U.PORTO
FEUP ENGENHARIA

# Reliable Pub/Sub Service

## Large Scale Distributed Systems


**Class 1, Group 14**

| Full Name | Student ID |
|---|---|
| Emanuel Trigo | 201605389 |
| Fábio Huang | 201806829 |
| Sara Pereira | 202204189 |
| Valentina Wu | 201907483 |


Porto, October 21, 2022

# Contents

# 1 | Introduction

This report aims to document the design and implementation of a reliable publish-subscribe service that guarantees "exactly-once" delivery of the messages on top of *ZeroMQ* [2]. Essentially, the system is composed of subscribers and publishers. The publishers can publish a message on a topic and the subscribers of a determined topic receive the messages posted on that topic.

Accordingly, this service consists of two simple operations (*GET* and *PUT*) but also supports commands to *SUBSCRIBE* and *UNSUBSCRIBE*. A **publisher** can publish a message on a topic through the *PUT* command. A topic's **subscriber** can consume a message from that topic with the *GET* command. Despite this, a subscriber can only consume messages from a topic if it has already subscribed to that topic with the *SUBSCRIBE* command. To unsubscribe a topic and block the *GET* command, the client has to enter the *UNSUBSCRIBE* command.

To guarantee "exactly-once" delivery, in the presence of communication failures or process crashes, the service has to ensure that when a publisher enters the *PUT* command, the message will eventually be delivered to all subscribers of that topic, as long as the subscribers keep calling the *GET* command. Furthermore, the service ensures that when a subscriber enters the *GET* command, that the same message will not be returned again on a later call to *GET* by that subscriber.

Throughout this report, we will explain in detail the solution obtained and implemented for this problem. We will describe the defined architecture, all the small aspects to take into account when using our program and how the program handles unexpected situations.

# 2 | Architecture

Concerning architecture, to design a reliable publish-subscribe service that guarantees "exactly-once" delivery of the messages, we have to establish a few rules:

- By "exactly-once" we mean that no duplicated messages can be received by a subscriber but also that every message published on a topic, has to be received by all subscribers on that topic, as long as they keep calling the *GET* command.
- All subscriptions are durable, which means every subscriber needs to receive all the messages sent to a topic unless it explicitly unsubscribes from the topic or stops calling *GET*.
- Each message belongs to a topic.
- Topics are identified by an arbitrary string.
- Topics are created implicitly when a subscriber subscribes to a topic that does not exist yet.
- The *GET* command is used by a **subscriber** to request a message from a topic.
- The *PUT* command is used by a **publisher** to publish a message to a topic.
- The *SUBSCRIBE* command is used by a **subscriber** to subscribe to a topic and start receiving messages from it.
- The *UNSUBSCRIBE* command is used by a **subscriber** to unsubscribe to a topic and stop receiving messages from it.
- A **subscriber** can only consume messages from a topic if it has already subscribed to that topic.
- Messages published on a topic with no subscribers are lost since there isn't any subscriber to request that message.

That said, we now present the general architecture of our service and a description.

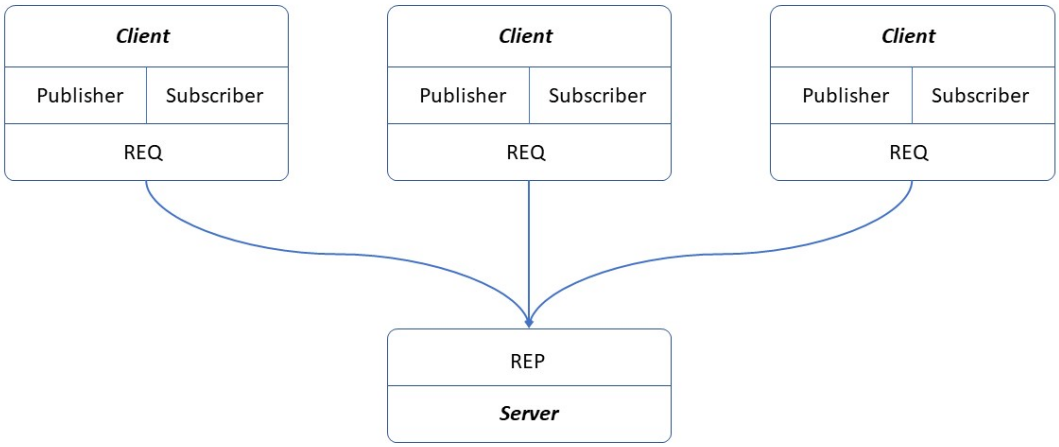


**Figure 2.1:** General architecture.

## 2.1 | Client

A client in our publish-subscribe service is a user that communicates with the server. Every client has a unique ID and we assume the clients already know their ID (there aren't clients with repeated IDs).

The client program in our implementation has two roles: **publisher** and **subscriber**.
A **Publisher** is a client that sends *PUT* messages to the server. A *PUT* message is defined by the client ID, a topic (to which the message belongs), and a message. When the *PUT* operation occurs without flaws, the publisher receives a confirmation message.
A **Subscriber** is a client that sends *GET*, *SUBSCRIBE* and *UNSUBSCRIBE* messages to the server and consumes the messages published on topics. These messages are defined by the client ID and a topic (to which the subscriber wants to receive a message, subscribe or unsubscribe).

Internally, a *GET* message sent to the server, also has the ID of the message the client wants to consume. We track the IDs of the messages that the subscribers request and receive to ensure "exactly-once" delivery as explained later.

## 2.2 | Server

The server program in our implementation receives the requests from the clients and replies to them.

When a publisher executes a *PUT* operation, the server receives the topic and the message, and stores them. When a subscriber requests a message from a topic, executing the *GET* operation, the server replies with the retrieved information from its state.

As so, the server has to store all the information about the messages published, the topics, and the subscribers as reported in the next chapter.

## 2.3 | Connection

In order to establish the connection between the server and the clients we chose a pair of **REQ-REP** sockets. With these sockets, we get a standard, strictly synchronous request-reply dialog. The REQ client must initiate the message flow and the REP server reads the request and sends a reply.
In our case, while the REQ socket talks to one peer at a time, the REP socket is connected to multiple peers. The requests are read from peers in a fair fashion, and replies are always sent to the same peer that made the last request.
Through these sockets, the clients and the server exchange all kinds of messages.

Although the **REQ-REP** combination is synchronous blocking, since the server doesn't have to wait for ACK messages from the clients as one can see in chapter 5, and doesn't slow down by doing so, we think they suit this project.

To ensure client-side reliability, we adopted the *Lazy Pirate Pattern* [1] which consists of a very simple reliable request-reply.
In the *Lazy Pirate Pattern*, instead of doing a blocking receive, the client polls the REQ socket and receives from it only when it's sure a reply has arrived. If there isn't any reply within a timeout period, the client resends the request. However, if there is still no reply after several requests, the client abandons the transaction.

# 3 | State

In this chapter, we will focus on describing how we store the states of both sides the client and the server. It's fundamental to save information about the current state in a disc to guarantee "exactly-once" delivery on publish-subscribe service.

## 3.1 | Client

Regarding the client side, we resolved to store the next message ID to be requested (through *GET*) on a topic. To organize these files considering each client and the topics they are subscribed we structured them in the following method:

```
status
├── status_clientID1
│   ├── topic1.txt
│   ├── topic2.txt
│   └── topic3.txt
└── status_clientID2
    ├── topic1.txt
    └── topic4.txt
```

As we can see in the directory above, we put all files in a folder 'status'. Then, we differentiate each client and the topic associated in different folders.

For each client, the intention of saving the next message ID to be requested on a topic is one more strategy to ensure the architecture of the service's "exactly-once" delivery. This ensures when the client crash, it can retrieve the last state of itself, and so it will invoke the correct message on the *GET* operation. Without this control, the client may receive duplicated messages due to the server doesn't perceive whether the message is delivered successfully to the client.

## 3.2 | Server

Coming now to the server side, as far as one can see, the directory path below demonstrates our storage:

```
memory
├── topic1.json
├── topic2.json
└── topic3.json
```

We have the root folder called 'memory' and for every topic that subscribers subscribe to, the server creates a JSON file with some essential information. We decided to choose a format in JSON because it's human-readable and it's easier to achieve the information we need through the nature of JSON with keywords.

The design of the JSON file is as follows:

```
{
    "topic_name" : 'name of topic',
    "subscribers": [
        {
            "subscriber_id": clientID1,
            "message_id": lastMessageID1
        },
        {
            "subscriber_id": clientID2,
            "message_id": lastMessageID2
        },
    ],
    "messages": [
        {
            "mesasge_id": messageID1,
            "message_content": 'Content of messageID1'
        },
        {
            "mesasge_id": messageID2,
            "message_content": 'Content of messageID2'
        },
        {
            "mesasge_id": messageID3,
            "message_content": 'Content of messageID3'
        }
   ]
}
```

In the JSON file, we have three fundamental keys: the *'topic_name'*, *'subscribers'*, and *'messages'*. The *'topic_name'* as the designation mentions is the name of the topic, the *'subscribers'* is where we keep the client ID on *'subscriber_id'*, and the key *'message_id'* saves the last message ID of the message requested from a *GET* operation. This will be important to clean unnecessary messages as described in the next chapter. The final and last key *'messages'* stores the information about the messages produced by a *PUT* operation: the ID of that message, and the content.

It's demanded, to update these files every time one operation occurs to ensure the server has its state updated in non-volatile memory. As so, in case of a crash it will recover as speedily as possible in an updated state.

# 4 | Cleaning Unnecessary Information

To keep the minimum amount of data possible, every message that was already received by every single subscriber in each topic is deleted from the persistent memory on the server.

To achieve this clean-up flow, the ID of the last received message is indispensable. Every time the server receives a GET command it updates the last requested message ID for the subscriber who sent the request. Then, if all subscribers received the message x all the messages with the id lower than x - 1 are deleted from the memory. Thus we can ensure that at least one message exists on each topic and if the client crashes and requests the same message, the server can still reply correctly.

To demonstrate the idea we will describe an example. Consider the current state of memory as below:

```
{
    "topic_name" : 'nameTopic',
    "subscribers": [
        {
            "subscriber_id": 1,
            "message_id": 3
        },
        {
            "subscriber_id": 2,
            "message_id": 3
        },
    ],
    "messages": [
        {
            "mesasge_id": 1,
            "message_content": 'First Message'
        },
        {
            "message_id": 2,
            "message_content": 'Second Message'
        },
        {
            "mesasge_id": 3,
            "message_content": 'Third Message'
        }
   ]
}
```

This scenery can happen when both clients subscribed to the same topic sent a *GET* command three times. In that case, the server will update the JSON file as above. Then, the server iterates over the list of 'subscribers' and calculates the `min(message_id)` requested. It advances for the array representing the queue of messages and deletes all messages with the ID lower than `min(message_id) - 1`. In our example, `min(message_id) = 3` so, the server will remove the messages with `message_id < 2`. Finally, we obtain a clean state as represented below:

```
{
    "topic_name" : 'nameTopic',
    "subscribers": [
        {
            "subscriber_id": 1,
            "message_id": 3
        },
        {
            "subscriber_id": 2,
            "message_id": 3
        },
    ],
    "messages": [
        {
            "message_id": 2,
            "message_content": 'Second Message'
        },
        {
            "mesasge_id": 3,
            "message_content": 'Third Message'
        }
   ]
}
```

# 5 | Protocols and Dealing with Losses

In this chapter, we'll explain how a client and server communicate. First, we'll talk about how the messages are exchanged in a perfect and flawless scenario, then we'll imagine possible errors and message loss and explain how our service handles them.

## 5.1 | Flawless communication between Server and Client

Considering a perfect scenario where messages are never lost and the server and client never crash, communication occurs as illustrated below in Fig. 5.1.

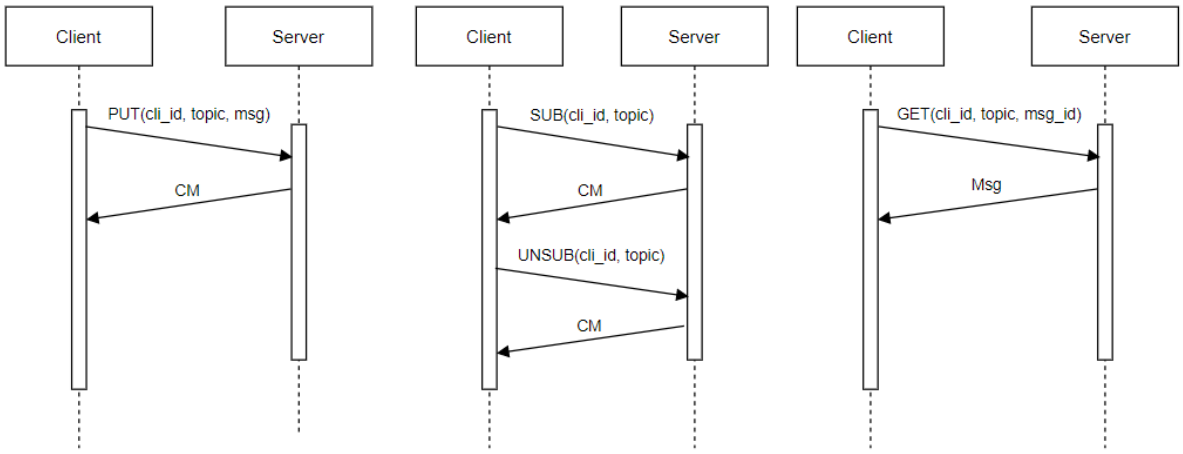


**Figure 5.1:** Client Server communication (*PUT, SUB, UNSUB, GET*). CM stands for confirmation message.

Since our client program has two roles, we'll explain each one at a time.
First, a **Publisher** sends a *PUT* message:

- PUT(client_id, topic, message) - composed of the client ID, the topic, and the message.

To which the server responds with a confirmation message or a denial message informing that the topic doesn't exist.

Conversely, a **Subscriber** sends three types of messages:

- SUBSCRIBE(client_id, topic) - composed of the client ID and the topic.
- UNSUBSCRIBE(client_id, topic) - composed of the client ID and the topic.
- GET(client_id, topic, message_id) - composed of the client ID, the topic, and the message ID.

If a client sends a *SUBSCRIBE* message, the server responds with a confirmation or a denial message informing that the client already subscribed to that topic.

On the other hand, if a client tries to *UNSUBSCRIBE* a topic, the server responds with either a successful message or an unsuccessful message (informing that the topic doesn't exist or that the client is not subscribed to that topic).

Finally, if a client does a *GET* operation, the server responds with the next message to be consumed by that client on that topic. However, if the topic doesn't exist, if there are no messages on that topic, or even if the client is not subscribed to that topic, the server just warns the client.

## 5.2 | Possible errors

There are a few cases we have to take into account to prevent communication failures or process crashes and ensure the "exactly-once" delivery of the messages.

Since the communication to execute the 4 operations is very similar (always a message sent by the client and a response sent by the server), we can generalize and obtain the following errors:

1. The Client can crash before sending the message.
2. The message sent by the Client can be lost.
3. The Server can crash before receiving the message.
4. The reply message sent by the Server can be lost.
5. The Client can crash before receiving the message sent by the Server.

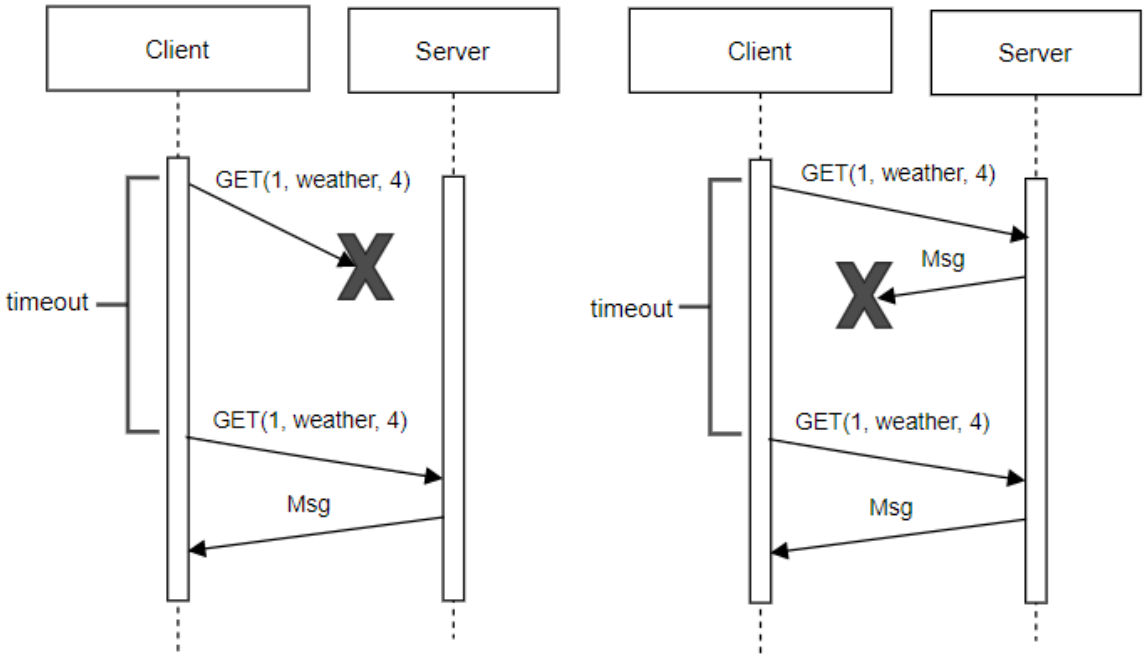


**Figure 5.2:** Client Server communication, lost messages.

To handle the first and fifth cases, the client program saves its state as mentioned in chapter 3. If the client crashes, when it recovers it will read the stored state and continue from there. For example, if a client is trying to send a GET(1, weather, 5) and crashes, when it recovers, it will send that same message GET(1, weather, 5), since it stored the next message id to be requested (5).

Cases 2, 3, and 4 are taken care of by the *Lazy Pirate Pattern* [1] explained in chapter 2 in section 3. When a client sends a message, regardless of the message type, it has to receive a response from the Server because of the synchronous communication socket. If that reply message doesn't arrive within a timeout period, the client resends the request up to 3 times before quitting.
One can think that in case 3, the server may be down for a relatively long period compared to 3 timeouts. When that happens, the client stops trying to send the request. We deal with this problem as we dealt with cases 1 and 5: since the client saves its state, it can request the same messages later on.

All of these strategies are used to ensure "**at-least-once**" delivery of the messages. This means that for each message, multiple attempts are made at delivering it, such that at least one succeeds.
Nevertheless, to guarantee "**exactly-once**" delivery, we also have to ensure "**at-most-once**" transmission of the messages. Which means that a message is delivered no more than one time.
This is established by the messages' IDs and the saved state of the client: a client doesn't request the same message if it has already received it and he knows the last message received (through its state).

It's important to remember that when a client subscribes to a new topic, since it doesn't know the ID of the first message of that topic that the server stores, it sends the first *GET* with the message ID 0. In that case, the server responds with the first message and also with the ID of that message so that the client knows the next message to be requested.

**Failure Cases**
We also have to take in mind that these strategies don't guarantee a perfect "exactly-once" delivery of the messages. If the client crashes right after receiving a message, but before storing its state, then it'll receive a duplicated message. Another failure case is when a client sends a *GET* message and the reply of the server takes too long. In that case, the subscriber will send two *GET* requests of the same message, because of the *Lazy Pirate Pattern* (the timeout of the first *GET* exceeds), and the server will think the message has been lost and will send the same message again.

# 6 | Conclusion

In this report, we presented our implementation of a reliable publish-subscribe service.
Throughout this project, we focused on satisfying the ”exactly-once” delivery of the messages taking into account possible points of failure and message loss.

Although there are many different approaches to this problem, we are satisfied with our final result since it is simple, and covers all the requested features.

In future work, we could improve this service through some changes. We could ensure asynchronous communication between the server and clients with different pairs of sockets, we could add an authentication system so that the clients don’t have to remember their ID, or even add parallel computing to the server’s request handling.

# 7 | References

[1] Pieter Hintjens and 100+ contributors. Client-side reliability (lazy pirate pattern), Ømq - the guide. https://zguide.zeromq.org/docs/chapter4/. Accessed: 2022-10-19.

[2] Pieter Hintjens and 100+ contributors. Zeromq. https://zeromq.org/. Accessed: 2022-10-12.