U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Reliable Pub/Sub Service

SDLE 21/22

MEIC 1º ANO

TURMA 2 - GRUPO 15

José Rocha, up201806371
Miguel Silva, up201806388
Pedro Ferreira, up201806506
Pedro Ponte, up201809694

Índice

1 Introduction

This project was developed within the scope of the course of **Sistemas Distribuídos de Larga Escala**. Its main objective was to design and implement a ***reliable*** *Publish - Subscribe* service, which ensures ***exactly-once*** delivery of each message to subscribers subscribed to a series of topics.

Furthermore, this service offers two main operations:

- **put()**, which allows a *Publisher* to publish a message to a certain topic;
- **get()**, which allows a *Subscriber* to consume a message from a certain topic.

To receive messages, each *Subscriber* must first subscribe to a topic (by using the **subscribe()** call). If he then wishes to stop receiving messages concerning this topic, he can choose to unsubscribe (by using the **unsubscribe()** call).

On the whole, this report aims to explain our implementation of this service in a more detailed and clear way.

2 Architecture and Design

Our project was developed using Java and the ZeroMQ library.

To implement this service and its protocols, we created four different classes: *Subscriber*, *Publisher*, *Topic* and *Proxy*.

The ***Subscriber*** class is responsible for the *subscribe*, *unsubscribe* and *get* protocols. The ***Publisher*** class is responsible for the *put* protocol implementation. The ***Topic*** class stores information concerning messages sent to the service for each topic, as stated on Listing 1 (Attachments). Finally, the ***Proxy*** class, the heart of our system, handles the messages sent both by the *Subscribers* and the *Publishers*, whilst updating the messages in the class *Topic*.

To allow the *Subscribers* and *Publishers* to communicate with the *Proxy* through the usage of a variety of ZMQ Sockets.

2.1 Subscriber - Proxy

In order to provide communication between *Subscriber* and *Proxy*, we have created some sockets, which help implement different patterns.

The **PUB/SUB** architecture is achieved through the usage of SUB sockets in *Subscribers* and a PUB socket in the *Proxy*. This socket allows the

subscribers to subscribe/unsubscribe topics, which will be explained further ahead in this report.

As the PUB/SUB pattern doesn't allow *Subscribers* to send messages to the *Proxy* and the *Proxy* to receive messages from the *Subscriber*, there was a need to implement a **REQ/REP** architecture as well. These REQ/REP sockets will prove useful in both the *get* protocol and other messages which will also be explained further ahead in this report.

Finally, we have also created **PUSH/PULL** sockets in order to allow the *Subscriber* to confirm the reception of a message in the *get* protocol and guarantee the *exactly-once* delivery.

2.2 Publisher - Proxy

Similar to what was previously described, communication between *Publisher* and *Proxy* was achieved through the creation of different types of sockets, which helped implement different patterns.

In order to allow a *Publisher* to publish new messages to a certain topic (and the Proxy to receive these messages), we used a **PUB/SUB** architecture, but this time the *Proxy* acts as SUB and *Publisher* as PUB.

In this scenario, we also have the same problem described previously - so **REQ/REP** sockets were, once more, needed to allow the *Publisher* to know whether their message was published or not.

3 Reliability and fault tolerance

When it came to reliability and fault tolerance, our main concern was ensuring the system was capable of dealing with all kinds of problems and crashes throughout its usage. Bearing this in mind, we started by focusing our attention on the ***Proxy***. Taking into account that it represents a Single Point of Failure, it was of the utmost importance that, in case of failure, all the messages and topics stored up until that point were preserved. Losing this sort of information would have catastrophic repercussions for the whole system, making the network extremely unstable. This problem was solved through the usage of **Class Serialization**. The purpose of this solution is to periodically store all the data locally, so that, in case of failure, the system has a copy of all the stored data on which it can rely. As mentioned previously, this data is stored on the ***Storage*** class, which is the one being serialized periodically, as stated in Listing 2:

Furthermore, we used a **ScheduledThreadPoolExecutor** to solve this problem - a separate *Thread* is created, which is responsible for serializing

the *Proxy*'s storage every 15 seconds. Moreover, the serialized data is stored in a separate folder, as can be deducted through Listing 2 and 3 (Attachments).

While this method ensured there was no significant loss of data on the Single Point of Failure, we also needed to make sure each protocol worked as planned. In other words, the *Proxy* needed to make sure the messages it sent were delivered. On the other hand, both *Subscribers* and *Publishers* would benefit from the acknowledgment of the *Proxy* concerning their requests. The message exchange for each protocol is listed below:

3.1 Subscribe

Since our architecture is based on the PUB/SUB pattern, this protocol was implemented in a fairly simple way. The *Subscriber* subscribes a topic by using the subscribe() call and the *Proxy* responds by confirming whether the protocol was successful or not. This communication is done through the XPUB/XSUB sockets.

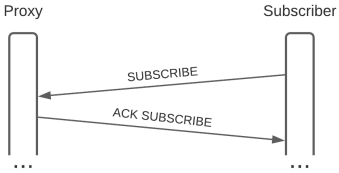

Figure 1. Message exchange during the Subscribe Protocol

3.2 Unsubscribe

The Unsubscribe protocol proved to be more complex than the latter. While the protocol itself can be done by using the unsubscribe() call, ensuring the *Proxy*'s data integrity is preserved involves a series of extra steps. Moreover, if nothing is done, it is impossible to differentiate an intentional Unsubscribe from one related to *Subscriber* crash (when a *Subscriber* crashes, ZEROMQ automatically closes all its sockets, unsubscribing all previously subscribed topics). In other words, if nothing is done, we risk *Subscribers* unsubscribing from several topics and losing messages they might be interested in, without them even realising this fact. In order to be able to tell the difference between these scenarios, the *Proxy* will first acknowledge the incoming Unsubscribe so it can be prepared to deal with it. This first exchange of messages is done through REQ/REP - the *Subscriber* sends a REQ message, and the *Proxy* dictates whether they can proceed (or not),

by replying with a REP message. Once this is done, the *Subscriber* uses the unsubscribe() call, concluding the protocol.

On the whole, the *Proxy* will know an unsubscribe is premeditated if it received a REQ message immediately before.

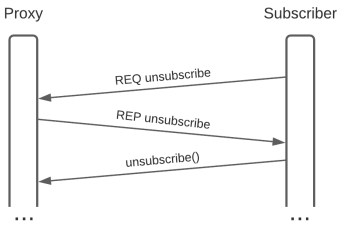

Figure 2. Message exchange during the Unsubscribe Protocol

3.3 Put

Similar to the previous protocol, there is an exchange of 3 messages in total. The *Publisher* starts by sending the message they wish to publish and the *Proxy* receives and deals with it appropriately. Since it is impossible for the *Proxy* to respond through a SUB socket, we used REQ/REP for the acknowledgments. Moreover, the *Publisher* will send a REQ message, to know whether the protocol was successful. The *Proxy* will respond with one of three possible answers:

- **PUT_ACK_SUCC**, stating the success of the protocol;
- **PUT_ACK_DENY**, stating the unsuccess of the protocol, which happens when the *Topic* in question doesn't exist yet.
- **PUT_NACK**, stating that the first message wasn't delivered in the first place.

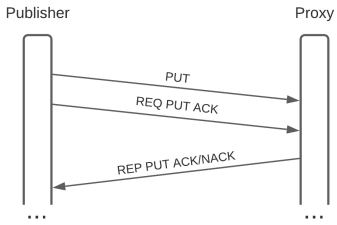

Figure 3. Message exchange during the Put Protocol

3.4 Get

This protocol is entirely done through the usage of REQ/REP and PUSH/PULL. The *Subscriber* requests a message concerning a specific topic,

and the *Proxy* replies with one of four possible answers:

- **GET_NONEW**, which indicates there are no new messages;
- **GET_SUCC**, which indicates there is a new message, which is sent through the reply as well;
- **GET_UNSUB**, which indicates the *Subscriber* isn't yet subscribed to the *Topic*;
- **GET_UNEXIST**, which indicates the *Topic* doesn't exist yet;
- **GET_EMPTY**, which indicates messages haven't yet been posted to the *Topic*;

To guarantee the *exactly-once* delivery of messages, the *Subscriber* will then send a final message to the *Proxy*, stating that the protocol was successful. This is done through the usage of the PUSH/PULL pattern. The *Proxy* pulls the message, acknowledging the protocol's success, and proceeds to make the necessary changes to its data structure. If this last step wasn't taken, there was no way of knowing whether the delivery of the message was successful, which could result in the loss of data.

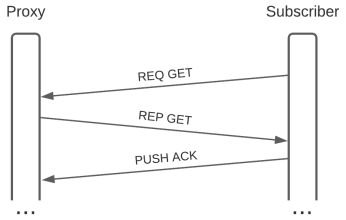

Figure 4. Message exchange during the Get Protocol

Moreover, we have implemented other methods which contribute to the system's robustness and reliability. On the one hand, when a *Subscriber* crashes and automatically unsubscribes due to *ZeroMQ*, we make sure he doesn't lose his subscription privileges once he reconnects. To achieve this, the *Subscriber* starts by asking the Proxy which *Topics* he is subscribed to. The *Proxy* responds with a list of *Topics*, to which the *Subscribers* resubscribes.

On the other hand, we also considered the possibility of having *Subscribers* never consuming messages from *Topics* they're subscribed to. This means certain messages might never be deleted from the system, which in the long run means there can be a huge overload of messages in the *Proxy*. To prevent this, every Subscriber that doesn't consume a message from a

Topic they're subscribed to in a period of 5 minutes will automatically unsubscribe that *Topic*. This is achieved by storing, for each topic, the last instant the Subscriber has consumed a message (or in case of not having done so, subscribed). A separate *Thread* created by the **ScheduledThreadPoolExecutor** is in charge of doing such.

4 Tradeoffs of our implementation

Throughout the project, we concluded our implementation has some key tradeoffs.

The main objective of the system is done more efficiently through the usage of PUB/SUB, whilst maintaining REQ/REP alongside PUSH/PULL for **fault tolerance**. The latter represents the robustness and reliability of the system as a whole. Moreover, the interaction between Publisher/Subscriber and Proxy becomes much more clear and transparent to both parties involved.

As stated previously, the **Class Serialization** prevents data loss from *Proxy* failures, and the automatic unsubscribe for *Subscribers* for not overloading the *Proxy*. Moreover, the main goal for this project - the ***exactly-once*** delivery - is secured due to the exchange of control messages as described above. Additionally, messages sent by Publishers to nonexistent Topics are automatically discarded.

Moreover, we also implemented a **Garbage Collector** mechanism. This mechanism is responsible for cleaning data no longer useful for the Proxy. This includes both deleting topics once no one is no longer subscribed to them and deleting messages which won't be consumed anymore.

However, our project has some liabilities. The usage of *RMI* plays a major role in this since we needed to implement a class responsible for managing it. This class runs indefinitely - without it, the project won't run, since it depends on *RMI*.

5 Conclusion

On the whole, we believe we accomplished all the required goals as documented on the project handout. We also had the opportunity to better understand topics relative to the project, such as the different patterns of message exchange through sockets.

6 Attatchments

```
1   public Topic(String name) {
2       this.name = name;
3       this.messageId = -1;
4       this.subscribers = new ArrayList<>(); // saves all topic subscribers
5       this.messages = new ConcurrentHashMap<>(); // saves all topic
    messages in format <messageId, message>
6       this.subsLastMessage = new ConcurrentHashMap<>(); // for each
    subscriber, saves the id of the last message received in format <subId,
     messageId>
7       this.messagesIds = new ArrayList<>(); // saves all messages Ids
8       this.subsLastMessageIds = new ArrayList<>(); // saves the id of the
    last received message for all subscribers
9   }
```

Listing 1: Topic.java - Structures to manage topic's informations

```
1   public void run() {
2       exec.scheduleAtFixedRate(serialize, 5, 15, TimeUnit.SECONDS);
3       ...
4   }
```

Listing 2: Proxy.java - Creating a *Thread* which serializes the storage

```
1   Runnable serialize = new Runnable() {
2       public void run() {
3           System.out.println("Serializing...");
4           String filename = "proxy/proxy.ser";
5
6           File file = new File(filename);
7           if (!file.exists()) {
8               file.getParentFile().mkdirs();
9               try {
10                  file.createNewFile();
11              } catch (IOException e) {
12                  e.printStackTrace();
13              }
14          }
15          try {
16              FileOutputStream fileStream = new FileOutputStream(filename)
    ;
17              ObjectOutputStream outputStream = new ObjectOutputStream(
    fileStream);
18              outputStream.writeObject(storage);
19              outputStream.close();
20              fileStream.close();
```

```
21              } catch (IOException e ) {
22                  e.printStackTrace();
23              }
24
25          }
26      };
```

Listing 3: Proxy.java - Serializing the Proxy's storage