Understanding TP11: A freshman's guide for other freshmen ...(or freshwomen)

Mehdi "Arcanite" OUESLATI
email: mehdi.oueslati@epita.fr

March 2019

Contents

1 Introduction

The purpose of this document is to help those in need of extra explanations for TP11. The explanations provided may sometimes be vulgarization, the goal being mainly that you understand the basics of networking, multi-threading and lambda-calculus (don't be afraid, we won't do maths here).

This document will be divided in two parts, those two being a theoretical part and the practical. The practical should be enough to do the job, but I strongly recommend you to read the theory because it will probably help you later on.

At the end of each of these two parts, you'll find additional links to look deeper into the materials.

Of course, I'll try to help whoever is in need, so feel free to send me an email. (Facebook and Discord is fine for those who know me in real life only)

2 Theoretical Background

Computer Science is a very peculiar domain of Science. It is one of the very few that you can apply and use without understanding anything about its theory. This being said, the understanding of theoretical Computer Science gives you the intellectual tools to tackle extremely complex problems with very elegant solutions. On top of that, it is really simple to understand, and its concepts are often general enough to allow the use of everyday life images to visualize their meaning.

This is why I am writing a theoretical section in this paper, even though I know that the majority of the readers will not care and skip to part 2. All I can say is: reading part 2 will help you with this TP; but reading part 1 will help you with most of your projects.

2.1 Networking

Computer Networks is the study of how to connect a machine with another, and that of the behaviour of a group of machines interacting with each others.

For this TP, you will need to use ”websockets”, and to use websockets, you will first need to know what the internet is.

2.1.1 The Internet Protocol

What is the internet? You may want to answer ”That thing that allows me to spam discord channels”, and that would be a correct from a consumer's perspective. However, you now have to write an IRC client, which means that that definition is no longer formal enough. The actual definition you need is the following:

The ”Internet” is a network of **Inter**connected **Net**works[1].

[1]If it may help, you can visualize that definition by looking at the structure of the Figure 1 graph.

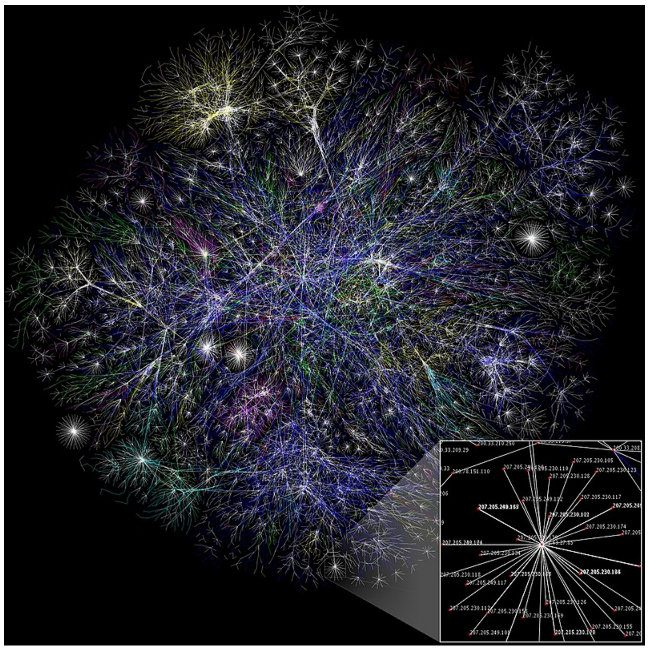

Figure 1: A partial map of the internet in 2015. Credits go to opte.org.

What this means to you is that you now have to imagine the internet as a sort of highway network. Different machines of a same network are in the same city, and they are linked to other cities through a route. More importantly, you can notice in fig1 that not all the nodes are connected to each other. This means that if Bhayem wants to send a message to Arcanite, Bhayem needs to find a way to tell A, B, C and D who Arcanite is. We hence need to have an unified naming system among all nodes (and obviously, each name needs to be unique, otherwise A wouldn't know to which Arcanite Bhayem wants to send his message).

The solution we decided to take is called **IP**, which stands for "**I**nternet **P**rotocol". Nowadays, two different kinds of IP adresses are in use: the IPv4 and the IPv6. Basically, IPv6 was invented when we ran out of unique names by the IPv4 format. There is a long story behind IP, but for now, just consider that IPv6 is better but IPv4 is more common, and thus most people use IPv4. This is already a problem, but will get worse if we don't start using IPv6.

2.1.2 The Transmission Control Protocol

Now that we solved the naming problem, we need to find a way for the machines to understand each others. Such a way to communicate is called a "protocol". The protocol we are going to use is TCP *for* ***T****ransmission* ***C****ontrol*

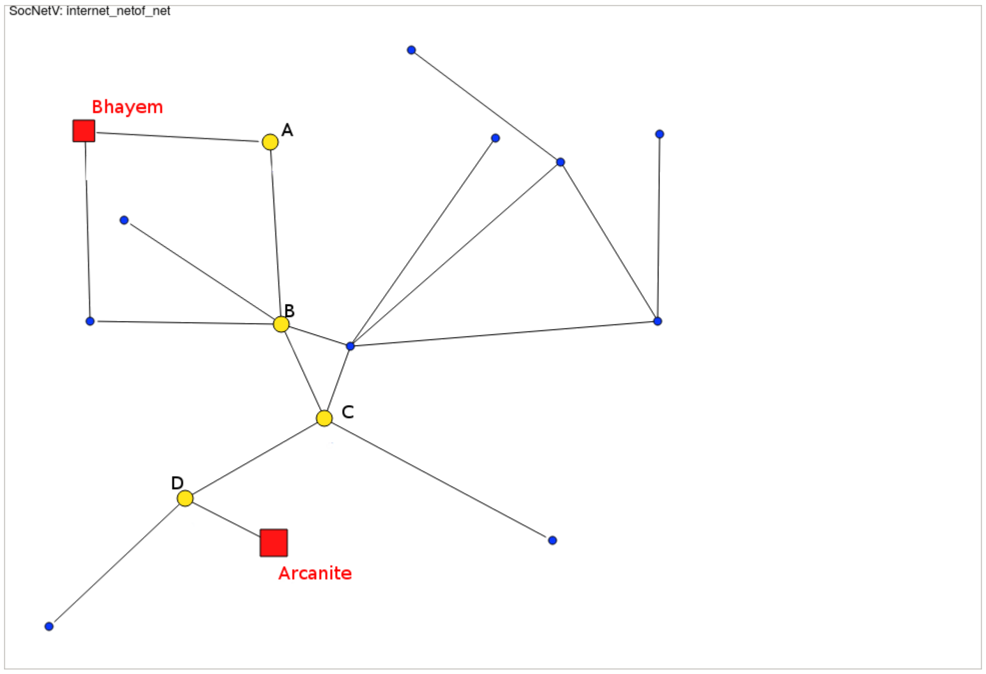

Figure 2: The simplified Internet model we are going to study

Protocol. For your knowledge, know that TCP is one of the most used communication protocols, and it guarantees that the data was sent and received properly.

2.1.3 Sockets

Now we are going to finally talk about **"sockets"**. Sockets were created to ignore all that extra complexity created by all the different networking details put together. We call the process of creating a new concept to ignore the mess created by the mashing of different others **"encapsulation"**. An easy example is the multiplication operation. In primary school, you learned that 2+2+2 can be written as 2*3, so instead of making 3 operations, you're doing just one. Therefore, the multiplication operation is a layer of abstraction on top of the addition operation. Another way to put it is that the additions are encapsulated by the multiplication.

Sockets can be seen as just a bridge between two machines. Instead of just sending one message at the time, the socket allows the two machines to engage in a conversation and to keep that conversation alive.[2]

[2]source of figures 4 and 5: https://www.ably.io/concepts/websockets

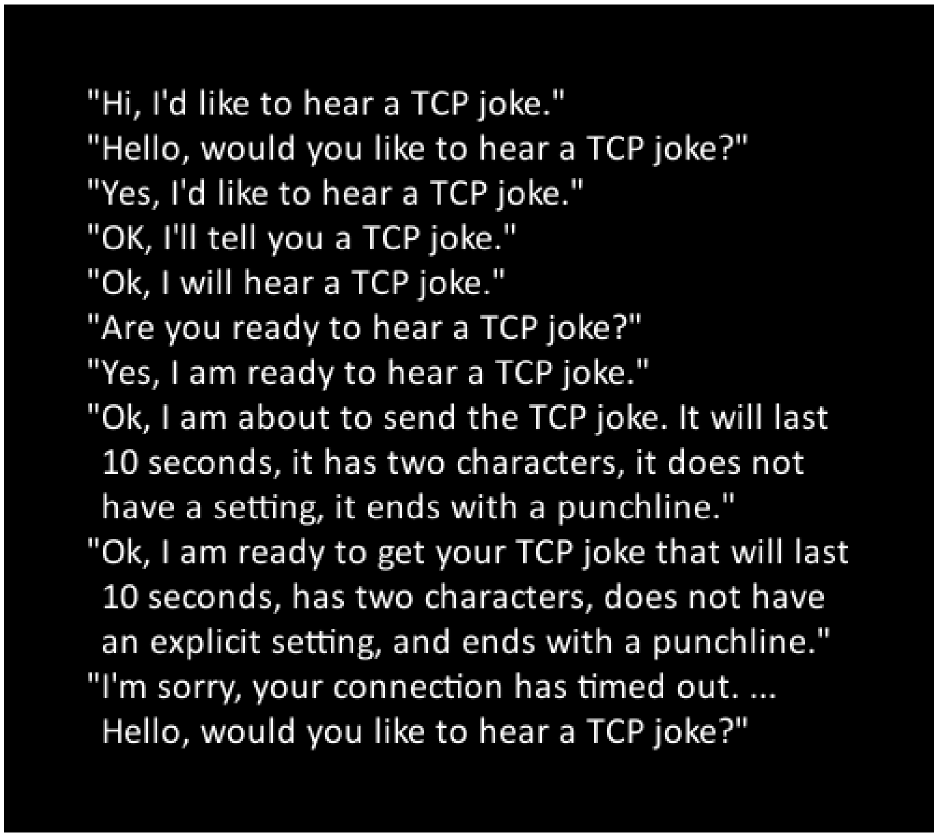
"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I'll tell you a TCP joke."
"Ok, I will hear a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."
"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."
"I'm sorry, your connection has timed out. ... Hello, would you like to hear a TCP joke?"

Figure 3: should give you a good idea of how TCP works without getting into too many details

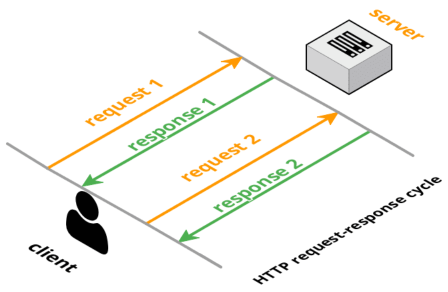

Figure 4: Without sockets

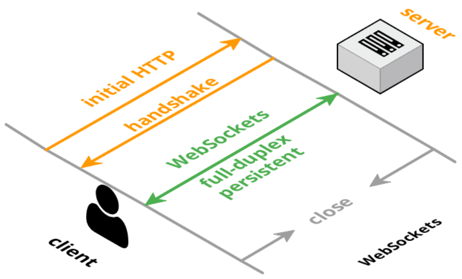

Figure 5: With sockets

2.1.4 packets

One last thing: You already know that computers deal with binary strings. You will thus not be surprised to know that computers communicate by sending **packets** of data. A packet of data is basically a binary word of fixed size.

2.2 Multi-threading

Understanding the theory behind multi-threading is extremely simple. All the issues are in putting it into practice.

Let's use an image: Bhayem owns a restaurant. Bhayem, as usual, is very dumb. He thus tries to do all the job as a single man. He welcomes the money, takes the orders, cooks, cashes the money, and waves the customers goodbye.

Obviously, this method is highly inefficient. Bhayem is too busy by one task at a time, and thus it takes him too much time to bring a meal, the customers are annoyed by the waiting time, and they just leave to other restaurants, like Arcanite's. Arcanite's restaurant is a normal restaurant. A specific person is in the doorman, another one is the waiter, then there's a cook, and the cashier. Each one of them is taking care of one specific process regardless of how fast the others act; we say that they take care of their task **asynchronously**.

In the computer science world, each person of the example is a thread. Threads work asynchronously on task to make sure the whole application works fast enough.

2.3 λ-calculus

A bit of nerdiness: λ-calculus is a very powerful theory that was created by Alanzo CHURCH. Alonzo CHURCH is a great mathematician and was the master of thesis of Turing, whom I talked about in the previous programming-sheet paper I wrote for Int-1. If you're nerdy enough you may even get excited to know that λ-calculus allows you to do all the cool stuff that you can do with Turing machines.

It really starts here: Now let's move on to what interests us for this TP. λ-calculus considers that everything is a function. And by everything I really do mean everything. For instance, the number "5" is not just a number but the constant function of value 5. This means that the following code is true:

$$\textbf{(A:)}\ \text{var a} = 5 \equiv \ \text{var a} = f(x) \rightarrow 5$$

Now here's the powerful part. If we can do that for the constant function, we can do that for pretty much any function. We call that a λ (lambda). For instance, the following is a valid lambda:

$$\textbf{(B:)}\ \text{var b} = f(x) \to \exp(-5 \cdot x)$$

Now, let's not get into the maths, because we could write books on the subject and this is not the main topic. So let's just conclude by saying one last detail: the "$f(x) \to \exp(-5 \cdot x)$" part of **(B)** is, from a programming perspective, an **expression**.

3 Practical Perspective

**"Give a man a fish and you'll feed him for one day;
teach a man how to fish and he'll feed himself forever."**

So... feedback told me that the part that interested the most people in last paper was the practical case. This is why I cut this paper in two sections. Again, I still emphasize on the importance of theory.

Good practice when working on a project is clearly defining our goal. In the case of this TP, what we want to implement is an IRC client. In other words, we want to send data to a server and display the server's response to the screen. We are first going to look at the architecture of the project before looking at actual programming techniques.

3.1 Details of the project

Command Line Interface is basically a fancy Console object you're going to use to display stuff and take input. Use it instead of Console.

NumericReplies a certain number of codes have specific meaning. The standards are in the RFCs.Read the RFCs. No, really. **Read the RFCs**.

StringTools is a class that you're going to use to parse your data. I know you love parsers, don't you?

Connection will handle the networking part of the project. Look at section 3.2.

Client is just a fancy Main(). You will launch the program through this.

Channels and Commands Channels are discussion groups. Commands are orders given to the program. Again, **Read the RFCs**. It's a good habit to **read the RFCs**.

3.2 Networking

Now that you know the theory, it should be extremely easy for you. We are using one single web socket. You don't have to implement it, you just use the Socket class from System.Net.Sockets. What you send and receive is Commands that need to be parsed.

Client in handles the input interactions between the CLI and the Connection.

Client out handles the output interactions between the CLI and the Connection.

3.3 Multi-threading

Everything has to work in parallel. For that, you're going to use **async** and **await**. Basically, to make a thread independant, you have to tell the computer that your method is asynchronous (definition of asynchronous was given in part1). This is why you must must use the "async" keyword. The "wait" keyboard just means that you're waiting for something to give you an output before continuing your work.

Let's take a look at how the waiter at Arcanite's restaurant does his job:

```
//this is in the Waiter class
Async Task<Plate> doJob(Client client){
    Plate plate = await Cook.FinishCooking(client.meal);
    this.give(client, plate);
}
```

Notice that the type of the method's output is Task¡Plate¿. This type is a **promise**. As you may have guessed, the "doJob" method **promises that it will output a Plate when it finishes its task**. It has to **await for the Cook to finish his own task** before continuing its work. While it is waiting, the rest of the restaurant's staff can carry on and do other things. Even the Waiter can continue his business and take the orders of other clients.

Task.Run I talk about it in the following subsection

3.4 λ-expression

3.4.1 Actual lambdas

I really don't understand why they introduced λs in this TP. You can do it without them, and we are using the Object Oriented Paradigm, so why mix it with Functional one?

In any case, I'm not going to complain, especially that I prefer FP over OOP. *(It makes things more pure and helps keeping everything in its place, without creating a mess. You only care about one input output thing at a task, and can thus focus more on the detail and reuse more code when moving on to another project. This is a personal opinion though, and I understand that OOP helps you deal more easily with the broader picture. Each paradigm has its uses, I guess.)*

To write a λ-expression in C#, you will use the following syntax:

```
(input) => expression
```

For instance, the $\lambda 5$ is:

```
() => 5
```

Since it is an expression, you can store it in a variable.

```
Func<int> constFive = () => 5;
```

If you really want to use λs in this TP, know that you can use it to create multithreading. Some methods can't be awaited because they won't promise anything. For instance, the task of "waving customers goodbye" doesn't return anything. In that case, you can't use the await keyword, and you'd rather use the following syntax:

```
Task.Run(Func<Task>);
\\or, if you actually have a return type:
Task<TReturn>.Run(Func<TReturn>);
```

3.4.2 Delegates

Delegates actually come from programming paradigm that is different from OOP or Functional: the **Generic** paradigm.

A little digression If that interests you, I will talk about next time I write a paper *(as usual, send me feedback at mehdi.oueslati@epita.fr if you don't know me in person, or use discord or facebook if you do. By the way, this paper is in Beerware so you could offer me a beer to get to know me.)*

Delegates are not lambdas. They are actual functions that will do a different task depending on their parameters. However, they do specify what input they want and what output they will give. Since a lambda is technically a kind of function, you can put a lambda in a delegate.

One last thing: Open C# programming question: give a clear explanation of the warning given at page 7 of the TP spec sheet.

4 Conclusion

Too lazy to write one. I still did not start working on the TP myself. GL HF.

5 Beyond New Worlds:

Here is a list of resources to learn new exciting things and look further in the material brought up in this paper:

λ-calculus	https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf*(from the Free University of Berlin)*
Networking	https://intronetworks.cs.luc.edu/current/ComputerNetworks.pdf *(from Loyola University of Chicago)*
Mutli-threading	https://www.cs.cmu.edu/ 15210/concurrency.html*(from Carnegie Mellon University)*

You could also want to **read the RFCs** and look for stuff on the MSDN.