FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

SDLE - Reliable Pub/Sub Service

Gustavo Sena
up201806078@edu.fe.up.pt
João Rosário
up201806334@edu.fe.up.pt
Iohan Sardinha
up201801011@edu.fe.up.pt
João Rocha
up201806261@edu.fe.up.pt

Master in Informatics and Computing Engineering

November 25, 2021

Overview

The purpose of this project was to implement a reliable PUB/SUB service, were a user could subscribe to a topic of interest, write and read messages to or from topic.

As an extra in this implementation, the subscriber can either, read a message from a specific topic or the last unread message from any topic it's subscribed to.

The service is durable and guarantees an "exactly-once" delivery.

The group project was developed in Python 3.9, using the libzmq python binding as required, for the use of ZeroMQ socket implementation.

To accomplish so, four methods were implemented:

- **Put**: Writes a message into a topic
- **Get**: Reads a message from a topic
- **Subscribe**: Subscribes a client into a topic
- **Unsubscribe**: Unsubscribes a client from a topic

The methods are distributed between three interfaces, as three Python programs:

- **Server**: Processes the messages and handles the persistence of the data
- **Publisher**: Sends to the server the messages to be published in a topic
- **Subscriber**: Reads from the server the messages from a topic

1 Server

The Server is the application responsible for processing the messages from subscriber's and publisher's and saving it's contents for persistence while guaranteeing the exactly-once delivery.

No other processing is done by the clients other then parsing the messages from the user input and making sure that the messages are delivered even in cases of server crash.

1.1 Communication

When invoked the server may receive two ports where it will listen for messages from subscribers and publishers. By default port 5556 and 5557 will be used.

The Server uses the zmq Poller class to register two sockets, one for the subscribers and one for the publishers, these sockets are both of type zmq.ROUTER, which allows for an asynchronous implementation of request reply pattern, thus allowing us more flexibility when it comes to communicating to and from the server.

In addition to polling, asyncio is used whenever we write or read from a file. When a message is received it's going to be called for processing using these asynchronous methods in order to keep messages flowing.

At any given time the server may be offline, when that happens subscribers and publishers will continue normal execution, meaning that they can keep sending messages through the sockets and messages will be sent whenever the server turns back on.

1.2 Message Processing

The server has different processing methods for publishers and subscribers. Both methods are called two times per message received, due to the nature of ROUTER-DEALER type sockets, which send a multi part message. The server first receives a message containing the ID of the publisher/subscriber its communicating with, and only then, receives the contents of said message.

For the publisher the processing is easier since we only need to check if there are subscribers subscribed to the topic of the message received, and if there are, we save the message on the **message.txt** file.

Processing the subscriber messages is a bit more complicated since we have to understand the command the subscriber is giving us, the first part is still the same, receiving the id of a subscriber, and then the message (in this case, the command). We wanted to implement a switch case statement but the python version we are using (python 3.9) doesn't support it. We were then forced to do our own implementation of a switch case statement, and ended up creating a class of our own. We named our class Switcher and created 4 methods inside it, one of the methods is shown in the code below.

```
def execute_command(self, command, sub_id, topic = None):
        method_name = str(command)

        # print(method_name)

        # create the method object and create a lambda function for when invalid
            commands are sent to the server
        method = getattr(self, method_name, lambda sub_id, topic: print("Invalid
            command"))
        return method(sub_id, topic)
```

Using `getattr` we create a function with the name of the command passed by the subscriber and proceed with message processing accordingly.

1.3 Server Methods

All of the following methods send a confirmation message back to subscribers/publishers.

1.3.1 Put

For publisher messages there are 2 main methods, process_pub_message and save_message. These functions are called after we receive a message from a publisher, they have the task of saving the contents to a file. Messages are saved by order they were received, if and only if there

are subscribers subscribed to that topic (this is due to the fact that there is no point in keeping messages no one is listening to. Each message has a topic associated and is saved in the format message:topic. In detail, process_pub_message is called two times, this is, like it's explained above, due to the fact that each message received is a multi part. On the first call the server sets the global pub_id variable and on the second call uses that pub_id to store the message contents.

For subscriber messages there is another main method, process_sub_message. The method, process_sub_message works in the same way as process_pub_message, meaning that it is called two times. On the second call of this method the class Switcher is used to decide what to do to the message received by the subscriber.

1.3.2 Get

Get is a method from the class switcher and is called when the the subscriber sends a get message, this is all handled by process_sub_message. The server stores the information of subscribers in a json file, which then it uses to load a dictionary, thus allowing the server to know all the information of a subscriber by simply finding the value of a key. This key is the subscriber id.

The value returned by the dictionary is then another dictionary which has key, value pairs corresponding to topic, and the line of the last message read of that topic. To clarify we will take a look at a concrete example; subscriber with id 1234 is subscribed to the topics, topic1 and topic2, in this case the dictionary will look like this: "1234": "topic1": 34, "topic2": 31. This means, the last message read from topic1 was on line 34, and last message read from topic2 was on line 31.

Get method supports overloading, in which case we can pass the topic we want to get a message from returning the first unread message from that topic. On a normal call of the method get, the server iterates through the lines of messages stored and returns the first that matches a topic the subscriber subscribes.

1.3.3 Switcher.subscribe

Subscribe is a method that belongs to the class Switcher, it loads the json file which stores information about subscribers and creates a new key, value pair associated to that subscriber. It then dumps the dictionary back in json format.

1.3.4 Switcher.unsubscribe

Unsubscribe, like the previous method belongs to the class Switcher, loads the json file all the same as subscribe, but instead of creating a new key, value pair, it simply removes it from the dictionary using del.

1.4 Persistence

Persistence of information is kept through the storing of all information on text files. This could be easily changed to a database. We achieve this persistence by saving all* messages in text files, and also all the information about subscribers. The server doesn't store anything

important in volatile variables which allows the server to crash and keep its state previous to said crash.

2 Publisher

The Publisher is the client responsible for sending put messages to the server in order to write a message to a topic.

Each publisher has an id, which is important to make sure that the messages are sent even if the server and/or the publisher crash during a put operation (see Publisher and Server Crash).

2.1 Put

Whenever a publisher wants to send a new message to a certain topic it will send a request to the server in the format `message:topic`. But before this, the publisher will save locally the message and the topic in a file to persist a crash from itself and the server. When the message is successfully sent and a response received, the message is removed from this local file.

2.2 Receiving Messages

Receiving messages from the server is done so in a separated thread keeping our main loop running, this allows us to send messages to the server continuously and also receive them at the same time.

2.3 Publisher and Server Crash

If both the publisher and the server crash or exit before the message is sent, the content and topic of the message will stay saved in a local file in the publisher. When the publisher is back online, it will read this file and send all previous unsent messages (unsent messages prior to the crash).

.

3 Subscriber

The Subscriber is the client that can read from a topic that it's subscribed to, through the get function, and subscribe to a topic. If a client subscribes to a topic that doesn't exist this topic will be created.

The Subscriber has an id that is used to know what topics it's subscribed to, and so to know what messages each client has already read in order to guarantee the exactly-once delivery.

3.1 Get and Subscribe

The client has two threads to manage the sending and receiving of messages. The processing of messages is done on the server so that the subscriber only parses the user input in order to send the message correctly to the server.

3.2 Receiving Messages

In a similar way to the publisher, receiving messages is done so via thread.

3.3 Subscriber and Server Crash

In order to the code to be resilient to crashes messages from the server while a message was being sent, the subscriber writes all the messages it sends into a file.

If it ever happens that the messages were not sent correctly to the server and the client is shot down, when it boots up again it will resend them.

Every time the client receives a response from the server, it knows that the messages were received, so it erases the content from this referred file so not to send the messages more then once, to satisfy the exactly-once delivery rule.

Since multiple threads are used to receive and send messages, and both access the same file a lock is used for there not to be concurrent access problems.

4 Final thoughts

We are confident enough that our work will guarantee what was asked in the assignment even in the most extreme cases, we have carefully analysed a lot of possible fail scenarios and we expect our implementation to stand.