

# CES-25 Computer Networks and Internet

## 2th Activity

Prof Lourenço

Carlos Matheus Barros da Silva

September 2019

## 1 Implementation

All the code was implemented using Python 3. The complete code can be seen on the Repository<sup>1</sup>.

The code was divided into five files, being four of them classes. The classes are *Commander*, *Client*, *Server*, *FileManager* and there is a utility file called *utils*.

### 1.1 Commander

It is an abstract class that will define methods related to command line. The classes *Server* and *Client* will inherit from *Commander*.

The commands methods parsing and routing are provided by the *Commander* abstract class they will be routed to the corresponding abstract method that are implemented by the *Server* and the *Client* classes.

Therefore if *Commander* receive “cd ../” it will first split the message by spaces and then pass the first element to a command hash that will lead to the specific method or to the *unknown* method in case no method related to that name exists, the others elements of the splitted string are passed as arguments for the method and each function will deal independently with it.

### 1.2 FileManager

The *FileManager* class is a class that deals with the computer files. Every thing related to read or write files or directories will be done through the intermediate *FileManager* class.

Therefore the code will be more specialized and will be better to organize and prevent race conditions on the server when more than one thread are trying to access the shared resources. On that way locks are implemented so that it is guaranteed that at most only one thread area accessing the *FileManager* methods at the same time.

*FileManager* also include path validation and simplification methods, wherefore it can verify that “a/../a/../a” is equals to “a/”, it also verify the validity of a given directory or file. It furthermore checks if the directory that is been accessed is inside the root folder defined by the server, that way it prevents the user to navigate and access every file on the server machine, it just allow the navigation through the files inside root folder defined by server.

### 1.3 Server

It is the class that controls the server, and in order to initiate a server it is just necessary to create a instance of *Server*. The server will initially define the port it will initiate the server and then define a password for the server. The password will be stored as an *Sha256* hash of the password given.

When the server is initialized it will be listening and waiting for new connections, when it receive a new connection request from a client it will initiate a new server object on a new thread specially for that connection so the main thread will continue waiting new connections, and every connection already established will have its own thread.

---

<sup>1</sup><https://github.com/CarlosMatheus/ftp-server>

On the specific thread connection, the server will be first waiting for the user provide a message with the correct password hash. Once the user is authenticated the server will be waiting for commands.

The commands methods parsing and routing are provided by the *Commander* abstract class they will be routed to the corresponding abstract method that are implemented by the *Server* and the *Client* classes.

Therefore if *Commander* receive “cd ../” it will first split the message by spaces and then pass the first element to a command hash that will lead to the specific method or to the *unknown* method in case no method related to that name exists, the others elements of the splitted string are passed as arguments for the method and each function will deal independently with it.

So for the commands that the server will be able to receive, all the commands requested were implemented following the arguments utilization. Thus the *cd*, *ls*, *pwd*, *mkdir*, *rmdir*, *get*, *put*, *delete*, *close*, *open* and *quit* were implemented.

Their complete utilization follows strictly the suggestion on the Activity proposal. Besides for the *get* and *put* it can be used with one more optional argument defining the folder to interact.

## 1.4 Client

This class implements the client side application. In order to run the client side application just need to create an instance of this class.

When initiate the client application it will be prompt a command line. At first until the user are not connected and authenticated to a server all commands are blocked except *open* and *quit*. When the client calls open passing the arguments to the server address and port it will be asked the server password. So for example “open 0.0.0.0:8080” will try connect to the server on the 0.0.0.0 address at 8080 port.

After connect the server will ask the password, once the user prompts the password the value will be hashed using *Sha256* and sent to the server and the server will answer telling if it is authenticated.

Once authenticated, the client will be waiting for the user input the command to send it to the server. Every command the user type will be parsed through the *Commander* methods and routed to the respective method, on it will follow the respective logic locally before send the message to the server with the command and arguments. If there are any error detectable from the client side it will be shown the errors message on the client side before send to the server.

## Implementation

### Suggested Test Case

As It was suggested, it was conducted a test with 3 terminal windows, on each one was opened one task of the program as shown in the Code ??.

The test was made according to the model represented on Figure ??. The results can be seen on the terminal windows shown from Figure ?? to Figure ??.

As expected, the logical clock on each process was updated to the bigger one, if the incoming message logical clock time was greater than the actual time on that process, and then it was also increased by one. This logic was applied and because of it the simulation on the terminals matched the model represented on Figure ??.

## Built Test Case

It was built a test case with 4 terminal windows, on each one was opened one task of the program as shown in the Code 1.

The test was made according to the model represented on Figure ???. The results can be seen on the terminal windows shown from Figure 2 to Figure 5.

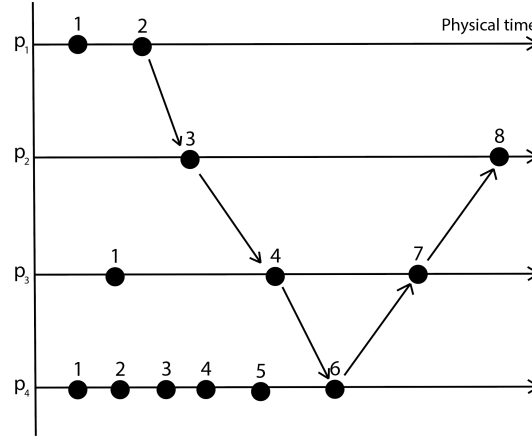


Figure 1: Model representing the execution of Task 1 built test case.

```
Terminal 1: go run Process.go 1 :10004 :10003 :10002 :10001
Terminal 2: go run Process.go 2 :10004 :10003 :10002 :10001
Terminal 3: go run Process.go 3 :10004 :10003 :10002 :10001
Terminal 4: go run Process.go 4 :10004 :10003 :10002 :10001
```

Code 1: Code that was run on each of the 4 terminal window on the execution of Task 1 built test case.



Figure 2: Window 1 after execution of Task 1 built test case.

As expected, the logical clock on each process was updated to the bigger one, if the incoming message logical clock time was greater than the actual time on that process, and then

```
carlosmatheus@carlosmatheus-VirtualBox: ~/Documents/CES-27-Labs/Lab01/task01$ go run Process.go 2 :10004 :10003 :10002 :10001
3
Current logical clock: 3
3
Destiny Id: 3
Current logical clock: 8
[]
```

Figure 3: Window 2 after execution of Task 1 built test case.

```
carlosmatheus@carlosmatheus-VirtualBox: ~/Documents/CES-27-Labs/Lab01/task01$ go run Process.go 3 :10004 :10003 :10002 :10001
3
Destiny Id: 3
Current logical clock: 1
Current logical clock: 4
4
Destiny Id: 4
Current logical clock: 7
2
Destiny Id: 2
[]
```

Figure 4: Window 3 after execution of Task 1 built test case.

it was also increased by one. This logic was applied and because of it the simulation on the terminals matched the model represented on Figure ??.

## Task 2

### Suggested Test Case

As It was suggested, it was conducted a test with 3 terminal windows, on each one was opened one task of the program as shown in the Code 2.

The test was made according to the model represented on Figure 6. The results can be seen on the terminal windows shown from Figure 7 to Figure 9.

```
Terminal 1: go run Process.go 1 :10004 :10003 :10002
Terminal 2: go run Process.go 2 :10004 :10003 :10002
Terminal 3: go run Process.go 3 :10004 :10003 :10002
```

Code 2: Code that was run on each of the 3 terminal window on the execution of Task 2 example test case.

```

carlosmatheus@carlos-matheus-VirtualBox: ~/Documents/CES-27-Labs/Lab01/task01$ go run Process.go 4 :10004 :10003 :10002 :10001
4
Destiny Id: 4
Current logical clock: 1
4
Destiny Id: 4
Current logical clock: 2
4
Destiny Id: 4
Current logical clock: 3
4
Destiny Id: 4
Current logical clock: 4
4
Destiny Id: 4
Current logical clock: 5
Current logical clock: 6
3
Destiny Id: 3

```

Figure 5: Window 4 after execution of Task 1 built test case.

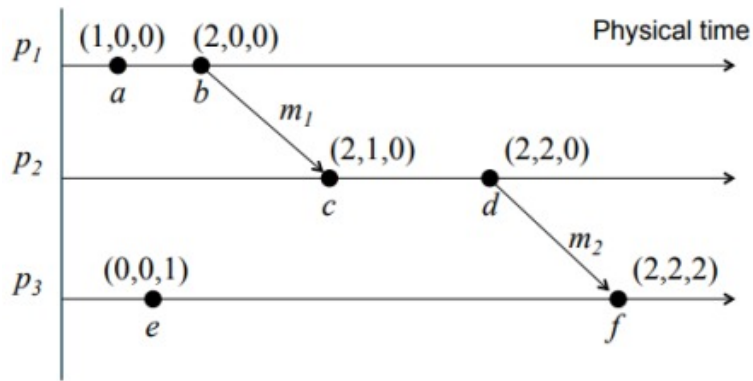


Figure 6: Model representing the execution of Task 1 example test case.

As expected, the logical clock on each process was updated to the bigger one, if the incoming message logical clock time was greater than the actual time on that process, and then it was also increased by one. This logic was applied and because of it the simulation on the terminals matched the model represented on Figure 6.

## Built Test Case

It was built a test case with 3 terminal windows, on each one was opened one task of the program as shown in the Code 3.

The test was made according to the model represented on Figure 10. The results can be seen on the terminal windows shown from Figure 11 to Figure 13.

```

Terminal 1: go run Process.go 1 :10004 :10003 :10002 :10001
Terminal 2: go run Process.go 2 :10004 :10003 :10002 :10001
Terminal 3: go run Process.go 3 :10004 :10003 :10002 :10001
Terminal 4: go run Process.go 4 :10004 :10003 :10002 :10001

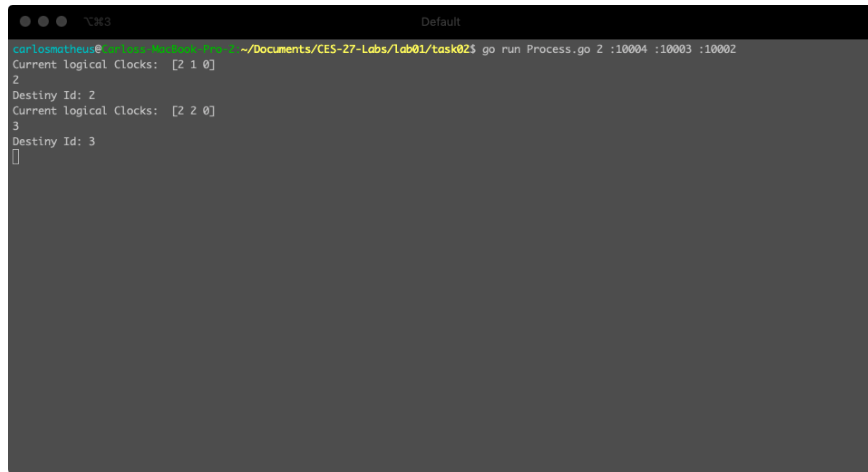
```

Code 3: Code that was run on each of the 3 terminal window on the execution of Task 2 built test case.



```
carlosmatheus@carlos-matheus-vm2: ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 1 :10004 :10003 :10002
1
Destiny Id: 1
Current logical Clocks: [1 0 0]
1
Destiny Id: 1
Current logical Clocks: [2 0 0]
2
Destiny Id: 2
```

Figure 7: Window 1 after execution of Task 2 example test case.



```
carlosmatheus@carlos-matheus-vm3: ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 2 :10004 :10003 :10002
2
Current logical Clocks: [2 1 0]
2
Destiny Id: 2
Current logical Clocks: [2 2 0]
3
Destiny Id: 3
```

Figure 8: Window 2 after execution of Task 2 example test case.

As expected, the logical clock on each process was updated to the bigger one, if the incoming message logical clock time was greater than the actual time on that process, and then it was also increased by one. This logic was applied and because of it the simulation on the terminals matched the model represented on Figure 10.

```

carlosmatheus@carlos-matheus-Virtual-Box: ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 3 :10004 :10003 :10002
3
Destiny Id: 3
Current logical Clocks: [0 0 1]
Current logical Clocks: [2 2 2]

```

Figure 9: Window 3 after execution of Task 2 example test case.

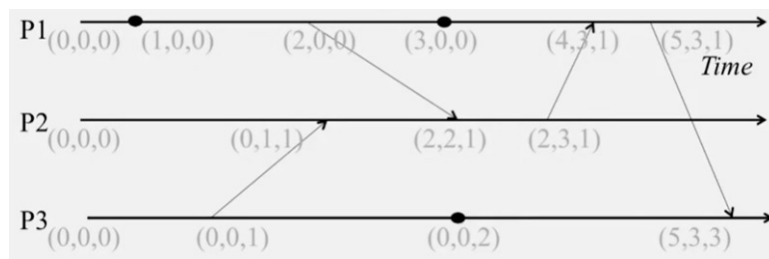


Figure 10: Model representing the execution of Task 2 built test case.

```

carlosmatheus@carlos-matheus-Virtual-Box: ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 1 :10004 :10003 :10002
1
Destiny Id: 1
Current logical Clocks: [1 0 0]
1
Destiny Id: 1
Current logical Clocks: [2 0 0]
2
Destiny Id: 2
1
Destiny Id: 1
Current logical Clocks: [3 0 0]
Current logical Clocks: [4 3 1]
1
Destiny Id: 1
Current logical Clocks: [5 3 1]
3
Destiny Id: 3

```

Figure 11: Window 1 after execution of Task 2 built test case.

```
carlosmatheus@carlos-matheus-vm: /~$ cd ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 2 :10004 :10003 :10002
Current logical Clocks: [0 1 1]
Current logical Clocks: [2 2 1]
2
Destiny Id: 2
Current logical Clocks: [2 3 1]
1
Destiny Id: 1
[]
```

Figure 12: Window 2 after execution of Task 2 built test case.

```
carlosmatheus@carlos-matheus-vm: /~$ cd ~/Documents/CES-27-Labs/Lab01/task02$ go run Process.go 3 :10004 :10003 :10002
3
Destiny Id: 3
Current logical Clocks: [0 0 1]
2
Destiny Id: 2
3
Destiny Id: 3
Current logical Clocks: [0 0 2]
Current logical Clocks: [5 3 3]
[]
```

Figure 13: Window 3 after execution of Task 2 built test case.