# Reliable Pub/Sub Service

FEUP - SDLE 2021

First Project - Group t4g15

João Carlos Machado Rocha Pires (up201806079)

João Diogo Martins Romão (up201806779)

Rafael Valente Cristino (up201806680)

Xavier Ruivo Pisco (up201806134)

## Contents

# 1 Introduction

This document describes in detail the architecture of our solution to the reliable publisher / subscriber service proposed as the first practical work for the Large Scale Distributed Systems curricular unit.

According to the project handout, the service should allow two types of users: publishers, which publish messages on a specific topic, and subscribers, which get the messages of the topics to which they are subscribed. The main goal was to make sure the exactly-once delivery requirement was implemented in the most reliable way we could, so that no repetitive messages would be sent to a user that subscribed a topic and that he would not miss any messages, at least in most occasions (in the rare occasions where that was not possible, we opted for an at-least-once requirement).

One of the constraints we had while developing our solution was to build it on top of the *libzmq* (also known as *zeromq*), which is a minimalist message oriented library. That said, the programming language we selected, and that is compatible with this library, is Java.

# 2 Early Stages

In this section of the report, we will describe our initial ideas for this project and quickly discuss both the reason for our idea selection and why we thought the other options we considered were not as good as this one.

## 2.1 Network Design

First, for the design of the network for this project, we came up with the simplest idea we could, which was to connect the subscribers directly to the publishers so that they could communicate with each other directly. Quickly we realised that this option wouldn't scale well, not only because the number of connections needed would be equal to the number of subscribers multiplied by the number of publishers, as well as all the subscribers would need to know the IP addresses and ports of every publisher, and vice-versa.

Secondly, in order to fix the problem faced with this initial solution, we decided to create a proxy that receives and establishes connections from both the subscribers and the publishers. With that, we make sure the number of known IP addresses and ports is limited to two, one for the subscribers to connect to and another for the publishers to connect to.

Without any further problem, we then settled with this design as our final design. The following picture represents an illustration for the structure of our Network Design.

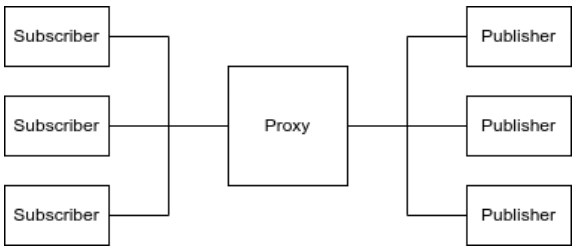


Figure 1: Network Design

## 2.2 Publisher, Subscriber and Proxy - Connections

After deciding the design of our network, we moved into the discussion of what would be the connections we would use for both sides. In the next sub section, we will discuss the connection between the subscribers and the proxy and, in the following one, we will discuss the connections between publishers and the proxy.

## 2.3 Subscriber-Proxy Connections

Starting with the Subscriber-Proxy Connections, the first thing we did was search for the available sockets in *ZeroMQ*. This library supports eight different types of sockets: *REQ, REP,*

*PUB, SUB, ROUTER, DEALER, XPUB and XSUB.*

After studying the different options, we stumbled upon some possible solutions that we found interesting for the problem we had in hands.

First of all, we decided to exclude some of the sockets types, as they didn't fit the right usage for the subscribers. The first we excluded was the *PUB* socket. We wanted the subscribers to receive messages from topics previously subscribed, and not to send messages, thus, the *PUB* socket would not be useful in this case.

After that, we excluded both the *DEALER* and the *ROUTER* sockets. According to the *zguide*, "*DEALER* is like an asynchronous *REQ* socket, and *ROUTER* is like an asynchronous *REP* socket". Since we thought the subscribers didn't need to have asynchronous connections, we chose not to implement these sockets.

On the same chapter of the book, we got to know that "*XSUB* and *XPUB* are exactly like *SUB* and *PUB*, except they expose subscriptions as special messages." Considering this would not add value to our solution, we decided not to have neither *XSUB* nor *XPUB*.

At last, to implement the operations described in the handout, the subscriber should be the one to initiate the "talk" with the proxy, and so, we discarded the *REP* socket, since the proxy would need to be the one to send the first message.

### 2.3.1 Subscriber with *SUB*

After excluding most of the socket types, our first idea was to have the subscribers' sockets as *SUB* sockets from *ZeroMQ*. We came up with this option due to a relation between the names, and so, we debated whether it would be able to do everything we needed for this project or not.

First of all, we found that, to be able to receive connection from a *SUB* socket, the proxy would need to have a *PUB* or a *XPUB* socket.

The first problem we faced with this was the fact that, with this sockets, the subscriber wouldn't have control of when the messages of a topic it has subscribed to would be received. In other words, when the proxy had a message from a subscribed topic available for one subscriber, it would send it immediately to the client. This could be overlooked by saving all the messages sent to the client on memory and only show them when the client sent a get message, not needing to send a message to the proxy on a GET message.

This approach would force us to save the same information multiple times across all the clients and, to prevent great damage on a eventual client shutdown, we would need to store the info of each subscriber in a different file.

Other problem we detected was the wrong keeping of subscriptions while the subscriber was not online. When the subscriber went offline, there would be no way to save the information that should be sent to him and, when it went back online, it would need to resubscribe all the topics it had previously subscribed.

This last problem would make it really hard to implement the needed requirements, and so, we agreed to not implement this type of connection.

### 2.3.2 Subscriber with *REQ*

Last but not least, we debated about the possibility of the subscribers having a *REQ* socket.

The first thing we decided was that, in the proxy, the socket that received the connections from the subscribers would either be a *REP* or a *ROUTER*, as they both work well with request messages. Since the proxy would need to be able to handle multiple connections at the same time, the *ROUTER* socket became our choice since it is asynchronous and, as described in the *zguide*, "The *ROUTER* socket, unlike other sockets, tracks every connection it has."

At first thought, everything seemed to be perfect. The subscriber would initiate every message exchange it needed, and the proxy would reply to it according to the info it had. When the subscriber wanted to subscribe or unsubscribe a topic, it would send a message with on of those purposes and the proxy would save the information like a server. When the subscriber decided to get a message, the proxy would handle all the processing and check if the subscriber had subscribed to the topic and send back the message that the client should receive.

We decided to keep this idea and start the project with this types of connections.

Later on, we thought that, in case of a client crash in the middle of a communication, there would be no reliable way to guarantee exactly-once deliveries for every message.

## 2.4 Publisher-Proxy Connections

On the other side of the network, we also excluded from the publisher some of the sockets available on the *ZeroMQ*.

The first ones we excluded were the *ROUTER*, *DEALER*, *XPUB* and *XSUB* for the exact same reasons we decided not to use them in the subscribers.

We also removed the *REP* option, because that would make the proxy initiate the message exchanges and we wanted them to be initiated by the publisher when it sent a PUT message.

The last one we excluded was the *SUB* socket because we wanted the publishers to send messages to the proxy and not the other way around.

### 2.4.1 Publisher with *PUB*

Since the publishers need to publish their messages to the proxy, we first thought about going with a *PUB* socket.

If the publisher had a *PUB* socket, the proxy could either have a *SUB* or *XSUB* socket. The *SUB* socket is used to receive messages from a topic, and the *XSUB* socket extends the *SUB* one and allows the proxy to send messages to the publishers. On the publisher perspective, we thought it would send every message it wanted to when PUT was called, so there was no need to send messages from the proxy to the publishers, meaning the proxy socket would be a *SUB* socket.

However, we gave it a thought and we found two problems, with one only happening when the proxy closed. In this case, when the proxy came back online it would need to resend subscribe messages for all the topics that one or more subscribers are currently subscribing.

The other problem would happen anytime a publisher would connect to the proxy, with the proxy having the need to send multiple messages to the new publisher, one for each of the topics that at least one subscriber had subscribed to.

Since we thought this problem would be hard to solve, we decided to not implement it this way.

### 2.4.2 Publisher with *REQ*

The other option for the connection between the publishers and the proxy was using a *REQ* socket on the publishers.

For this to be implemented, the proxy would have a *ROUTER* socket like the one explained in the subscribers section.

The only possible problem this implementation might had is similar to the one described in the subscriber with *REQ* section. In case of failure from the proxy or from the publishers, there is no perfect way to guarantee that a topic is successfully stored in the proxy and, thus, it may not be stored in case the proxy does not receive it or it will be stored twice in case the publisher does not receive a confirmation after the proxy successfully stored the message.

Besides those niche problems, we didn't found any drawback or anything that would slow us down on the implementation and, as such, we went forward with it and decided to implement the publishers with a *REQ* socket and the proxy that receives the publishers connections with a *ROUTER* socket.

## 2.5 Final Design

After our initial debriefs about the possible solutions for both the network and the sockets used in it, we ended up with a mirrored implementation, in which both the subscribers and the publishers implemented *REQ* sockets and the proxy two *ROUTER* sockets, one for each of the types of users.

The Figure 2 schema represents this last version.

# 3 Service Design

## 3.1 Proxy Communication Interface

Based on our network design, all of our communications start with a request from either a subscriber or a publisher, being then processed by the proxy, which will store all the needed information and reply to those requests.

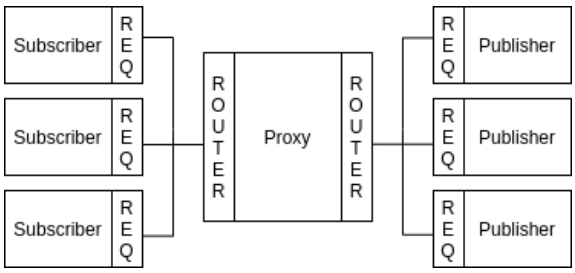


Figure 2: Final Design

In order to handle the communication, we created the following types of messages:

- **GET** - contains the TOPIC and the SUBSCRIBER ID
- **PUT** - contains the TOPIC and the MESSAGE
- **SUBSCRIBE** - contains the TOPIC and the SUBSCRIBER ID
- **UNSUBSCRIBE** - contains the TOPIC and the SUBSCRIBER ID
- **CONTENT** - contains the TOPIC and the MESSAGE CONTENT
- **ACK** - contains a MESSAGE
- **NACK** - contains a MESSAGE
- **RECEIVED** - contains the TOPIC and the SUBSCRIBER ID

## 3.2 Subscribe to a topic

When a subscriber wants to subscribe to a topic, it will send a *SUBSCRIBE* message to the proxy containing the topic it wants to subscribe and its unique identifier.

When the proxy receives that message, it will process it and will output one of the two possible results. Either the subscriber didn't subscribed before to that topic, and, if that's the case, the proxy will save the client information and accept its subscription, replying with an *ACK* message, or it was already a subscribed topic and the proxy will maintain it's subscription, but will reply with a *NACK* message to make clear it was the repetition of an already made request.

In both cases, the subscriber will be subscribed to the topic, thus, no more communication is necessary. In case a byzantine failure occurs, it will be up to the user to re-subscribe to the topic, in order to make sure that he receives later on the messages published for that topic.

## 3.3 Unsubscribe from a topic

To unsubscribe from a topic, the subscriber sends a *UNSUBSCRIBE* message to the proxy with the topic to unsubscribe and the subscriber's id.

The proxy will receive that message, process it, and reply with a *ACK* message in case the user was already a subscriber of the topic or a *NACK* message if the user wasn't a subscriber.

Similar to the behaviour that occurs on the subscription to a topic, despite the responses given to the subscriber, we guarantee the user is not subscribed to the topic anymore. That said, there's no need for further communication. If there is an unexpected problem, like a byzantine failure, the user should unsubscribe again as a precaution, since the *UNSUBSCRIBE* message is idempotent.

## 3.4 Consuming a Message

When a subscriber wants to read a message from a topic already subscribed, it will send a GET message to the proxy containing the topic and the ID of the subscriber.

After receiving the message, the proxy will check if the subscriber had already subscribed the topic and, if it had not, the proxy will send a *NACK* message. If the subscriber had already subscribed to the topic, the proxy will check if there are new undelivered messages for that subscribed

topic. If that is true, it will send a *CONTENT* message with the next message from the topic that the client hasn't received already. Otherwise, the proxy will send a *NACK* message.

If the subscriber received a *NACK* message, the communication will end at this point. Otherwise, it will send a *RECEIVED* message to confirm that it received the message, and the proxy will simply accept that message and reply with an *ACK* message, because every request must have a reply, and end the communication.

In case there is a failure, either from the subscriber or from the proxy, there is no perfect way to make sure the message is sent exactly-once. We thought about two different options. On one side, we could just assume that the *CONTENT* message the proxy sends to the subscriber will be received and, in case it does not, when the user asks for the message again, he will receive another message and only that message will be received, effectively skipping the previous one. This would make our project execute in an at-most-once model in case of failure.

On the other side, with a *RECEIVED* message, the client needs to inform the proxy that it received the *CONTENT* message. In this case we are assuming that the proxy receives the message and if that's not true, when the user asks for a new message the proxy will think the first one was not successfully sent and will send the same message again. In this case we would have a at-least-once model.

Between the two available options we thought the best one for this project was to make sure that, if there was a failure on some parts of the connection, the subscriber would receive every message at least once.

There is a third possibility, which represents a more complex solution to the problem, but it still does not fix it and that's why we didn't implement it. The client could store information about which was the last message he received or the number of messages it had received from a topic and, by sending that information to the proxy, the proxy would be able to tell which would be the next message. Nevertheless, if the client unexpectedly shuts down, that information would be lost and, even if we wrote it to a file, there would still be the possibility of a crash between sending the message and writing to the file and, so, the file would have wrong information.

### 3.4.1 Publishing a Message

If a publisher wants to send a message to the proxy, it will send a *PUT* message with the topic and the content of the message.

When the proxy receives the *PUT* message, it will process accordingly and, in case there is at least one subscription to the topic, the proxy will save it's content, sending an *ACK* message to the publisher. Otherwise, the message will be discarded and a *NACK* message will be sent back.

In case of a byzantine failure, our program has no way of making sure the message was successfully sent and/or received and, thus, it will be up to the user to decide if it's better to resend the message and take the risk of having the message duplicated or taking the risk of having no message at all on the proxy.

## 3.5 Saving the information

All of the information needed during the execution of our solution is in the proxy side. The subscribers and the publishers only need to communicate with the proxy and don't need to store any information.

This means that, in case the proxy crashes for some reason, all the info would be lost and, upon coming back online, the program would be reset to its initial state. Since this is not the expected behaviour, we decided to store all the necessary information in disk and, when the proxy launches, it will recover its previous state by reading from that saved state.

This method has a problem: if we write to many times to a file, it will slow down the proxy, thus slowing down the whole network. This isn't the desired outcome and, to overcome this problem and to control it, the user can mention, through an argument on the program start, the preferred behaviour and thus decide how much time should take between every write to file. If the user doesn't specifies any time, the proxy state is stored once for a received operation.

In order to minimize resource usage, a "garbage collection" step is applied to the proxy's database. Messages from a topic that were already received by all subscribers are no longer needed and can be removed, as well as messages from topics that have no active subscribers (that have unsubscribed the topic). These messages are being removed so that the proxy's state is stored without useless information.

# 4 RMI Interface

In order to ease the testing and to make it easier for the end user to run the developed application, each Subscriber, Publisher and Broker (Proxy) instances are connected to RMI so that commands can be sent while the processes are running. The Subscriber interface allows the user to send commands to subscribe, unsubscribe and get messages. It also includes an extra feature called "massGet" that allows each user to specify the number of messages that wants to receive. The Publisher interface allows each user to execute put commands, as well as a "massPut" feature that works similarly to the one referenced previously. The Broker interface allows each user to log the current state of the proxy.

# 5 Results

The described implementation was developed with success, fulfilling the project requirements and was tested against several edge cases. Unit tests were implemented to test scenarios where get, put, subscribe and unsubscribe should work properly (or not). The following list describes some of the scenarios that were tested:

- GET - Multiple get calls on the same topic to check if subscriber advances in the topic's messages.
- GET - Test get call to a topic that does not exist or is not subscribed.
- GET - Test situation where RECEIVED message is not received by the proxy.
- PUT - Test put message on topic that did not exists to check if topic is created.
- PUT - Test put message on topic that already exists to check if message is added to topic.
- PUT - Test put message on topic that has no subscribers. Message is discarded.
- SUBSCRIBE - Simple valid subscribe calls.
- SUBSCRIBE - Test subscribe call to topic that is already subscribed. Message is discarded and a NACK message is sent.
- UNSUBSCRIBE - Simple valid unsubscribe calls.
- UNSUBSCRIBE - Test unsubscribe call to topic that does not exist or is not subscribed. Message is discarded and NACK message is sent.
- PROXY - Proxy crash and restart, check if state was preserved.

# 6 Conclusion

As a conclusion, this project allowed us to gain deeper knowledge on a powerful and intuitive tool to interact with sockets, that is *ZeroMQ*.

By building a reliable publisher-subscriber service from scratch, it was possible to analyse not only the implemented solution, but also other several valid architectures, understand the pros and cons of each implementation and reach the conclusion that no service is perfect. The service flaws will always have to be identified and mechanisms to overcome and minimize them will have to be implemented, as we did.