subject (like mixing text and binary in a protocol).

Java Tips and Reminders:

- do not use `==` to compare Java Strings: use `s1.equals(s2)` instead of `s1 == s2`
- drawing a random number (between 0 and 1) can be done with `Math.random()`
- waiting for 10 milliseconds can be done with `Thread.sleep(10)`

Python Tips and Reminders:

- you need to run python as a program (no notebook etc), so you need to know how to start a kind of command prompt (terminal)

Windows Tips and Reminders:

- you can use "Putty" to make plain TCP connections by using the "Telnet" connection type and setting the "Passive" mode in the Connection>Telnet section
- in "Putty" you can save the connection settings to avoid retyping them every time

---

**Q1)** Download and unzip the `practical1.zip` archive from the course website. The folder contains a set of Python sources and a maven (Java) project, that you can use in eclipse with "import", "Import Existing Maven Projects". Alternatively, you should be able to easily import the project in any development environment, or even build it from the command line if you have maven installed.

---

What is called for this session a `Display` object holds a (circular) list of integers, and prints them when we call `render()`. We will first understand how it works.

**Q2)** Open the `A1` file, read it, run it, hack it, re-run it… Your goal is to understand what the `Display` and its methods do.

---

Now we will try to understand what are threads and how to use them.

**Q3)** Open the `A2` file. Read and understand the code and what it should do. When we create and *start* a thread, the program acts as if it had multiple execution points in parallel. Here, the *producers* are run in parallel and each is executing its own instance of the "Add Elements" function. At some point, run the program to validate your understanding and check that it actually runs in "parallel".

---

Now, you will make a multi-threaded application that access a `Display` object in response to network requests.

**Q4)** Open the `A3` and un-comment its body. Possibly taking inspiration from the code written during the courses (or the `Z*` files), implement the server code by following the comments and filling the blanks (`???`).

**Q5)** Start your server and test it manually using tools such `Telnet` or `Putty`. First use a single client, then multiple ones. NB: If you don't manage to install any of these tools, you can alternatively adapt the "TCP Client" written during the courses (also as a `Z` file). Iterate over implementing/(re)starting/testing until you obtain the proper behavior: a `1` is added to the `Display` at a random location every time you connect to the server and it is subtracted from the same location when you disconnect. The sum of the display thus represents the number of connected clients. Test what happens if a client does not disconnect, or if it disconnects before sending anything…

---

**Q6)** Design your (very simple) text-based protocol by choosing a *magic word*. When the server receives this word from the client, it will "move" the value associated to this client in the display.

The idea is that whenever a client sends this *magic word*, the server will move its `1` either to the left or to the right (chosen randomly once for each client) and re-render.

**Q7)** Modify your function that processes clients: every time the magic word is sent, as a line of text, from a client to the server, the server must:

- move the value in the chosen direction (`by`, chosen once for each client),
- increment `i`,
- call `display.render()` to update the window.

**Q8)** Test your functionality with `Telnet` or `Putty` to validate that it is actually working as expected. The sum of the display still represents the number of connected clients but now, in addition, we see when a client sends the *magic word*.

---

**Q9)** Open the `A4` file that will contain the client code. Eventually, the client must:

- connect to the server using TCP, then
- wait for a random amount of time between 0 and 100 milliseconds,
- send the magic word to the server,
- repeat this wait+send 50 times in total,
- and, finally, quit.

**Q10)** Implement your client and test it by checking that you get the expected behavior when the server is running.

**Q11)** While your server is running, start you client multiple times, rapidly (use keyboard shortcuts or shell loops if necessary). Try first with 2 clients and if everything seems normal, try to add more. What happens when there are multiple simultaneous clients? (understand the problem but *do not try* to solve the problem for now)

---

**Q12)** Open the `A5`. Read and understand the code and what it should do. Looking at the code, what should be the state and the sum of the `Display` at the end of the program?

**Q13)** Now, run it a few times and observe (hopefully) that things get awkward.

**Q14)** Now open the `A6` file. Read and understand the code and the differences with the previous one. Draw (using pen and paper) a diagram of the setup of this program. Your diagram should show the involved threads and the different objects/data structures with which they interact. You can also run it to ensure that everything is fine. (Java: also look at the "Better" example to have a cleaner version).

---

**Q15)** Copy your server program, naming it something like "A7 Robust Server".

**Q16)** Modify "A7 Robust Server", using the communication pattern you just learned from the example. The goal is to have a server that is robust to multithread aspects. Iterate over the implementation and testing of your new server until it works as well as the first version. Here are some helping guidelines:

- a queue will be used, as in the example,
- a single consumer will be responsible for accessing the `Display` (and calling `.render()`),
- the method that processes a client will be a kind of producer,
- the signature of this method might need to change.

**Q17)** Do you still observe the problems when you start multiple clients?

**Q18)** (Optionally) Run `wireshark` and record your local connection. Run your server and client and have a look at the TCP exchanges, their size, their sequence numbers, ack numbers, etc.

**Q19)** (Optionally) Make a multiplayer chat application.