Department of Informatic

Large Scale Distributed Systems

Reliable Pub/Sub Service

Author:
Tiago Antunes
Tomás Fidalgo
Vasco Alves

October 2022

Table of Contents

1 Introduction

The goal of the project was to develop a reliable publish-subscribe service. This service should allow subscribers to create topics, subscribe or unsubscribe them and get their messages. The publishers are responsible for putting messages on a topic. The service should also guarantee exactly-once delivery (except under rare circumstances), which means that a message placed in a topic, after the subscription of that topic by a subscriber, should be received once and only once by that subscriber.

2 Implementation Aspects

2.1 Design

After reading the ZMQ Guide, we decided to use the socket combination REQ and ROUTER.
As we can see in the figure below, we have multiple clients (subscribers and publishers), each one with a REQ socket, and one server with a ROUTER socket to handle the requests from the clients. We implemented this design using Python and the pyzmq library. The server is responsible for keeping information about the existing topics, such as their subscribers and messages, and also the next message each subscriber is supposed to receive. We keep this information on non-volatile memory so that we can recover the information after a crash.

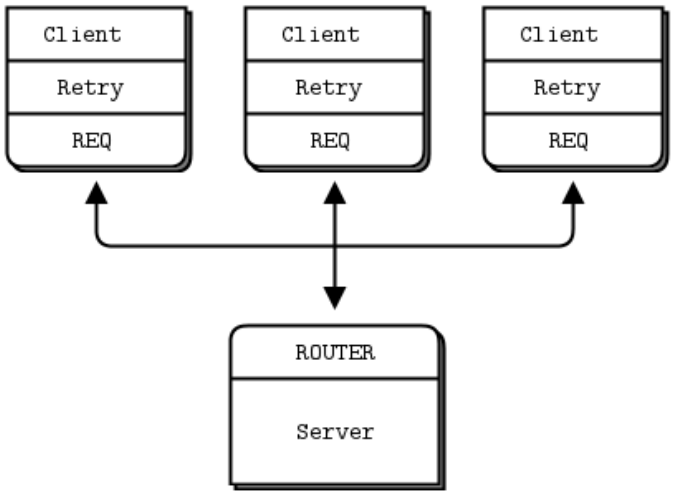

Figure 1: Design of our project

2.2 Data Structures

Regarding data structures, we created four different classes to create and store the needed information: Topic, Server, Subscriber, and Publisher.

2.2.1 Topic

Each Topic has a name (works like an identifier, unique), a list of its subscribers, a list of its messages, and a dictionary that matches every subscriber to the next message they are supposed to receive (message index). When a new Topic is created, we create a new directory with two text files - subscribers.txt and messages.txt - to save the topic's information in non-volatile memory.

```python
# Class Topic
# Represents a topic created when a subscriber subscribes to a given topic.
class Topic:
    def __init__(self, name, subs=[], messages=[], subs_next_message={}) -> None:
        # Topic ID
        self.name = name

        # Arrays/Dictionary to save information about the topic
        self.subs = subs #list of sub_ids
        self.messages = messages #list of all topic messages
        self.subs_next_message = subs_next_message #dictionary: key-sub_id value-next message index
        for key in self.subs:
            self.subs_next_message[key] = 0

        # Create directory to save topic in persistent memory
        if not os.path.isdir('topics/' + self.name):
            os.mkdir('topics/' + self.name)

        subs_file = open('topics/' + self.name + "/subscribers.txt", "w")
        for i in self.subs:
            subs_file.write(i + ' ' + str(self.subs_next_message[i]) + '\n')
        subs_file.close()

        msgs_file = open('topics/' + self.name + "/messages.txt", "w")
        for i in self.messages:
            msgs_file.write(i+"\n")
        msgs_file.close()

        print('Topic ' + name + ' created successfully!')
```

Figure 2: Topic class

2.2.2 Server

Next, we have the Server Class. Every time the server starts, we create a new socket of type ROUTER and a poller to handle the messages given by the clients. We also create a dictionary that matches the topic identifier as key and the Topic object as value. We then create a list with the keys from the dictionary, which are the topics' identifiers so it's easier to verify if a topic has already been created.

```python
# Class Server
# Receives messages from Subscribers and Publishers
class Server:
    def __init__(self) -> None:
        # Create ROUTER socket
        context = zmq.Context()

        self.router = context.socket(zmq.ROUTER)
        self.router.bind("tcp://*:5555")

        # Create Poller to handle the messages
        self.poller = zmq.Poller()
        self.poller.register(self.router, zmq.POLLIN)

        # Create a directory to save topics
        self.topics = {} # Key --> topic name; Value --> topic object
        self.topics_key_view = self.topics.keys() # It will help check if a topic exists
```

Figure 3: Server class

2.2.3 Publisher

The Publisher class has a socket of REQ type associated to make requests to the server. It allows the client to publish messages on specific topics.

```python
# Class Publisher (REQ)
# Creates and publishes messages about a topic given
class Publisher:
    def __init__(self) -> None:
        # Create Publisher Socket
        context = zmq.Context()
        self.proxy_socket = context.socket(zmq.REQ)
        self.proxy_socket.connect('tcp://localhost:5555')
```

Figure 4: Publisher class

2.2.4 Subscriber

Each subscriber has a unique identifier and a socket of type REQ to communicate with the server. The Subscriber client can perform three operations, subscribe to a topic, unsubscribe from a topic, and get a message from a topic to which it is subscribed.

```python
# Class Subscriber (REQ)
# Subscribe and unsubscribe topics
# Get messages from topics that is subscribes to
class Subscriber:
    def __init__(self, id) -> None:
        self.id = id

        # Create a Subscriber socket
        context = zmq.Context()
        self.proxy_socket = context.socket(zmq.REQ)
        self.proxy_socket.connect('tcp://localhost:5555')
```

Figure 5: Subscriber class

2.3 Message Format

On the server side, we have a function called `parse_msg` that receives two arguments: the server itself and the message in an array of bytes. First, we decode the message in `'utf-8'` format and then split that message into three parts (depending on the message the server receives).
The first part of the message says what is the operation:

- `sub`: subscribe topic
- `put`: put a message in topic
- `get`: get message from topic
- `unsub`: unsubscribe topic

The second part is the name of the topic that the server will operate on.

The third and last part of the message will depend on the operation. If the operation is:

- `sub`, `get` or `unsub` : subscriber id
- `put`: content of the message

2.4 Algorithms and Implementation

Inside the Topic class, we have four main functions:

- `add_sub`: This function allows a client to become a topic subscriber and adds his id to the topic's respective subscriber list. First, we check if the subscriber had already subscribed to the topic. If not, we add the subscriber id to the list of subscribers of the topic and since we haven't done yet a get message request, the next message index (subs-next-message[sub-id]) of that subscriber is equal to zero. Also, we need to change the subscriber.txt by adding a new line with the subscriber id and the next message
- `add_message`: Called when a put command is executed, this function adds the respective message to our topic and writes it on its messages text file.
- `remove_sub`: Called when there is an unsubscribe command, this function removes a subscriber from the topics subscriber list. Firstly, it verifies if he is indeed a subscriber and if he really is, the subscriber id is removed from the subscriber list and the respective line from the subscriber's file.
- `get_msg`: This function accepts as an argument the subscriber id and returns the next message from the topic requested by the client. It first verifies if the client is a subscriber and then gets the message index associated with his id. After this, the next message index of the subscriber is incremented, so the next time the client will get the right message. Finally, the message is retrieved, after verifying if the index doesn't exceed the number of messages and then garbage collection is executed. The fact that the next message index is increment after the get operation allows us to make sure a message will only be received once.

We also implement garbage collection to delete messages that have already been received by all the subscribers of a topic at a given moment. Since we have the index of the next message each topic is supposed to receive, we can obtain the lowest index, and delete the messages which have an index lower than that one from the messages list and from the messages.txt file. We then update the indexes of the messages each subscriber has to receive to match the alterations on the messages list.

2.5 Handling Errors

In order to handle server fails and their outcomes, we implemented a client-sided function (client-process-msg in zmq-utils) that processes requests from both subscribers and publishers and the respective responses from the server.

This function consists of an application of the Lazy Pirate Pattern, found in the ZMQ guide, and polls the client socket for a server response until it gets it or the timeout (1 second) is over. In the second case, it retries the process and reconnects to the server, until the number of tries possible is over (5 tries)
The main functions and code parts that most commonly produced errors, like the function in charge of polling, were also involved in a try/except block so we could catch and process the errors accordingly.

```python
def client_process_msg(context, client , request_msg):
    retries_left = nr_tries
    request = str(request_msg).encode('utf-8')
    print("Sending...")

    try:
        client.send(request)

        while retries_left != 0:
            if (client.poll(timeout) & zmq.POLLIN) != 0:
                reply = client.recv()
                return reply.decode('utf-8')

            retries_left -= 1
            print("No response from server")
            # Socket is confused. Close and remove it.
            client.setsockopt(zmq.LINGER, 0)
            client.close()
            if retries_left == 0:
                print("Server seems to be offline, abandoning")
                return -1

            print("Reconnecting to server…")
            # Create new connection
            client = context.socket(zmq.REQ)
            client.connect(endpoint)
            print("Resending (%s)", request)
            client.send(request)
    except zmq.ZMQError:
        print('ZMQ error! Aborting...')
        sys.exit()
```

Figure 6: Client-process-msg function

2.6 Rare Circumstances

Relatively to the rare circumstances, we tested a situation in which the server goes down immediately after receiving a get request. As the server is unable to reply, the client tries to send the request again five times. What may happen is that the server processes one of these requests before going down while being unable to respond, and in this case, the client will not get the right message (or even any) next time because the previous get request was already supposedly processed by the server. This problem also happened in a similar way with the other commands, leading to future failures, because as the server goes down, the client doesn't acknowledge that his previous command was processed. It also happened that the server, before going down, processed more than one request while being unable to reply, leading to the repetition of the described problem.

3 Trade-offs

In the case of the server going down, we have implemented on the client side a verification of a maximum of 5 retries to send a request. This allows the client to know whether the server is up or down but can result in some requests being processed more than once.

Since we keep the topic's information in non-volatile memory, it can take longer to restart the server after the crash, since it has to read the files to recover the state. However, we consider this a positive trade-off because keeping the server's state is an important thing to guarantee the durability of subscriptions and the delivery of messages.

References

[1] ZMQ, 2022, ZØMQ - The Guide.

[2] PyZMQ, 2022, PyZMQ Documentation