U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Large Scale Distriuted Systems

Project 1 - Publish-Subscribe Service

T6G15

João Rodrigo	*up201705110*
Mª Francisca Almeida	*up201806398*
Bernardo Ferreira	*up201806581*
Telmo Botelho	*up201806821*

Indíce

Overview

The goal of this project was to develop a reliable publish-subscribe service. To create a service, ZMQ libraries were used.
The basic architecture of the service is descibred below:

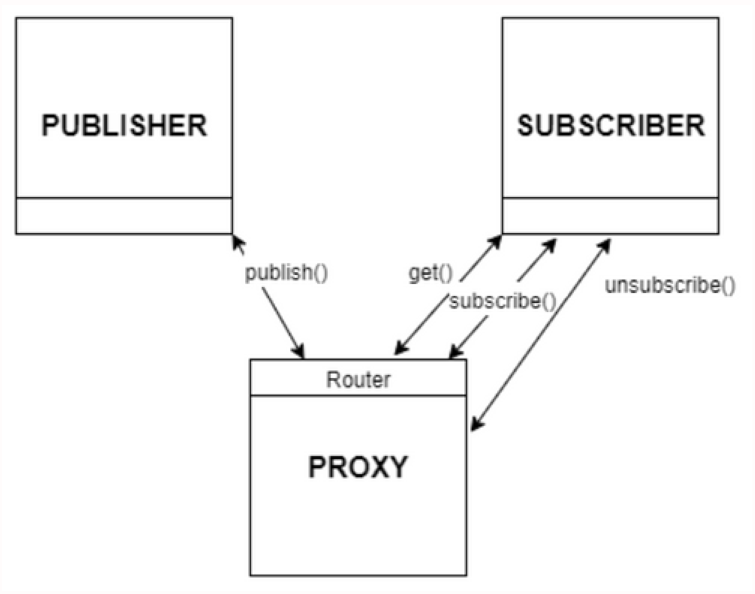
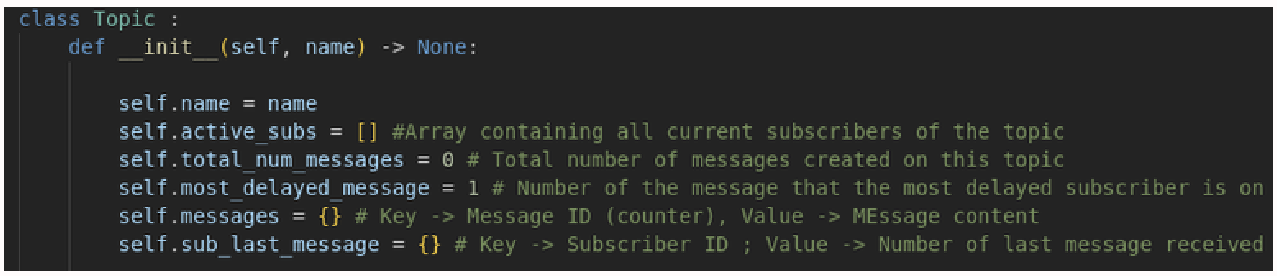


Figure 1 : Architecture of the service

As shown in in figure 1, this service implements 3 classes:

- The Proxy is responsible for managing the messages published by a Publisher to a topic and sending those messages to the appropriate Subscribers. The topics are maintained by the Proxy.
- The Subscriber can subscribe or unsubscribe to a Topic. If the topic doesn't exist, then the proxy creates it. The Subscriber will then get all the messages posted to that topic as long as it requests messages from the proxy.
- The Publisher can post a message to a topic. If the topic doesn't exist the messages are discarded.

Proxy works as a middle man between the many subscribers and publishers. It keeps updating status on all Topics and all subscribers each time there is a put(), get(), sub or unsubscribe() so that all components of the system are in sync. The proxy keeps all it's knowledge of the system using a dictionary in which the key is the topic's name and the value the Topic Object.



```
class Topic :
    def __init__(self, name) -> None:

        self.name = name
        self.active_subs = [] #Array containing all current subscribers of the topic
        self.total_num_messages = 0 # Total number of messages created on this topic
        self.most_delayed_message = 1 # Number of the message that the most delayed subscriber is on
        self.messages = {} # Key -> Message ID (counter), Value -> MEssage content
        self.sub_last_message = {} # Key -> Subscriber ID ; Value -> Number of last message received
```

Image 1 - Topic class

Operations

Unsusbcribe Operation

This operation is only available for Subscribers and is implemented in the class Subscriber.
In this operation a subscriber ceases to receive messages from the publishers. He sends a message to the proxy to unsubscribe from the topic.
In order to unsubscribe a topic, the subscriber needs to be subscribed to the that topic. If not subscribed a message is shown saying that the subscriber is not subscribed to the topic. Also, the topic needs to exist for the operation happen without any problems.
Plus, on the proxy side, when an unsubscription is done sucessfully, it checks if there is any other subscriber to that topic, in case there ins't all mesasges from that Topic are erased (since they woulnd't be use for now)

Susbcribe Operation

This operation is only available for Subscribers and is implemented in the class Subscriber.
In this operation a subscriber joins a topic and he starts to recieve messages available in that topic (as long as he sends GET message to proxy).
To join a topic the subscriber sends a SUBSCRIBE message to the proxy.
In order to subscribe, the subscriber can't be already subscribed to that topic. If the topic doesn't exist a new topic is created.

Put Operation

This operation is only available for Publishers and is implemented in the class Publisher.
In order to publish a message to a topic, the publisher sends a message to the proxy and adds a new message to the topic unless the topic isn't created, in that case, it does not save the message and the Proxy answer the Publisher with an error message.

Get Operation

This operation is only available for Subscribers and is implemented in the class Subscriber.
In order to get a message, the subscriber sends a message to the proxy, asking for new messages availabale in the subscribed topic.
To be able to perform this operation, the topic needs to exist and the subscriber needs to to be subscribed to that topic.

Fault-Tolerance

The service presented in the previous chapter guarantees that messages are delivered "exactly-once" when no failure occurs. In order to guarantee "exactly-once" delivery, in the presence of communication failures or process crashes, it has been implemented a way to deal with said crashes and communication failures.
In order not to lose messages, we created a Dictionary in the class Topic to save said messages. Even if the Subscriber fails, when it starts to work again, he will receive the message. The message is saved until all the Subscribers received it, after that the system deletes it, or if a topic with saved messages is left with no subscriber, than all messages belonging to the topic are deleted.
Another way to ensure messages are not lost, is through a JSON file. On proxy start up it reads the JSON file containing the Topic dictionary. On every proxy "cycle" the JSON file is updated with new values.
.

Trade-offs

Using JSON isn't the most optimize way to prevent not being able to recover system information since it has to rewrite the file everytime. Another alternative would be with a SQL database because that way we could have more singular operations.

Conclusions

As it is shown in this report, the service implemented can successfully guarantee "exactly-once" delivery, in the presence of communication failures or process crashes, except in "rare circumstances".
All the operations were implemented. A Subscriber can consume messages of a Topic (get()), subscribe and unsubscribe to Topics. The Publisher can publish messages to a Topic (put()).
The Proxy class guarantees that all subscriptions are durable and messages on Topics are only removed when they are not needed anymore.
With all this, it can be concluded that the service created implements all the objectives for this project.