Distributed Systems

Design and implementation of a publisher/subscriber system with POSIX sockets

Fernando García Sanz (100346043)

David Eguizabal Pérez (100303552)

May 12, 2019

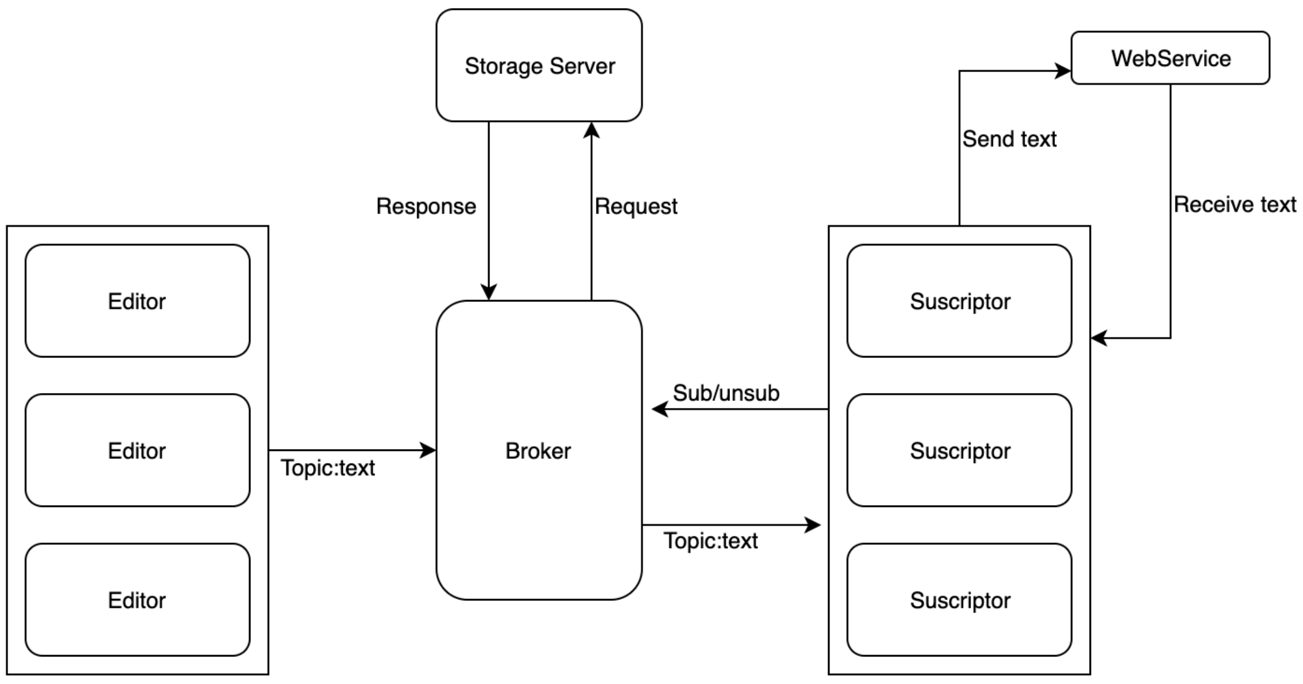
uc3m | Universidad Carlos III de Madrid

Contents

1 Introduction

In this assignment we will build a publisher/subscriber app. The development of the project has been divided into three parts, the first one will be implemented using sockets, the second will implement over the first stage the RPC ONC-RPC model to enhance the application functionalities, and the last one will implement a web service to parse the different messages a subscriber receives.

2 System Design



The main part of the system will be formed by four entities: the editor, the broker, the subscriber and the storage server. Editor, broker and storage server will be coded using C language and the subscriber will implement Java.

The editor function is to publish messages related to a specific topic, both being sent to the broker, which will be the middleman between editor and subscriber. Once the broker receives topic and message, it will process them, communicating to the storage server if necessary, and deliver them to each of the subscribers previously signed up to that specific topic.

The subscriber will then receive a message from a subscribed topic once it has been published and processed by the broker, showing it in the screen to the final user after being processed by a third party implemented web service. A subscriber can also send messages to the broker according to subscribing to or unsubscribing from a certain topic. Once the broker receives these messages, it will perform the pertinent operations to fulfill the request, and will let it know to the subscriber once the operation has been completed.

3 System Components

3.1 Editor

The editor entity is able to deal with topics of until 128 characters and texts of 1024 characters. The program will first check that the program operation is correct according to the number of arguments, before performing any operation over them. They has to be a total of 9, where 4 are decided by the operator: the IP address and port of the server, the topic to publish and the message to be sent.

The editor will then create a client socket to perform a connection to the server using the input arguments. Error check functions are included to be aware of any problem that could happen during the process. If the connection has been successfully performed, topic and message arguments are processed.

The character ”\0” is included at the end of each of the arguments, which will act as separator between them. Then, three messages are sent to the broker using the previously declared socket, the operation code, which will be ”PUBLISH”, and the ones correspondent to both topic and text once processed.

3.2 Broker

The broker is the main component of the system, with the capacity of receiving and processing requests of both subscriber and editor and send information to the subscriber too.

When the broker is initialized the port to process the different requests is provided by the operator as an argument. By means of this port, it is possible to initialize a socket on it, which will be in charge of listening to all the requests sent to the broker. Once a request is listened and accepted by the socket, a specific thread is created to process it. This thread will execute a function called *manage_request*, where this request will be processed and solved.

Depending on the operation code received in the message three different operations can be performed:

- **PUBLISH**: If the operation code is ”PUBLISH”, it is known that the request comes from the editor. Both topic and text sent are temporally stored in the broker to be processed. Then, it is checked if the received topic is a new one or it corresponds to a previously sent topic during this broker session, since all different topics are saved in an array. If it is a new one, it will be added to that list.

  After this, the topic and text are sent to the storage server component, which will be later explained, to be stored for future usage. Once this operation has been properly performed, the broker will search into a list in which all subscribers are stored those who are subscribed to that topic, and it will then recover from that list both IP address and port to which the message has to be sent to the subscriber and open a new socket using this information, sending topic and text. Once the entire list is traversed, the execution of the thread finishes.

- **SUBSCRIBE**: If the operation code is ”SUBSCRIBE”, it is known that the request comes from the subscriber. The broker will then read the topic wanted to be subscribed to and will check that it exists into its list of existent topics. If it does not, it will send the code ”1” back to the subscriber, indicating that the request is impossible to be processed.

  If it does exist, it will get subscriber IP address from the socket and the port to be replied from the request. The broker will then check if this tuple exists in its list of subscribers. If it does not, it will create a new node with these parameters and the topic to be subscribed as the first topic this node in the list is subscribed. If the node does previously exist but it was not subscribed, the list of topics this node is subscribed will be updated.

  Once this is done, the code ”0” will be sent to the subscriber, indicating that everything went well.

  After it, a call to the storage server is done by means of RPCs, retrieving from it the last message which was sent by the subscribed topic. Then, a socket is generated, connecting to the address and port previously sent by the subscriber. The topic and last sent message are sent to the specified subscriber’s address and port and then, the execution of the thread is finishes.

- **UNSUBSCRIBE**: If the operation code is ”UNSUBSCRIBE”, it is known that the request comes from the subscriber. The broker will then read the topic wanted to be unsubscribed from and will check that it exists into its list of existent topics. If it does not, it will send the code ”2” back to the subscriber, indicating that the request is impossible to be processed.

  If it does exist, it will get subscriber IP address from the socket and the port to be replied from the request. The broker will then check that the node correspondent to this tuple is subscribed to the topic. If it is not, it will send the code ”1” back to the subscriber, indicating that it was not subscribed to the topic before. On the other hand, if it is subscribed to the topic, it will remove it from the list of subscribed topics of that node in the list of subscribers the broker manages. Once this is done, the execution of the thread finishes.

3.3 Subscriber

The subscriber is the client which will receive each one of the messages from a topic it is subscribed when they are send by the editor and processed by the broker.

When executed, it will display a kind of command line, in which the operator is capable of introducing several commands:

- **SUBSCRIBE <topic>**: With this command, the subscriber will send a request to the broker by means of a socket asking for subscribing to a topic. It will send the operation code, ”SUBSCRIBE”, the topic to be subscribed to, and the port in which the messages are going to be received. These elements are separated by the symbol ”\0”, so they can be properly read by the broker. Depending on the response sent by the broker, if it is ”1”, the subscriber will know that it was impossible to be subscribed to that topic; if it is ”0”, the subscriber will know that it is now subscribed to that topic and that the next communications published by the editor will be received in the specified port.
- **UNSUBSCRIBE <topic>**: With this command, the subscriber will send a request to the broker by means of a socket asking for unsubscribing from a topic. It will send the operation code ”UNSUBSCRIBE”, the topic to be unsubscribed from, and the port used to receive communications from that topic. Depending on the response sent by the broker, if it is ”2”, the subscriber will know that it was impossible to be unsubscribed from that topic for some reason; if it is ”1”, the subscriber will know that it was already not subscribed to that topic; and if it is ”0”, the subscriber will know that the operation was properly performed.
- **QUIT**: Writing this into the command line, the program finishes and the subscriber is closed.

At the same time, the subscriber creates a server socket in charge of listening to communications from the broker by means of a random free port, which will be operated by a new thread. This socket will be in charge of receiving all communication from all subscribed topics of that subscriber. Whenever a communication is accepted, a new thread will be generated to process it. It will receive from the broker both topic and text. Before displaying the received message, the text is processed.

The text is divided into the different words which conform it, and only those which can be parsed as numbers are sent to a third party web service, which will provide the word representation of those numbers, building the same text which was received, but substituting all numbers by their representation in words. Once this is done, both topic and text are displayed in the command line.

3.4 Storage Server

The storage server is a component developed using RPCs, which is in charge of initializing the elements necessary to store all messages sent by each of the topics in different files.

It will first create the folder */data* in case it does not exist when the broker is executed, before it performs any other function. When a new topic is published, it will create a *.txt* file with the topic name, writing on it in different lines each message published by that topic once the file has been created.

If a subscriber subscribes to a topic, the storage server is in charge of reading the last line of the file correspondent to that topic, which will be the last published message, and retrieving it to the broker so it can send it to the subscriber.

4 RPCs Implementation

For the implementation of the RPCs, a *.x* file was written, and by means of the command *rpcgen -a -N -M* all necessary files are generated, being only needed to complete the code of the *storage_server.c* file to perform the required operations, and call it from the broker, which will act as a client.

5 Web Service Implementation

For the web service implementation, it was only needed to use the command *wsimport –keep <web service URI>* to import all necessary components and perform a request/response command to send the number in the proper format and receive its text conversion.

6 How to Compile and Execute

This section can be also read in the included README file, but it will also been explained here:

In order to deploy and execute the application:

- Execute the Makefile writing ”make” in the command line.
- Execute the following two programs first:
  - Execute the storage server for RPCs writing ”./storage_server” in the command line.
  - Execute the broker server writing ”./broker -p <port-number>” in the command line.
- Once these two programs are running, execute the two following ones:
  - Execute the editor (content publisher) writing ”./editor -h <ip-address> -p <port-number> -t <topic> -m <message>” in the command line.
  - Execute the java client writing ”java -cp . suscriptor -s <ip-address> -p <port-number>” in the command line.
- Inside the suscriptor command line:
  - SUBSCRIBE <topic>
  - UNSUBSCRIBE <topic>
  - QUIT (to finalize the execution)
- In case of wanted to go back to the pre-deployment stage write ”make clean” in the command line.
- To regenerate the files from the .x file, execute ”rpcgen -a -N -M storage.x”, but not recommended since the server class will be entirely erased, so it has to be written again.

7 Battery of Tests

To prove the correct performance of the implemented system, the following tests have been performed:

- **Test 1**: The storage server is first executed. Then, the broker in executed in the port 4200. After that, a subscriber is executed once provided the same port and IP address used by the broker as arguments. Finally, an editor starts with the same parameters and a topic and text to be published.

  The subscriber is then subscribed to the previous topic published by the editor:

  /* The input is */

```
c> SUBSCRIBE <topic>
/* The output is */
c> SUBSCRIBE OK
c> MESSAGE FROM <topic> : <text>
```

The result of the test is satisfactory, the subscriber gets the message and displays it correctly, substituting any numeric character by its word representation.

- **Test 2**: When all services being executed, unsubscribe from a previously subscribed topic:

  ```
  /* The input is */
  c> UNSUBSCRIBE <topic>
  /* The output is */
  c> UNSUBSCRIBE OK
  ```

  The result of the test is satisfactory. Now, if new messages are sent by this topic, the subscriber will not receive them.

- **Test 3**: When we unsubscribe from an existent topic but to which we were not subscribed:

  ```
  /* The input is */
  c> UNSUBSCRIBE <topic>
  /* The output is */
  c> TOPIC NOT SUBSCRIBED
  ```

  The result of the test is satisfactory, since we were not subscribed, the proper answer is received.

- **Test 4**: When we subscribe to a non-existent topic:

  ```
  /* The input is */
  c> SUBSCRIBE <topic>
  /* The output is */
  c> SUBSCRIBE FAIL
  ```

  The result of the text is satisfactory, the proper answer is received.

- **Test 5**: When we unsubscribe from a non-existent topic:

  ```
  /* The input is */
  c> UNSUBSCRIBE <topic>
  /* The output is */
  c> UNSUBSCRIBE FAIL
  ```

  The result of the text is satisfactory, the proper answer is received.

- **Test 6**: If */data* folder is deleted before executing the broker, once it is executed, the folder will be created back so it can perform its operations properly.

- **Test 7**: Use a big amount of characters, but lower than the maximum allowed by the system, for the topic to be published. The system behaves normally.

- **Test 8**: Use a big amount of characters, but lower than the maximum allowed by the system, for the text to be published. The system behaves normally.

- **Test 9**: Use a big amount of characters, greater than the maximum allowed by the system, for the topic to be published. The system returns a warning about the topic size and no operation is performed, so the behaviour is the expected one.

- **Test 10**: Use a big amount of characters, greater than the maximum allowed by the system, for the text to be published. The system returns a warning about the text size and no operation is performed, so the behaviour is the expected one.

- **Test 11**: Use a big amount of characters, but lower than the maximum allowed by the system, for the topic to be subscribed. The system behaves normally.

- **Test 12**: Use a big amount of characters, but lower than the maximum allowed by the system, for the text to be subscribed. The system behaves normally.

- **Test 13**: Use a big amount of characters, greater than the maximum allowed by the system, for the topic to be subscribed. The system returns a warning about the topic size and no operation is performed, so the behaviour is the expected one.

- **Test 14**: Several subscribers are executed in parallel. They all perform their operations properly and receive the expected responses, being able to receive messages of the subscribed topic at the same time each, so the test is successful.

- **Test 15**: Several editors are executed in parallel. They all perform their operations properly, the topic files are created and the message is delivered to the subscribed entities, so the test is successful.

- **Test 16**: Perform any operation from subscriber when broker is not executed. Error in the connection to broker is returned, so the test is successful.

8 Conclusion

This assignment has been a complete set of practices which covers many different implementations of distributed system technologies, so we think that it has helped a lot to understand each of the tools used, their benefits and drawbacks.

The difficulty level of the practice has been descendant, thus, when the first part was up and running, the following two just consisted of implemented new small variations and functionalities, so it has also been useful to understand the ”modularity” of these kind of programs, as well as their versatility.

Even though, the development process was not easy. Many different kind of errors appeared due to the huge amount of technologies used together, such as leakages of memory, non-closed sockets, connection errors between components, etc. Nevertheless, with some investigation work by our part, they could be eventually solved, reaching to fully complete the assignment.

Summing up, it has been quite a complete assignment which covered most parts of what has been learned during the course, and we are quite pleased about the job done.