Assignment 1

Design and implementation of a communication system with POSIX sockets

9th of April, 2017

Carlos Gutiérrez Paradela – 100291121
100291121@alumnos.uc3m.es

Rubén López Lozoya – 100292107
100292107@alumnos.uc3m.es

Table of Contents

1. Introduction

The aim of this practical assignment is to understand the way programs communicate with each other by means of sockets. This kind of implementation allows different programs running on different machines to communicate with each other regardless of their implementation. This is possible due to sockets acting as an intermediate software layer between to programs. They run on top of the actual implementation of the programs and they provide a channel of communication between two ends in a bidirectional way. For this reason, different machines are able to talk to each other without having to know what each programs does indoors. The only necessary thing to do is to follow a common protocol in both sides so that at every moment both machines receive what they expect.

To prove the theoretical background underlying the concept exposed, we will implement a messaging system where the server handling message disposal and distribution is developed in C code and communicates with clients implemented in Java which are responsible for sending messages to the server in order to communicate with other clients. The running environment will be a Linux system, even though the system is intended to work in other operating systems that support the POSIX standards.

2. Code Description

This section will contain a description of the code structure and design decisions that were made during the implementation process. Every design decision will be presented in bullet points later on the section and properly reasoned based on the criteria that was followed.

2.1. Server Side

Before getting into any explanation, it is important to point out that there are different code modules provided in the submission: *server.c, user_list.c* and *msg_list.c*; with their corresponding header files. The reason for having three separated code files is merely for simplicity purposes. It was decided to split the code and separate the functionalities: one for server functionalities (thread creation, connection handling), another for dealing with the list of users and all the operations supported; and one for handling the list of pending messages. This, together with the header files help to the readability and logical separation of the code.

msg_list.c

- FIFO behavior of the message queue. This way the order in which the messages are sent can always be kept.
- No maximum number of pending messages is delimited for the problem. When the memory of the server is full, it will retrieve an error when trying to *'malloc'* a new message, and this exception will be handled returning the code 2 to the client.

user_list.c

- As it is not specifically stated in the guidelines of the practice, the maximum length of a username will be of 256 characters in order to not interfere with the protocol.
- All username are capitalized and converted to upper case letters before storage. This is a way to simplify the functionality and avoid storing two different users for the same username but with different capitalization, e.g.: silvina, SiLvInA, SILvina, silVINA, etc. This capitalization will be performed before calling any operation of the list of users (to avoid replicating this invocation inside every operation listed above).
- The *last_id* parameter of a user will be set to 0 every time a user is registered. When the user unregisters and registers back, this ID will be reset to 0. Otherwise, it is needed to keep a log of every registered user and its last ID since the start of the server (maybe using a file), and every time a user is to be registered, this log has to be accessed to check if the name is in the list, and get the last ID. This way we keep things a little easier and focus on other functionalities.
- No maximum number of users is delimited for the problem. Same fashion reasoned for the messages.

server.c

- Once the process is started, the main thread that we will refer as 'father thread', is looping listening for incoming requests in the father thread's socket. With each request, it will create a thread on demand that will be in charge of dealing with the command received from the client. Right after the thread creation, the father thread is stopped using a conditional variable until the 'child thread' locally copies the socket descriptor, resulting from the *accept,* so that is not overwritten by the *accept* of the next client request.
- The buffer to hold the type of command (CONNECT, REGISTER…) will have a maximum size of 11 bytes, 10 for the maximum possible string length ('DISCONNECTED') plus 1 for the termination 0.
- If any error is detected that prevents the server from communicating with the client (operations with sockets, error when trying to get the address of the client…) then the child thread will try to close the socket of the so-called 'root request' (the one coming from the *accept* in the father thread). A child thread will not terminate the execution of the father thread unless it encounters an error when trying to close the socket of the root request. This way other threads are still able to concurrently handle (if possible) the request of other clients.
- Every call to an operation that accesses the list of users or messages must be encapsulated with the corresponding mutex lock-unlock. The list of users and messages are thought to be a database, and any operation performed to the database is protected from race conditions. This will not achieve the maximum possible concurrency and an optimum performance, but dealing with numerous mutexes that are private for each user implies a high cost in terms of time and implementation complexity, taking into account that each user is affected by the previous user in the list, as each user has a pointer to the next one. Same happens with the messages.

- Termination signal (ctrl+C) will be detected by the father thread to shut down the server after closing the socket.
- The SEND operation is considered to be successful when the received message is either sent to the destination client or stored in its pending queue, because once the server stores the message, it will be sent unless its holding queue is emptied by an UNREGISTER operation. If any of these conditions happen, then it responds with code 0 back to the sender.
- The server (child threads) will not deal nor handle communication errors that happened when trying to send an acknowledge message to the sender client. This is because as the order in which the messages are sent is guaranteed by the logic of the program, whenever an ACK is received with some associated message ID 'X', every 'X-1' messages that were sent to that particular user are also acknowledged. The only scenario in which this case will not happen is when a user has pending messages and then unregisters from the system, thus deleting all pending messages associated to it and that will never be delivered. In order to make sure that the ACK messages are also delivered to the sender, another pending ACK messages queue has to be implemented a linked to that user, and this functionality is not required for this practice, so assuming ACK losses will slightly simplify the problem while keeping the needed functionality.

2.2. Client Side

client.java

- In order to stick to the abstraction of the functionality of a shell, which is in fact parsing the commands from the console and distributing them among the corresponding function, it was decided to handle the prompts and console prints inside the code of every operation function (*connect, send…*), so that is not the shell the one in charge of disseminating and interpreting the return values coming from the functions that are invoked. However, this led the return values of those functions to be lost and unhandled, so these "enum" return values are left invariant in case any further implementations need them.
- The *printstacktrace* that is usually included in the *catch* blocks are commented in the provided code. This was decided to keep the error retrieval simpler and the console as clean as possible, and will only be commented out from the code for debugging purposes.
- Whenever a CONNECT operation is performed and the server returns back a 0 code (OK), the client is then bound and linked to the user for which the operation was executed. This username is stored in a static global variable *connected_user* so that when we SEND a message to a particular user, the sender username is extracted from this global variable and sent to the server as states the protocol. Hence, it is clear that a SEND operation can only be called if is preceded by a CONNECT command, otherwise this will be handled and returned a SEND FAIL before requesting any connection to the server. Furthermore, there can only be **one connected user at the same time**, so when the client receives an OK from the server when trying to connect,

no other user can perform a CONNECT operation, it will be dropped out before reaching the server. This was preferred over the option of waiting until the response of the server, to avoid the server of successfully connecting two different users called from the same client, in which only one of them will be able to send messages. Considering a multiple user connection inside the same client will increase the implementation cost and is not required in the scope of the practice.

- Several users can be registers from the same client but only one can be connected/bound. With this implementation we achieve the possibility of registering our user from a local machine but then connecting from a different one, in case the machine shuts down or for portability purposes. Similar functioning as other messaging services like Telegram or Whatsapp.
- When the client is shut down with a kill signal (Ctrl+C) from the CLI, the client will check if there is a *connected_user*. If there is any, it will try to perform a DISCONNECT operation with that username before terminating the program. This way it is avoided that a client can perform a CONNECT in one machine, but then exit the client in that machine without disconnecting and try to CONNECT back in another machine (unsuccessfully, because the server will not match the IPs).
- In case of the DISCONNECT operation, either if a code 3 is obtained from the server or an exception is caught during the process because of communication errors, the client will check if the user provided in the command matched the *connected_user*. If it does, it will unbind it from the client as it is assumed in the guidelines that the user is disconnected successfully. By doing this we free the *connected_user* variable for other user to CONNECT and bind to it.

3. Build Guide

In order to build and start using the application it is necessary to follow some simple steps that will be described inside this section. Before executing any command, the submission folder should be unzipped, either using the GUI or the CLI:

```
unzip file.zip -d destination_folder
```

Server

Firstly, the server will be compiled, for which the *make* command will be executed and automatically compile all the server code. When is done, to run the server we will execute:

```
./server –p 12345
```

This will make the server to run in port number 12345, and if executed inside the *guernika* server, it will display the following prompt:

```
s> init server 163.117.142.238:12345
s>
```

Client

The client code can be executed in any machine. Now in order to compile the client we will use:

```
javac client.java
```

Finally, we can put the compiled *client.class* to work using this command:
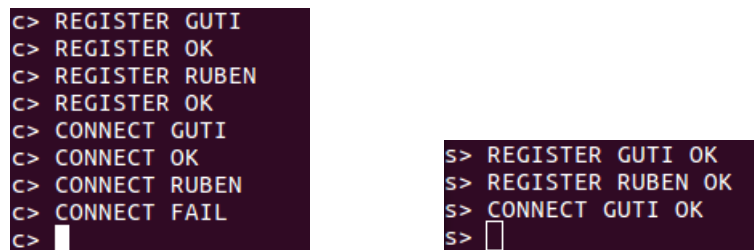
```
java –cp . client –s 163.117.142.238 –p 12345
```

See that we use the IP and port number of the machine where the server is running. Note that these two values are obtained from the first prompt when the server code is executed.

At this point, we can execute any desired command within the client and start using the application.

4. Test Plan

This section will not contain most of the trivial tests but instead include a couple of tests that are of particular interest based on the design decisions taken for implementation, including some complex tests that require slight alteration of the normal functioning of the code to simulate other factors like network congestion or sudden shutdowns. However, the fact that they are not included in the present report is basically because of the page number limitation and the extensive focus on the design decision making explanation made in section 2.

Here we check the functionality of the client user binding. It can be seen that several users can be registered from the same client, but only one can connect and bind to it. The subsequent CONNECT requests do not reach the server (see that no "CONNECT RUBEN FAIL" is obtained in the server).

```
c> REGISTER GUTI
c> REGISTER OK
c> REGISTER RUBEN
c> REGISTER OK
c> CONNECT GUTI
c> CONNECT OK
c> CONNECT RUBEN
c> CONNECT FAIL
c>
```

```
s> REGISTER GUTI OK
s> REGISTER RUBEN OK
s> CONNECT GUTI OK
s>
```

Figure 1: client side (left) and server side (right)

The following test will be run in the local virtual machine. The reason is that it was desired to slightly modify the code to simulate that a user disconnects or shuts down while it is receiving the list of pending messages. So a *sleep* of 10 seconds is introduced in the loop that sends the queue of messages (to introduce a delay between messages) and a message in the server that indicates when the thread sleeps. Then we have two subjects, RUBEN and CARLOS. CARLOS will be sending three first messages to RUBEN while he is disconnected, and then RUBEN will connect to receive the pending messages. But after the first message he disconnects. The server is able to detect that and avoid sending the rest of the messages. Then CARLOS sends another message after the disconnection to see if it still appends to the pending queue. When RUBEN connects back again, he recovers all the messages from the disconnection, included the latter mentioned.

Figure 2: Log shown in the server

Figure 3: At the left, sequence of commands for user CARLOS. At the right, sequence followed by RUBEN and the disconnection process shown in red

With this test we prove the correct functioning of several aspects: sending and receiving messages, storing messages, proper acknowledgement to the sender, correct behavior of the queue, sending of pending messages and correct responsiveness upon sudden failure.

Other procedures like synchronization tests require a more intensive usage of the application by several clients to make sure that works properly. This test requires either of a script that runs multiple clients with hundreds of commands at the same time; or code and logic implementation for looping requests; or collaboration of multiple users. None of the three options could be performed due to time constraints of the practice.

5. Conclusions

No hesitation in stating that this could be the most challenging lab assignment that we have ever performed in the university, but yet being the most interesting. It is definitely a good practice to make the student understand how the lowest level of abstraction of a distributed system works. However, it is well known that there are other technologies that ease the implementation of such systems like RPCs, but this leads to the designers to think of almost every aspect that takes part in the communication between a client and a server.

As this is a new topic that we have never dealt with, there were lots of standard library functions that gave a bit of a headache the first time you use them, and that hinders a lot the implementation flow. Using a dynamic linked list that is attached to each node of another dynamic linked list can be particularly tedious to be implemented in C, especially when making use of double pointers.

Finally, as a critical review of the assignment it is worth mentioning that the protocol and guidelines defined for the practice does not take into account some security aspects like unregistering a user from a different client that is being used in another client, but this would increase the complexity of the assignment and require to dedicate time in implementations that are not of particular interest regarding the scope of the practice, which is in fact the communication process. That is why some security mechanisms were not implemented in the development of the application.