

# Assignment 2

Design and implementation of a  
communication system with RPCs and web  
services

7<sup>th</sup> of May, 2017

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

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

# Table of Contents

|  |          |
|--|----------|
| <b>1. Introduction.....</b>                    | <b>2</b> |
| <b>2. Code Description.....</b>                | <b>2</b> |
| 2.1. Improvements over Lab 1 .....             | 2        |
| 2.2. Storage service .....                     | 3        |
| 2.2.1. Server .....                            | 3        |
| 2.2.2. Monitor (client).....                   | 3        |
| 2.2.3. Messaging server (client) .....         | 3        |
| 2.3. MD5 service .....                         | 4        |
| 2.3.1. Server .....                            | 4        |
| 2.3.2. Client (same as the previous Lab) ..... | 4        |
| <b>3. Build Guide.....</b>                     | <b>4</b> |
| <b>4. Test Plan.....</b>                       | <b>6</b> |
| 4.1. General functionality of the system.....  | 6        |
| 4.2. Disconnection of MD5 service.....         | 7        |
| 4.3. Disconnection of Storage service.....     | 7        |
| 4.4. Reconnection of both services .....       | 8        |
| <b>5. Conclusions .....</b>                    | <b>9</b> |

# 1. Introduction

The aim of this assignment is to get familiar with the utilization of SUN's RPC framework and JAX-WS API for the creation of a server (built with the former) and the creation of a web service (built with the latter), and to compare the implementation procedures with the first assignment. In this first assignment, the complexity relies on defining and managing sockets between the client and server, but now this complexity is hidden by these frameworks, so that the implementation efforts can be directed towards defining the services that both offer, instead of dealing with the connectivity at a lower level as well.

## 2. Code Description

This section will contain a description of the code structure and design decisions that were made during the implementation process. Additionally, there will be stated the improvements that were made over the previous submission of this lab practice to make the code more robust.

### 2.1. Improvements over Lab 1

There were two major problems in the submission of the previous assignment:

- Makefile was not included in the submission folder.
- `NullPointerException` that appeared when trying to execute a `DISCONNECT` operation from the client without having received any message.

The first problem will be solved by simply including the Makefile in the submission folder of the current assignment. However, we wanted to go beyond that problem and include the compilation of **all files** with one *make* command, including the Java files. So now, all the files will be compiled at once in order to simplify the compilation process at its maximum.

Secondly, during the development of this assignment it was noticed that when a `DISCONNECT` operation was made before the client's listening thread had received any message (and thus not created any socket for the connection), the cleanup process that was triggered from this `DISCONNECT` command would try to access a variable with value `NULL`, in particular, this socket variable. This issue is solved in this version of the Lab.

In addition to this, there was a complication that made the `DISCONNECT` process a little tedious when the messaging service shuts down or is terminated and rerun again. If a user name was bound to the client (as explained in the previous report), then the `DISCONNECT <username>` would not work because the messaging server will not have any users in its memory (as it has been terminated), and the user would not be unbound from the client, thus preventing its disconnection and reconnection without terminating the client as well. So this was solved so that when a user is bound and connected to the client, and the server terminates for any reason, now the `DISCONNECT` will unbind the client correctly. This way we can re-register and reconnect again without terminating the client.

Furthermore, other minor fixes were included so that the console prints will look a little more consistent, but that did not hinder the main functionalities of the application.

## 2.2. Storage service

### 2.2.1. Server

In order to create the storage service server, the .x interface was created based on the following criteria:

- *void init*: this service will traverse the list of users and messages (if any) and free the memory resources associated to all the nodes for a full cleanup. Subsequently, it will initialize the pointer to the head of the list to NULL. It is a void method because it cannot fail for any reason.
- *int store*: it will receive all the necessary information relative to the message (sender, receiver, message id, message body and MD5 hash of the message) and will store it in memory. It will return a -1 if any error occurred in the process (like full memory). Returns 0 if the message was stored correctly.
- *int getnummessages*: retrieves the total number of messages sent by the input user. Returns -1 if the user was not found in the list.
- *struct getmessage*: based on an input user and message ID, retrieves a structure containing the message body and its associated MD5. If the message could not be found in memory, then it will send back two empty strings (length 0).

The implementation of the lists is similar to the one derived in the messaging server. It is created a first list of users, in which each node corresponds to a sender. Then, for every sender, a list of sent (and stored) messages is appended. This is based on the reasoning that the number of users/senders is much smaller than the number of messages sent, and that messages IDs can be repeated for different senders. So the user list is first traversed, and when the sender is found, its message list will be traverse as well, thus reducing the number of total accesses to the list and achieving consistency in the message retrieval.

When the storage service server is executed, the IP address of the machine where is running is prompted, and the *init* service will be self-invoked, so that it initializes the user list to NULL. In order to include this functionality, the autogenerated svc file had to be slightly modified.

### 2.2.2. Monitor (client)

The monitor process is the client for the storage service. It was decided to include an additional parameter to provide the IP of the server to connect to. So the command will now receive three parameters: <storage\_service\_IP> <client\_name> <message\_id>.

When the monitor calls for the *getmessage* service, it will first check the length of the response strings. If it is 0, then an error indicating that the message does not exist is shown. It will also detect if the function could not be completed nor established connection with the server.

Additionally, it includes a fragment of commented code with the invocation of the *getnummessages* service, used for debugging and checking purposes during the development of the assignment.

### 2.2.3. Messaging server (client)

The messaging server acts as client of the storage service whenever a message is received from and end user. When this happens, it will try to send the message to the destination user (or store it if the user is not connected to the server). If the message was **successfully** sent or stored in the server, this will invoke the *store* service for the storage server to append it to the list. Note that if any error occurred in the messaging server when trying to send or store the message, this will not be

considered as sent nor stored and the service will not be called. This was decided because the storage service represents a “log” of all the messages sent or stored through the messaging server, so if an error happens during any of these processes, the messages will not be considered as such.

## 2.3. MD5 service

### 2.3.1. Server

The implementation of the MD5 server was quite simple, therefore not needing of elaborated design decisions. The only decision taken here was the definition of the URL of the service, which consists of the IP address of the machine, the port number 8080, and the termination ‘/ws/md5’ to give it a more identifying name. So in the end, an example of URL will look as follows: <http://127.0.0.1:8080/ws/md5>

### 2.3.2. Client (same as the previous Lab)

As the server takes the IP of the machine as address for the endpoint where the web service is published, this led to a change in the command that executed the client of the previous lab. Now, the client will be run including the ‘-w <IP:port>’ option, where -w stands for web service, and the IP-port tuple are the ones published by the server. If the URL was the one from the example of section 2.3.1, the command parameter would be ‘-w 127.0.0.1:8080’ (See building section 3 for more details).

Once the client parses the parameters, it will store this address in a variable, and it will try to connect to the corresponding web service every time a SEND command is executed. If the MD5 service is unavailable or raises an Exception for any error, the service will be considered as unavailable, and the message will not be sent to the messaging server (this will be shown in the test section 4).

The message listening thread is also modified to expect an additional message from the messaging service containing the MD5 of the message to be received from other client.

## 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 ssdd_p2_100291121_100292107.zip -d <destination_folder>
```

When the files are extracted, with an opened terminal navigate to the root *destination folder*. Make sure that the Makefile is located in that folder, if not, navigate to the folder containing it. Then execute the command **make**. This will start the compilation of all the necessary files. After a few seconds, it will finish and all the files will be compiled and ready to be executed. Now open 5 terminals and navigate to the same root destination folder:

### Terminal 1

Firstly, we will execute the Java web services with the command:

```
java -cp . md5.server.endpoint.MD5Publisher
```

Then the terminal will show a prompt indicating that the service is deployed and the address of the web service, as below:

```
^Ca0292107@guernika:~/ssdd_p2_100291121_100292107$ java -cp . md5.server.endpoint
MD5Publisher
Publishing MD5 service at endpoint: http://163.117.142.238:8080/ws/md5
```

## Terminal 2

Then, we will execute the storage service server:

```
./storeServer
```

And again, another prompt will indicate the IP address in which the service is running:

```
a0291121@guernika:~/ssdd_p2_100291121_100292107$ ./storeServer
Store service running at: 163.117.142.238
```

## Terminal 3

To run the messaging server:

```
./server -p <port_number> -s <storage_service_ip>
```

Where the <port\_number> will be the port number in which the service will be running, and the <storage\_service\_ip> is the IP address where the storage server is running, which in the example of Terminal 2 would be '163.117.142.238'. Then it will prompt a message with the IP and port number where the server process is running:

```
a0291121@guernika:~/ssdd_p2_100291121_100292107$ ./server -p 12345 -s 127.0.0.1
s> init server 163.117.142.238:12345
```

## Terminal 4

The client will be executed with:

```
java -cp . client -s <server_ip> -p <port_number> -w <md5_service_ip:port>
```

Where <server\_ip> and <port\_number> are the ones prompted in Terminal 3, in this example would be '163.117.142.238' and '12345', respectively; and <md5\_service\_ip:port> are the IP and port number shown in Terminal 1, in this case '163.117.142.238:8080'. It is important to keep the same format <IP:port>. This terminal can be replicated in parallel as many times as desired to run other clients.

## Terminal 5

Finally, the message monitor is run by introducing:

```
./monitor <storage_service_ip> <client> <message_id>
```

Where <storage\_service\_ip> is the one obtained in Terminal 2; and the <client> and <message\_id> are the name of the client and message ID that he/she sent.

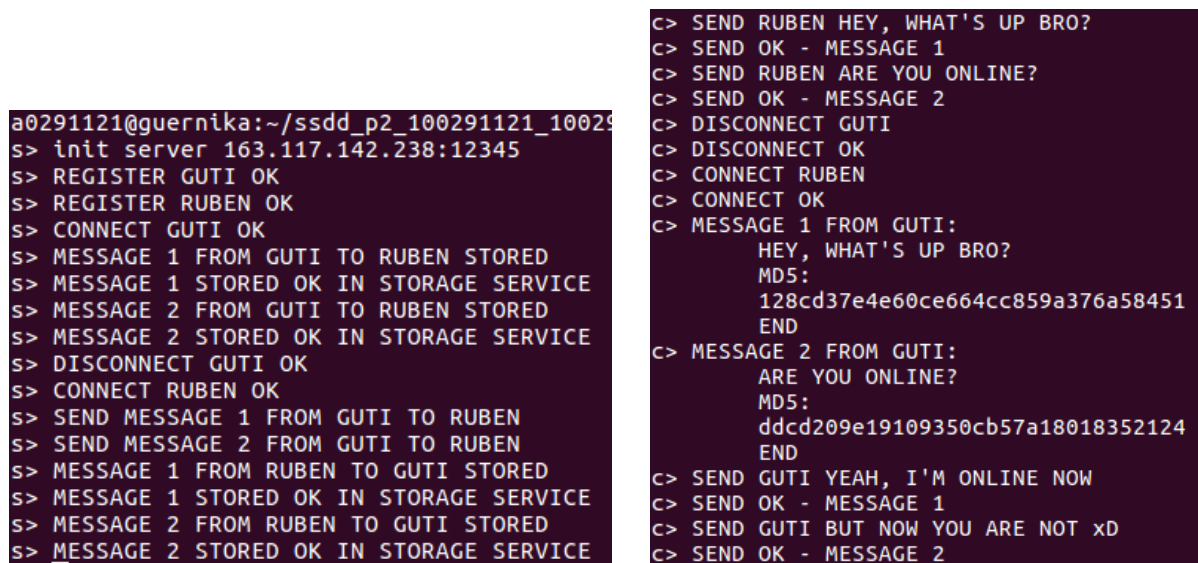
***\*\*IMPORTANT NOTE: IF THE PROGRAMS ARE TO BE RUN IN THE SAME MACHINE, IT IS RECOMMENDED TO USE LOOPBACK IP ADDRESSES (INSTEAD OF THE PUBLIC IP) TO AVOID CONNECTIVITY PROBLEMS\*\****

## 4. Test Plan

This section includes the tests run to check the correct functioning of all the acting parts in the distributed system.

### 4.1. General functionality of the system

From the client, we run a sequence of commands simulating the normal use of the application:



```
a0291121@guernika:~/ssdd_p2_100291121_100291121$ s> init server 163.117.142.238:12345
s> REGISTER GUTI OK
s> REGISTER RUBEN OK
s> CONNECT GUTI OK
s> MESSAGE 1 FROM GUTI TO RUBEN STORED
s> MESSAGE 1 STORED OK IN STORAGE SERVICE
s> MESSAGE 2 FROM GUTI TO RUBEN STORED
s> MESSAGE 2 STORED OK IN STORAGE SERVICE
s> DISCONNECT GUTI OK
s> CONNECT RUBEN OK
s> SEND MESSAGE 1 FROM GUTI TO RUBEN
s> SEND MESSAGE 2 FROM GUTI TO RUBEN
s> MESSAGE 1 FROM RUBEN TO GUTI STORED
s> MESSAGE 1 STORED OK IN STORAGE SERVICE
s> MESSAGE 2 FROM RUBEN TO GUTI STORED
s> MESSAGE 2 STORED OK IN STORAGE SERVICE

c> SEND RUBEN HEY, WHAT'S UP BRO?
c> SEND OK - MESSAGE 1
c> SEND RUBEN ARE YOU ONLINE?
c> SEND OK - MESSAGE 2
c> DISCONNECT GUTI
c> DISCONNECT OK
c> CONNECT RUBEN
c> CONNECT OK
c> MESSAGE 1 FROM GUTI:
    HEY, WHAT'S UP BRO?
    MD5:
    128cd37e4e60ce664cc859a376a58451
    END
c> MESSAGE 2 FROM GUTI:
    ARE YOU ONLINE?
    MD5:
    ddc209e19109350cb57a18018352124
    END
c> SEND GUTI YEAH, I'M ONLINE NOW
c> SEND OK - MESSAGE 1
c> SEND GUTI BUT NOW YOU ARE NOT XD
c> SEND OK - MESSAGE 2
```

First of all, notice that the DISCONNECT commands did not trigger any NullPointerException, proving that the problem with the previous assignment is solved.

On the other hand, we can observe in the right image (client) that two clients, RUBEN and GUTI exchanges messages. First, GUTI sent RUBEN two messages while he was offline, and the server (image at the left) shows that they were correctly stored in the storage service. Then RUBEN comes back online, receives the two pending messages correctly and sends another two messages back to GUTI after he DISCONNECTED, and again the server (at the left) shows that they were correctly stored in the storage service server.

Now to check if the messages were correctly stored in the storage service server, we make use of the monitor, asking for the messages that GUTI and RUBEN sent each other. Note that when we ask for message with ID=3 that RUBEN sent, the monitor tells us that it does not exist.

```

guti@guti-VirtualBox:~/workspace/Distr
$ ./monitor 10.0.2.15 GUTI 1
MESS: HEY, WHAT'S UP BRO?
MD5: 128cd37e4e60ce664cc859a376a58451
guti@guti-VirtualBox:~/workspace/Distr
$ ./monitor 10.0.2.15 GUTI 2
MESS: ARE YOU ONLINE?
MD5: ddc209e19109350cb57a18018352124
guti@guti-VirtualBox:~/workspace/Distr
$ ./monitor 10.0.2.15 RUBEN 1
MESS: YEAH, I'M ONLINE NOW
MD5: d12830e3f7bdf81911f1f584211a9a9c
guti@guti-VirtualBox:~/workspace/Distr
$ ./monitor 10.0.2.15 RUBEN 2
MESS: BUT NOW YOU ARE NOT xD
MD5: 419f48f65ff8b02645c30ca4ac675af2
guti@guti-VirtualBox:~/workspace/Distr
$ ./monitor 10.0.2.15 RUBEN 3
ERROR , MESSAGE DOES NOT EXIST

```

## 4.2. Disconnection of MD5 service

For this test, we shut down the web service that serves the MD5 hash of the messages and try sending a message:

|   |  |
|---|--|
| <pre> s&gt; init server 163.117.142.238:12345 s&gt; ERROR, STORAGE SERVICE UNAVAILABLE s&gt; REGISTER GUTI OK s&gt; REGISTER RUBEN OK s&gt; CONNECT GUTI OK s&gt;  </pre> | <pre> c&gt; REGISTER GUTI c&gt; REGISTER OK c&gt; REGISTER RUBEN c&gt; REGISTER OK c&gt; CONNECT GUTI c&gt; CONNECT OK c&gt; SEND RUBEN LOOK, NO MESSAGE IS SENT TO SERVERS c&gt; ERROR , SEND FAIL / ERROR IN MD5 c&gt;  </pre> |
|---|--|

Now GUTI tries to send a message to RUBEN, but the client prompts “ERROR, SEND FAIL / ERROR IN MD5” and it can be observed in the left image (server) that the message was not received at the messaging service server-side.

## 4.3. Disconnection of Storage service

This time the Storage service is shut down, and RUBEN tries to send another message:

|  |   |
|--|---|
| <pre> s&gt; MESSAGE 3 FROM RUBEN TO GUTI STORED s&gt; ERROR, STORAGE SERVICE UNAVAILABLE s&gt;  </pre> | <pre> c&gt; SEND GUTI DON'T IGNORE ME c&gt; SEND OK - MESSAGE 3 c&gt;  </pre> |
|--|---|

The message was correctly received at the messaging service server, but a prompt is shown indicating that the storage service is unavailable. If we try to monitor any message, an error will be shown. Additionally, as the messaging service server tries to perform an *init* call when is first launched, it should also retrieve an availability error, as it happens below:



```
a0292107@guernika:~/ssdd_p2_100291121_100292107$ ./monitor 127.0.0.1 RUBEN 2  
ERROR , SERVICE NOT AVAILABLE
```

```
a0291121@guernika:~/ssdd_p2_100291121_100292107$ ./server -p 12345 -s 127.0.0.1  
s> init server 163.117.142.238:12345  
s> ERROR, STORAGE SERVICE UNAVAILABLE  
s>
```

## 4.4. Reconnection of both services

Now both services are rerun and another message is sent:

|   |  |
|---|--|
| <pre>s&gt; init server 163.117.142.238:12345<br/>s&gt; ERROR, STORAGE SERVICE UNAVAILABLE<br/>s&gt; REGISTER GUTI OK<br/>s&gt; REGISTER RUBEN OK<br/>s&gt; CONNECT GUTI OK<br/>s&gt; MESSAGE 1 FROM GUTI TO RUBEN STORED<br/>s&gt; MESSAGE 1 STORED OK IN STORAGE SERVICE<br/>s&gt; DISCONNECT GUTI OK<br/>s&gt; CONNECT RUBEN OK<br/>s&gt; SEND MESSAGE 1 FROM GUTI TO RUBEN<br/>s&gt;</pre> | <pre>c&gt; SEND RUBEN LOOK, NO MESSAGE IS SENT TO SERVERS<br/>c&gt; ERROR , SEND FAIL / ERROR IN MD5<br/>c&gt; SEND RUBEN NOW BOTH SERVICES ARE CONNECTED<br/>c&gt; SEND OK - MESSAGE 1<br/>c&gt; CONNECT RUBEN<br/>c&gt; CONNECT FAIL<br/>c&gt; DISCONNECT GUTI<br/>c&gt; DISCONNECT OK<br/>c&gt; CONNECT RUBEN<br/>c&gt; CONNECT OK<br/>c&gt; MESSAGE 1 FROM GUTI:<br/>    NOW BOTH SERVICES ARE CONNECTED<br/>    MD5:<br/>    7662972c2608b1c9965a7b5cf1912c88<br/>    END<br/>c&gt;</pre> |
|---|--|

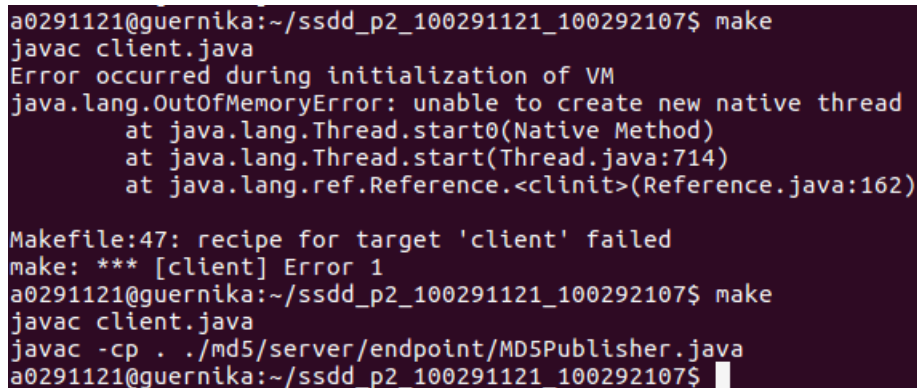
It can be observed that at the beginning of both server (left) and client (right) the services are not yet available. Then these are reconnected and the message is sent. Note that the MD5 is correctly calculated and the server stored in the storage server:

```
a0291121@guernika:~/ssdd_p2_100291121_100292107$ ./monitor 127.0.0.1 GUTI 1  
MESS: NOW BOTH SERVICES ARE CONNECTED  
MD5: 7662972c2608b1c9965a7b5cf1912c88
```

## 5. Conclusions

Having thoroughly tested the application, it can be concluded that this version seems much more robust than the one previously implemented, and that the usage of SUN's RPC framework and JAX-WS API greatly simplifies the implementation effort. As with any other framework or tool, you first need to get used to it, but once the mechanics are understood, it results in a very easy way to create and deploy distributed system services. Hence, the difficulty of this assignment resides in understanding the core functioning of RPC and the commands to generate the stubs and example codes. The rest remains in the same level of difficulty as in the previous lab, and as it was already solved, the problems could be easily surpassed.

However, there are a few important notes to be taken aside. The first one is related to the limitations that *guernika* imposes on the sessions that it provides. Seems like there is a memory limitation when trying to execute several Java programs with running threads. Under some situations, executing either *client.class* or *md5.server.endpoint.MD5Publisher* will prevent the other from executing. The error prompted is related to the initialization of the Java VM and the number of concurrent threads that are allowed to be executed in parallel. This will also prevent the code from being compiled if there are Java threads already running. When all the conflicting Java processes are shut down, the compilation is preformed correctly. The image below shows an example of the error that is prompted:



```
a0291121@guernika:~/ssdd_p2_100291121_100292107$ make
javac client.java
Error occurred during initialization of VM
java.lang.OutOfMemoryError: unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at java.lang.ref.Reference.<clinit>(Reference.java:162)

Makefile:47: recipe for target 'client' failed
make: *** [client] Error 1
a0291121@guernika:~/ssdd_p2_100291121_100292107$ make
javac client.java
javac -cp . ./md5/server/endpoint/MD5Publisher.java
a0291121@guernika:~/ssdd_p2_100291121_100292107$
```

It is worth mentioning that this error **never occurred** in the machine where this code was developed, so the reason behind this relies on the limitations that are established on *guernika* servers for the Java Runtime VM.

Lastly, to avoid any conflict, if we intend to run all the programs in the same machine, it is recommended to use loopback IP addresses (127.0.1.1, 10.0.1.2, etc.). It was checked that using the machine's public IP address as parameter of the execution commands stated at section 3, resulted in connectivity issues when used for connecting all the processes running in the same machine (same IP). However, when using the external public IP from different machines, this issue did not appear. This was checked during the first days of implementation at the laboratory classrooms, in which the system was distributed within the class machines, but no screenshot was taken as proof. Moreover, this test could not be driven during the weekend because the lab rooms are shut down, and no SSH connection could be made from the *guernika* session.