U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Large Scale Distributed Systems

Implementation of a Reliable Pub/Sub Service

Alberto José Ribeiro da Cunha
up201906325@up.pt

João Bernardo Castelão Dias Assunção e Costa
up201907355@up.pt

Diogo Filipe Ventura Martins
up201806280@up.pt

Marina Tostões Fernandes Leitão Dias
up201806787@up.pt

October 2022

Index

1 Introduction

The following work, that consists of an implementation of a Reliable Publish/Subscribe Service, was done for the subject of Large Scale Distributed Systems.

The project is supported by topics, each identified by an arbitrary string, that include messages that are associated to them and that can be subscribed. This way, subscribers get the messages of topics they subscribe to.

The work also relies on the operations *get* (to consume a message from a topic), *put* (to publish a message on a topic), *subscribe* (to subscribe a topic) and *unsubscribe* (to unsubscribe a topic).

This service was developed by our team, aiming to guarantee durable subscriptions and "exactly-once"delivery. Therefore, we believe that the implementation of the project will lead us to explore more the concepts taught in the subject and consolidate them.

2 API

In order to make our project come to life, we decided to establish that it would be comprised of an API that would use a Proxy to control the whole system in the back-end. This API allowed other processes to interact with our system, by allowing them to create topics, publish data on them and retrieve data published by other processes. The proxy was essential to control the flow of information within our system and to store the data and metadata of the topics. The whole API was built on the principle that one should be able to use all functionalities of the API by just using 4 main commands (*subscribe() unsubscribe() push() and get()*) and that it should be reliable, meaning that at least the user should be notified if data was lost while being transmited.

3 Design

The API implements three main classes:

- **Proxy** - It is the central piece of the API and is used to create the process that is responsible for handling the whole messaging system we created and is necessary for the system to work.
- **Subscriber** - It is used retrieve data from topics. It has three main methods: *get(topic)* to retrieve a topic update; *subscribe(topic)* to subscribe a user to a given topic; and *unsubscribe(topic)* to unsubscribe the user from a given topic.
- **Publisher** - It is used to put data into topics. It has only one main method, *put(message)*, which puts a message in a topic.

The API also includes two auxiliary classes, Message and Topic, in order to store the metadata of this elements in our system.

Further implementation details are described in section 4.

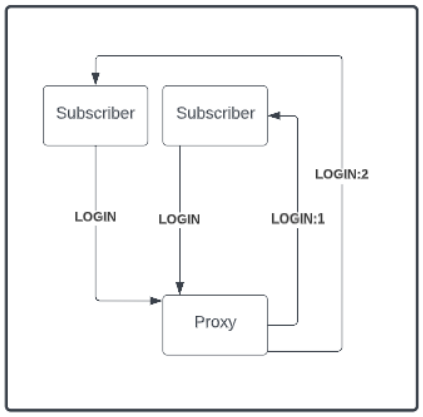
The following image describes the flow of information within our system:

4 Implementation Details

4.1 Subscriber

Subscriber Creation

When the subscriber object is creates a socket and it tries to connect to the Proxy [1]. If it is successfully connects it now needs and ID. We decided not to allow the user to choose this ID, but have the server decide one for them. This is done by sending a message *LOGIN* to the Proxy and it replying with *LOGIN:user_id*, in which the user id is an integer that is unique for each subscriber.



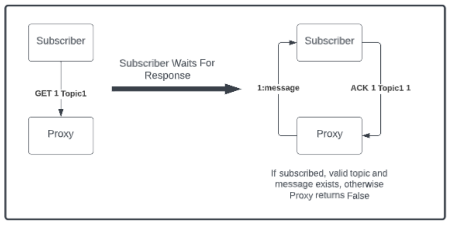
Subscriber Creation

get(topic) method

This method is used when the subscriber wants to retrieve data from a topic. It first sends the message *GET subscriber_id topic* to the Proxy and waits for the response. If the response fails to arrive, is *NOT_SUBSCRIBED* (meaning that the subscriber is not subscribed to the topic), is *INVALID_TOPIC* (meaning the topic does not exists and the name given is not valid) or is *NO_MESSAGES* (meaning there are no messages to be

[1]the IP and port default to *127.0.0.1:5556* but they can be set manually

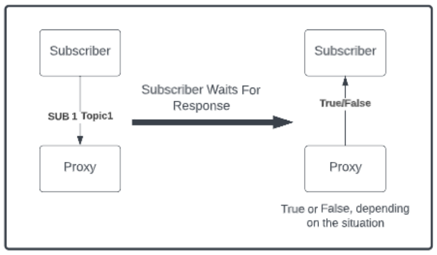
returned) it returns *False*. If it receives a message in the format *message_id:message* then it knows it is valid and will respond with an *ACK user_id topic message_id*. It also updates the last message received on that topic as explained in 4.6.


Subscriber
GET 1 Topic1
Proxy
Subscriber Waits For Response
Subscriber
1:message
ACK 1 Topic1 1
Proxy
If subscribed, valid topic and message exists, otherwise Proxy returns False

get(topic) method

subscribe(topic) method

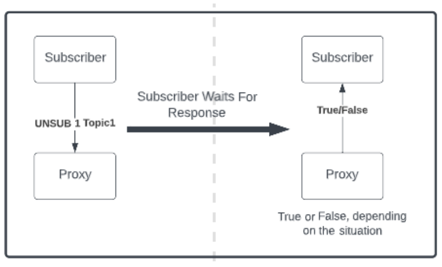
This method is used when the subscriber wants to subscribe a topic. It sends *SUB user_id topic* to the Proxy and waits for a reply. If none arrives it returns *False*, otherwise it returns *True*.


Subscriber
SUB 1 Topic1
Proxy
Subscriber Waits For Response
Subscriber
True/False
Proxy
True or False, depending on the situation

subscribe(topic) method

unsubscribe(topic) method

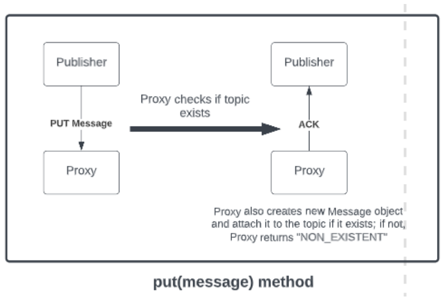
This method is used when the subscriber wants to unsubscribe from a topic. It sends *UNSUB user_id topic* to the Proxy and waits for a reply. If none arrives or it is *NON_EXISTENT* (meaning the topic does not even exist) it returns *False*, otherwise it returns *True*.


Subscriber
UNSUB 1 Topic1
Proxy
Subscriber Waits For Response
Subscriber
True/False
Proxy
True or False, depending on the situation

unsubscribe(topic) method

4.2 Publisher

The Publisher class is used to associate messages with the desired topic. It has only one main method: *put(message)*. This method puts a message in a topic.


Publisher
PUT Message
Proxy
Proxy checks if topic exists
Publisher
ACK
Proxy
Proxy also creates new Message object and attach it to the topic if it exists; if not, Proxy returns "NON_EXISTENT"

put(message) method

4.3 Proxy

When it's created, the Proxy first generates two REQ/REP sockets, a subscribers socket and a publishers socket [2]. Then it also uses a ZMQ Poller to then iterate through incoming requests.

To start the proxy, one simply needs to invoke the method *run* which starts an infinite loop. In this loop, first we use the Poller to check for incoming requests and, if they exist, the Proxy handles them depending of their content. The types of messages the Proxy can receive and the ways to handle them are:

- **SUB subscriber_id topic**
  This is the case in which a subscriber wants to subscribe a topic. Firstly the Proxy checks if the topic given already exists. If not it tries to create a new topic. Due to the format of the messages exchanged by our API, white spaces are not allowed in the topic name, so the Proxy automatically switches the white spaces to underscores. If for example someone tries to create a topic "my data", the Proxy will create the topic

[2]The ports default to 5556 and 5555 respectively but can be changed manually

"my_data". Then the proxy will append this subscriber to the topic's subscriber list and finally respond with and ACK_SUB, acknowledging the subscription.

- **UNSUB subscriber_id topic**
  This is the case in which a subscriber wants to unsubscribe a topic. Firstly the Proxy will check if the topic exists. If it does, it will remove the subscriber from the topic's subscriber list and respond with SUCCESS to acknowledge the unsubscription. If the topic does not exist, the Proxy will reply with NON_EXISTENT to alert the subscriber that the topic does not exist.

- **LOGIN**
  This is the case in which a subscriber wants to log in the system The Proxy will check the id of the last subscriber to log in the system and it will attribute the next id to the new subscriber. For instance, if the last user's id was 45, it will attribute 46. It will reply with LOGIN:new_user_id.

- **PUT topic data**
  This is the case in which a publisher wants to put data into a topic. The Proxy will firstly check if the topic exists, if not it will reply with "NON_EXISTENT". If the topic does exist, it will create a new Message object and attach it to the wanted topic and finally respond wiht and ACK to confirm the reception of the data.

- **ACK subscriber_id topic message_id**
  This is the case in which a subscriber is confirming the reception of a message. The Proxy first checks if the topic exists and if it does it looks for the message in the topic and removes the subscriber from the list of subscribers who still haven't received this message (by invoking the method *message.removeSubscriberToDeliver(suscriber_id)*).

4.4 Message

The Message class is responsible for the messages in the context of the project. It has one main method: *removeSubscriberToDeliver(subscriberName)*, that removes a subscriber identified by *subscriberName*, when a message is delivered. This class is composed by the text of the message (*text*), by topic it belongs to (*topic*), by the list *subscribersLeft* that represents the subscribers that haven't received the message yet, and by *messageID*, that identifies it, matching acknowledgements from clients.

4.5 Topic

The Topic class is responsible for the topics. It has four main methods: *subscribe(userID)*, that subscribes the user identified by *userID* to the topic; *unsubscribe(userID)*, that unsubscribes the user identified by *userID* from the topic; *messageCheck(messageID)* that checks if a message identified by *messageID* is one of the messages of the topic; and *messageIndex(messageID)*, that returns the index of the message identified by *messageID* inside of the list *messages*, that represents all of the messages of the topic. This class is composed by *name*, *subscribers* (a list of the subscribers of the topic), *messages* and *lastMessageNr* (that represents how many messages the topic has).

4.6 Fault Tolerances

Ensuring reliability on the subscriber side

In order to make the subscriber reliable, we had to keep track of which messages where received and which weren't, therefore we created a property for the subscriber called lastReceivedMessageID, which is a dictionary containing pairs of topic names and the last message ID received from that topic. This allows the subscriber to allways know if it is receiving a valid or replicated message. How is this achieved? If the Proxy receives a GET from a subscriber it will reply with the next message following the last acknowledged message. If for any reason the message does not reach the subscriber, when it asks for the message again, the Proxy will retransmit the same message, not move on to the next one. If the user receives the message but the acknowledgment is lost on the way to the Proxy, the next time the subscriber asks for a message it will receive the same message (because, since the Proxy does not know the subscriber received the message, it will retransmit) and will discard it, sending an ACK back to the Proxy to tell it to move on to the next message in the next get.

5 Conclusion

The implementation fulfilled the requirements of the project and was tested against various cases. During the development of this project, some challenges were found, which led us to investigate which architectures were more adequate for our work and how to implement them.

This way, we think it was an efficient way to help us understand more the subject of Large Scale Distributed Systems. It also allowed us to get more knowledge regarding the networking library ZeroMQ and consequentially the connection of sockets.