uc3m | Universidad Carlos III de Madrid

BACHELOR’S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

# FINAL PRACTICE
# -
# Design and implementation of a communication system

Students:
Moisés Hidalgo Gonzalez - 100405918
Álvaro Morata Hontanaya - 100405846

# SYLLABUS

**SYLLABUS** **2**

**CLIENT IMPLEMENTATION** **2**

DESIGN 3

**SERVER IMPLEMENTATION** **3**

DATA STRUCTURE 3

DESIGN 4

**COMPILATION** **4**

COMPILATION USING CMAKE 4

**COMMUNICATION PROTOCOL** **5**

**TEST PLAN** **5**

**PROBLEMS FOUND & CONCLUSIONS** **6**

## CLIENT IMPLEMENTATION

In this part of the document we will give a brief description of the different design choices we have made to develop this exercise.

### DESIGN

First, we have defined a method called threaded_listener(), which will be in charge of listening to incoming messages.

We have added the necessary code to the defined methods (such as `register()`) so that they send through a socket the operation name and the necessary information that the server needs in order to execute the operation.
We have then managed the code returned by the server and managed its decodification, to match the error handling in the client side.

## SERVER IMPLEMENTATION

### DATA STRUCTURE

Like in previous practices, we have followed a file structure as a database. We think that it allows for permanent storage and it's easier to implement than other solutions but also effective. However, as suggested by the profesor, we have implemented this time binary files instead of .txt files. We designed it like the following:

```
./src
    |- /messages
           |- 0
           |- 1
           |- 2
           |- ...
           |- N
    |- /users
           |- user1
           |- user2
           |- user3
           |- userN
```

Our source code is inside the ./src folder. In it, we can also find two main directories:

- **messages**: it stores the different pending messages that have not been able to be delivered since the recipient user was not logged in the system. Each message has its own identifier and we keep a list of the messages undelivered on each user, so whenever they log in they retrieve the message from this folder and delete it. The structure of the messages is as follows:
  `id|message contents|sender`

- **users**: It stores the information of our different users. Each file will be a different user with the username as the search key in the directory. For each user, we will store the following fields:

- Username
- Status
- IP
- Port
- Pending messages (to receive)
- Identifier of the last message received

They will be stored in a format like the following:

- `user1|Off|10.0.127.6|45677|5,67,78|4`

The “`|`” will be our token to separate the different fields, retrieve them, modify them, and so on.

## DESIGN

We have developed the code for the data structure as well as the methods for registration, unregistration, connection and disconnection of a user to the system.

For the data structure, we have used various methods to manage different operations, like encoding the contents of the files for both messages and users, or parsing those contents to work with them. There’s also a method for removing a directory which we used in the previous exercises. We have been able to reuse part of these methods from different practices, or by combining them. We have created an `init()` function to initialize the database.

We have created the functions `register_user()`, `unregister_user()`, `connect_user()` and `disconnect_user()`, which implement the functionalities as requested in the statement of the final assignment. We have also included a function `send_u2u()` which would perform the job of passing the messages from user to user, but we have not got the time to implement it.

To deal with the requests the client makes, we have made the function `deal_request()`, which is in charge of receiving the contents passed through the sockets and calling to the different functions of the system, later returning a value for confirmation of the execution of the method and the result. This function has the help of the method `readLine()` which we took from the file `lines.c` of the sockets lab. This function extracts line by line the data received from a socket file descriptor.

The server also has the `main()` method which is in charge of the loop for the continuous execution of the server and listening for new connections via sockets.

## COMPILATION

The compilation of the project has been realized using CMake. In this section we will explain how to compile the project using this tool.

### COMPILATION USING CMAKE

The file “`CMakeLists.txt`” has been written using the different instructions that allowed us to compile the source code and the shared library in the same way we were able to do with GCC.
To compile the different files we need to enter the following commands into a linux terminal being in the same folder as the file “`CMakeLists.txt`” is and the C files:

```
> cmake .  #It will create a makefile to compile the project
> make     #Will compile the files using gcc
```

With these two commands the project will be compiled, delivering us the executables “`server`”, “`client.py`”
To run the programs, we need to run the following commands:

```
#Server terminal:
> ./server -p <port number>
#Client terminal:
> ./python3 client.py -s <ip server> -p <port number>
```

## COMMUNICATION PROTOCOL

- **Server to Client**:

Data will be sent with a buffer. In most methods, this will just be a return statement, indicating whether the operation went OK or there were any errors. For that, there will be a *socket.recv(nbytes).decode()* at the end of each method, waiting for the response of the operation on the server side after having sent all different parameters. The *.decode()* will be used to decode the bytes received as a response into strings. This method will make sure to block the program flow until it reaches a server response.

For the message receiving part of the different clients, we have implemented the *threaded_listener(socket)* method which will receive a socket (client.server_socket) created for each client (outside of methods but inside the class client) that will be called inside a thread within the whole scope of the Client class (client_listening) in the connect(user) method. This will make sure that when the connection is done and the response from the server is received, this thread will execute for each client until disconnected, receiving data and decoding it. Then, on the disconnect(user) method, we will join the thread running this function and close the socket opened for listening to messages on the client side.

- **Client to Server**:

Data will be sent field by field as an independent string (except some integers), which will be sent with a .encode() to encode it to bytes and be able to send the strings through the sockets with the sockets.send() method.

On the receiving end, the server will read line by line with the readLine() method learned in the labs. It will interpret it sequentially with the same order we send it on the client side, with different if statements that will follow the workflow for the development of our program. First, the operation code will be read with its own error handling, then, that will go to a switch case that will execute different methods with their own error handling and so on.

## TEST PLAN

We have test methods that do the same functionality for our example thread 1 and thread 2. The only difference is the data they handle (different keys, values, etc).

The test plan aims to test in different scenarios the functions required for this exercise:

- `def register(user)`
- `def unregister(user)`
- `def connect(user)`
- `def disconnect(user)`
- `def send(user, message)`

Following the aforementioned objective, since now we don't have a test function in the client but the main itself, we tested manually our whole set of functions, both in client and server side. For each method, we checked the different error messages and situations, and did limit testing:

- `register(user)`: We checked
- `unregister(user)`: We checked
- `connect(user)`: We checked
- `disconnect(user)`: We checked

## PROBLEMS FOUND & CONCLUSIONS

Mainly, there has been a lack of time to dedicate to the practice on our side. Even though it has been uploaded for a month and a half, both of us have 6 subjects and we work part time, so it has been really difficult to balance everything. Also, we wanted to dedicate a lot of time in understanding and solving the previous exercises in their totality with a good quality.

Trying to balance everything, we haven't disposed of a lot of time to put into the final practice. Had we had more time, we would have improved:
- Finished the server-side implementation, completing the message passing between client → server → client
- Make a more comprehensive test plan
- Use fine-grain locking for the critical sections (both directories and files)
- Implemented WebServices
- Curated more the code, since we might have missed some situations

Regarding the difficulty of the practice and the duration of the course, we must say that it's been hard to develop but with the knowledge acquired throughout the semester we already knew how a lot of things had to be done and knew where and when to reuse our previous code as well. It has been a good challenge and a mind stretch to learn more and consolidate the different aspects of the practice altogether.

We have also been able to apply different solutions to the suggestions by the professor throughout the semester. We started applying fine grain locking, using binary files instead of txt, fixed a lot of segmentation faults, started sending separate fields over sockets instead of structs, and fixed a lot of segmentation fault issues we had.

Overall, we feel an improvement and that we have learned a lot in the time we applied to this final practice.