U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Class 3 Group 14

Alexandre Abreu	up201800168@up.pt
Diana Freitas	up201806230@up.pt
Juliane de Lima Marubayashi	up201800175@up.pt
Simão Lopes Lúcio	up201303845@up.pt

26th November 2021

SDLE Project — 2021/22 — MEIC

Professors

Pedro Souto	pfs@fe.up.pt
Carlos Baquero	cbm@fe.up.pt

Index

1 Overview

The goal of this project is to design a reliable publish-subscribe service that guarantees "exactly-once" delivery of the messages, which means that no duplicate messages can be received by a Subscriber. It also needs durable subscriptions, so 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, which is defined by a string, and the Subscribers get the messages according to the topics they are subscribed to.

Operations

The service must provide the following operations:

- PUT: A Publisher can publish a message to a topic;
- GET: A Subscriber can request a message from a topic;
- SUBSCRIBE: A Subscriber can subscribe to a topic to receive messages from it;
- UNSUBSCRIBE: A Subscriber can unsubscribe from a topic to stop receiving messages from it.

2 Architecture

The program uses the *ZeroMQ* library for messaging and can be executed in Publisher, Subscriber or Server mode. Each mode is exclusive, which means each program implements one and only one of them.

Publishers are the clients that only send `PUT` messages. Each Publisher has an unique ID, which is used in the handling of loss `PUT` messages from the Publisher to the Server. Each message sent to a topic by a Publisher has a sequential ID, so that the Server can detect out-of-sequence messages and, consequently, missing messages.

Subscribers are the consumers, and can make `GET`, `SUBSCRIBE` and `UNSUBSCRIBE` operations. Each Subscriber has an unique ID, which is used to ensure the "exactly-once" delivery.

The **Server** receives all the requests from the clients, whom only communicates with the Server. It works like a proxy, receiving the published messages and storing them in its data structures, so it can send them when a Subscriber makes a `GET` request. When a Subscriber requests a message from a topic, the Server can just retrieve this information from its state. If there is no message available, it will send the next published message to the Subscriber.

2.1 Connections

Four pairs of sockets are used for the communication between the clients and the Server, two of them for Publishers and two for Subscribers:

- **PUB** (Publisher) — **XSUB** (Server): this pair of sockets is used for publications and to forward `SUBSCRIBE` and `UNSUBSCRIBE` requests from Subscribers to Publishers;
- **SUB** (Publisher) — **PUB** (Server): each Publisher has also a SUB socket, which is used to receive messages from the Server when it detects an out-of-sequence message. To receive this messages, each Publisher subscribes a topic whose name matches its ID;
- **DEALER** (Subscriber) — **ROUTER** (Server): the main pair of sockets for the communication between Subscribers and Server, which allows Subscribers to subscribe and unsubscribe topics, and also to send `GET` requests and receive the correspondent replies;
- **DEALER** (Subscriber) — **ROUTER** (Server): to sync the Subscribers with the Server, the Subscribers send a `SYNC` message each time they start after a crash.

The Server is the only type of program that needs to read from two different sockets and, therefore, the only one that requires the use of a *ZeroMQ* poller.

The figure [1] illustrates the way Publishers and Subscribers communicate with the Server.

Motivation

The existence of subscriptions makes the PUB-XSUB sockets an interesting choice, as they allow the Server to just forward the subscription messages to the Publishers, taking advantage of *ZeroMQ's* default sockets for the PUB-SUB messaging pattern. Other socket options would invalidate this logic and the program would need to manually manage the current Subscribers of each topic, thus, not taking advantage of this library management system.

The PUB-SUB pair could also be implemented using other models, like the ROUTER-DEALER, but its usage is unilateral, so the definition of PUB-SUB makes it a desirable option for the communication model, the Server has a role of a publisher and the Publishers only receive information.

The need to know which Subscriber sent each reply justifies the choice of ROUTER-DEALER sockets in the Subscribers-Server connections. The Server sends the response only to the requester Subscriber, which is easily achieved with this sockets. A REQ-REP model would imply only synchronous messages and a PUB-SUB one would not only make the individual replies impracticable but would also require the Subscriber to store messages that he did not request with `GET`.

3 State

One of the main goals of the project was to build a distributed system that implemented an "exactly-once" guarantee regarding the message delivery. In order to accommodate this constraint, it was the authors' opinion that a state management system should be implemented for all the modules. This ensures a registry of activity for the communication between the instances of these modules and helps to support the error recovery processes to be left in place for eventual crashes or loss of information in the network.

The most sure way to guarantee an up-to-date state in the event of a crash, was to persist it at every change. But, as described in sections [3.2] and [3.3], the direct connection between the amount of state changes and message exchanges implies that, for a very high volume of data traffic in the system, modules would perform the same volume of persistent storage writes. Even though the best option, as considered by the authors, was to use separate threads for saving the state, a compromise was made. Both Publisher and Server save the changes in state every 5 messages, while the Subscriber, since the message processing cadence is usually lower than other modules, updates the state in storage at every message.

3.1 Publisher

The state of the Publisher includes only a list with the last message sent for each topic - `put_topic_dic`. It is updated and read in the following circumstances:

- When the Publisher starts, it verifies if a state file is available, and, if it is, the previous state is restored. Otherwise, a new empty state is created;
- Before sending a message to a topic, the state is read to know which is the next message to send;
- When the Publisher sends a new message to the Server, it updates the state internally.

3.2 Subscriber

The state of a Subscriber includes:

- Topic list - `topics`;
- Last message received for each topic (*ID* only) - `messages_received`;
- Last requested topic - `last_get`.

When a new Subscriber starts executing, it subscribes to one or more topics. In the case of a returning Subscriber, the previously subscribed topics are loaded from storage. When a Subscriber unsubscribes from a topic, that same topic is removed from the list. Then, in order to help validate the integrity of the sequence of messages received, for each topic, the *ID* of the last message that was received is saved. This allows the

Subscriber to send the last message *ID* back to the Server alongside with the topic it is requesting in every `get()`.

3.3 Server

The state of the Server includes:

- A list of messages for each topic, which includes the ID and content of each message - `topic_dict`;
- The ID of the last message received by each Subscriber for each topic - `client_dict`;
- A list of pending Subscribers for each topic - `pending_clients`. A pending Subscriber is as a Subscriber that made a `get()` operation when no message for the topic existed and is still waiting for a message to be published in that topic;
- A list of Publisher's states - `publish_dict`. For each topic a Publisher publishes to, a state is kept by the Server. This state includes the ID of the last message received by the Server and the list of lost `PUT` messages.

Right when it starts executing, the Server restores its previous state, if available. It then starts to read the messages from the Publishers and stores them internally, but only if there is at least a Subscriber to the topic of the message. This way, the Server can send the messages when a `GET` request arrives from a Subscriber, and he can also track the last message received by each Publisher, which facilitates the detection of lost messages.

When a Subscriber requests a message and the queue is empty, the Subscriber is added to the list of pending clients. When a message is available, it is relayed to all pending Subscribers, which are removed from `pending_clients`. When an acknowledgement is received from a Subscriber, the client's last received message is updated. When an acknowledgement message is lost the server appeals to other resources to update its state. This subject will be better discussed in the chapter 5.

Finally, to allow the Subscribers to wait for messages of topics that do not yet exist, a subscription to a topic that does not exist implicitly creates a new topic. Yet, to prevent non usable information in the state, the Server deletes the state kept for a topic when there are no Subscribers to that topic.

4 Garbage Collector

As described above, the Server stores the messages of a given topic, provided that there is at least one Subscriber to that topic. This allows the Publishers to keep publishing in any topic that has at least a Subscriber, without overflowing the queue of the Subscribers with unsolicited messages, that is, with messages from a topic they did not explicitly request with `get()`.

This solution, however, is only advantageous if there is a mechanism that, by discarding messages that are no longer needed, prevents the storage of the Server from growing indefinitely. Therefore, in this project, a Garbage Collector was implemented in order to discard messages that were already received by all the current Subscribers, a schematic code fragment is included in listing [6.1].

To implement the Garbage Collector, the Server keeps, for each Subscriber, a dictionary that stores the ID of the last message received by the Subscriber on each topic. This implies that each Subscriber, when returning successfully from `get()`, must acknowledge the reception of the message by specifying its topic and ID. With each new acknowledgement, the Server updates the last message received by the Subscriber in the acknowledged topic `update_client_last_message`.

After that, the Server runs the Garbage Collector for that topic (`collect_garbage(topic)`). It starts by identifying the first message of the topic that is currently stored in the `topic_dict`, which will be the one with the lowest ID. Then, it identifies the last message of the topic that was received by every current Subscriber `last_message_received_by_all`. Finally, given that the messages are sent by the Server and received by the Subscribers in ascending order, every message, from the first that is stored to the last that was acknowledged by all Subscribers, are discarded.

5 Protocols

This section is reserved to approach how the program handles errors and how it's supposed to work in the absence of failures.

5.1 Communication between Publisher and Server

Considering a scenario where messages are never lost and are always sent in sequence, the Publisher could keep sending `PUT(x, topic)` messages to the Server in a `PUB-SUB` approach. The schema can be checked in the appendix figure [2].

Possible errors

As a consequence of an asynchronous approach, the following errors may occur:

- The `PUT` message may be lost.
- The Publisher may crash before sending the `PUT` message.
- The Server may crash before receiving the `PUT` message.

In this type of communication, crash errors are not problematic:

- If the Server fails before receiving a `PUT` message *ZeroMQ* handles the problem: when the Server recovers from the crash, it will receive the messages in the queue. Since the Server is always saving its state in persistent memory, when it recovers from the crash it will know what was the last message sent, thus can continue to send the data it was supposed to generate.
- Since the Publisher also saves his state regularly, as long as the state is up to date, he will be able to know which message to send next when he recovers.

However, errors on the network are not handled by the *ZeroMQ* framework. Therefore, the `PUT` message is not spared from being lost. Luckily, this scenario was identified and gracefully handled: the TCP protocol guarantees that all the messages are delivered in sequence thus, if the Server receives `PUT(x)` messages out-of-sequence (e.g. `PUT(x)` followed by `PUT(x+y)` | $y > 1$), it means that a message has been lost! To handle this error case, the program uses the protocol presented in the figure [4].

In the example represented in figure [4], the Publisher tries to send 4 messages in sequence. However, the `PUT(2)` is lost. When the Server receives the message 3, instead of 2, it recognises that one message was lost and sends an `OS` (out-of-sequence) message to the Publisher, who will resend the lost `PUT(2)`.

There are two main problems with this approach: there is no guarantee that the information is delivered in sequence (even though this was not a requirement), and the `OS` message can be lost.

Asynchronous communication

To make sure that the messages are delivered sequentially, the communication between Publisher and Server could be made synchronous or, the asynchronous delivery could be kept if the out-of-sequence messages were discarded. Both approaches, have some disadvantages, which are worth to be discussed:

- The synchronous communication implies that the Publisher must wait for an `ACK(x-1)` message before sending any `PUT(x)`. This would slow down the system and would force the use of threads to keep the Server listening to other Publishers and redirecting their messages, while waiting for the `ACK(x-1)` of a certain Publisher. Besides, the creation of new threads would raise race condition environments that would also need to be taken care of.
- Still, to guarantee the sequential delivery of messages, the Server could still send an `OS(x)` message every time it detects that a message has been lost and ignore all messages with a sequence number higher than x until it receives `PUT(x)`. Event though this approach uses slightly less computational resources than the previous one, in a system with frequent errors, the performance would be sacrificed.

5.2 Communication between Server and Subscriber

Considering a scenario where failures do not occur, the protocol followed by Server and Publisher follows the schema represented in figure [3].

As the image suggests, the Subscriber requests the next message from a specific topic to the Server by sending `GET(x,topic)`, where `x` is the sequence number of a specific message in a topic. By the time the Subscriber receives `MSG(x, topic)` the Subscriber returns an `ACK(x, topic)`, attesting that he received the message, so the Server can update the state. After sending the `GET`, the `Subscriber` blocks. Therefore, the Server does not wait for the `ACK`, as it could take a long time to arrive if no messages are available on the requested topic. When an `ACK` is received, the Server uses the information to clean the state.

In contrast with the communication `PUB-SERVER`, the communication `SERVER-SUBS` may fail in many ways. Let's analyse how messages can be lost and the possible alternatives to solve the problems.

Possible errors

As explained previously, the protocol includes three types of messages:

- `GET(x, topic)`, where `x` is the sequence number of the requested message and `topic` is the desired topic.
- `MSG(x, topic)`
- `ACK(x, topic)`

Each one of these messages can be lost and a crash may occur either in the Subscriber or Server while trying to receive/send one of the messages.

Server or Subscriber crash

Server crashes do not affect the system as long as the queue limit is not reached. In this scenario, the publications, subscriptions and acknowledgements will be waiting in the respective queues when the Server recovers, and he will be able to proceed with its normal flow.

To ensure the "exactly-once" delivery even when the Subscriber crashes, two possible scenarios were considered:

1. The Subscriber crashes after returning from `GET`, but before sending the `ACK`.
2. The Subscriber crashes while waiting for a message.

In the first scenario, when a Subscriber joins before a crash, he will make sure that the last message he received was acknowledged. For that, he keeps in his state the last topic he requested (`last_get`) and sends an `ACK` for the last message he has received of that topic, which is stored in his state (`messages_received`). In the second scenario, there are two possible situations:

1. While the Subscriber was down, the Server had already replied and the message is waiting on the Subscriber's queue.
2. The Server did not have any message of the requested topic, and the Subscriber was added to the waiting queue.

To handle this cases, the Subscriber sends a synchronization message with the last topic he requested. When the Server receives this message, it checks if the Subscriber is in the waiting list of that topic and responds with `WAITING` or `NOT WAITING`. If the response is `WAITING`, the Subscriber must block waiting for a message. On the other hand, if the response is `NOT WAITING`, the Subscriber must check, without blocking, if there is a response to a previous `GET` in the queue, which might have been sent while he was crashed.

Finally, it is important to clarify that, even though the system handles the majority of the crashes, there is always a small chance that the crash occurs before the state is persisted, which can possibly result in the violation of the "exactly-once" delivery, but only in extreme cases.

ACK message is lost

If the `ACK` message is lost, the Subscriber does not resend it and does not handle this type of error. The Server is responsible for identifying that an `ACK` is lost. Before explaining how this is possible, it's important to understand what happens in the backend, until the moment the `ACK` is lost.

The message's flow starts in the Publisher, where the message is generated and then sent to the Server.The Server will receive messages from multiple Publishers, so, as it was stated in chapter 2, the Server gives new sequential numbers to the received messages, and redirects it to the Subscriber.

As a result, the sequence number of the messages received by the Subscriber is relative to the order the Server receives the information. When the Subscriber requests the information by calling `GET(x, topic)`, he will block until it receives the message. Given that the messages of a topic are delivered in the same order they are generated, the Subscriber expects to receive the messages in the same order as the Server.

Thus, imagine the case presented in figure [5].

The `ACK(1, topic)` message is lost and, as a consequence, the Server will not update its internal state about the Subscriber. However, when he receives the next request from the same Susbcriber, `GET(2, topic)`, he can assume that the Subscriber actually received the `MSG(1, topic)`, otherwise it would still be blocked or would resend the same `GET(1, topic)`.

GET or MSG are lost

This is a tricky situation. None of the many possibilities that were discussed to fix this specific error case guarantees the "exactly-once" condition. Hence, this subsection will explain some ideas and discuss the advantages and problems of each approach.

Consider the scenario specified in the figure [3]. If a Subscriber sends a `GET(x, topic)` message and it is lost, the Subscriber will keep waiting for a response forever, since the communication is synchronous. To avoid this, the Subscriber must have a timeout, and send another `GET(x,topic)` when it ends. However, the internet may be slow or the Server may have crashed, and, in these cases, the Subscriber would wrongly assume that the message was lost and a duplicate would be sent. In other words, the Server would receive repeated `GET`'s and, consequently, send duplicated `MSG`'s.

By simply ignoring recent `GET(x, topic)` messages for a certain period of time, the Server could handle this situation easily. However, we are analysing the case of lost `GET`'s as an isolated problem, when the whole picture might be a little more complex.

Suppose that, instead of a `GET`, a `MSG` is lost. By applying the approach discussed previously, a timeout on the Subscriber would result in another `GET` message being sent to the Server. During a certain period of time, the `GET` messages would be ignored, but eventually the Server would compute it and send once again the `MSG`. Finally, the Subscriber would receive the lost message.

Although this sounds a reasonable solution, it still may happen that the `MSG` wasn't lost after all. As mentioned previously, other factors may contribute to the delivery delay and, if the `MSG` was not really lost, the user will receive a duplicate `MSG`.

As a result, a measure to handle the loss of `GET` and `MSG` was not implemented: the client is responsible for stopping the program and launching it again, so that the `GET` may be sent again.

6 Appendix

6.1 Images

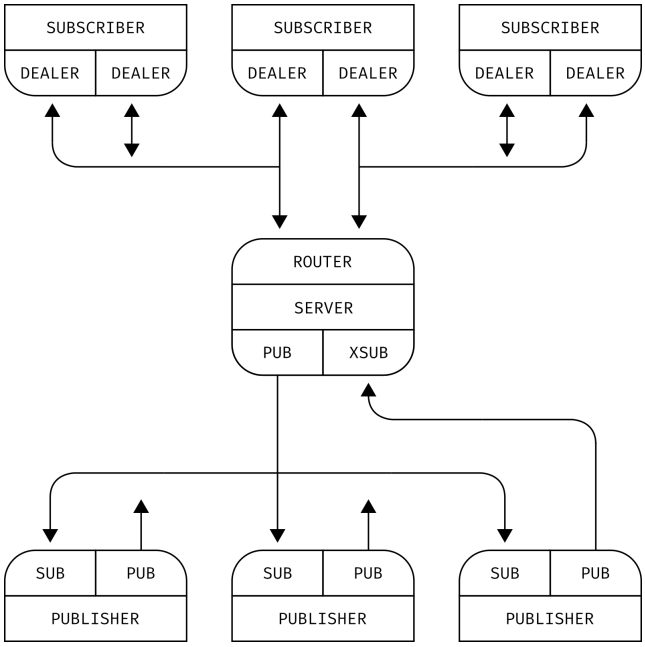

Figure 1: Socket diagram

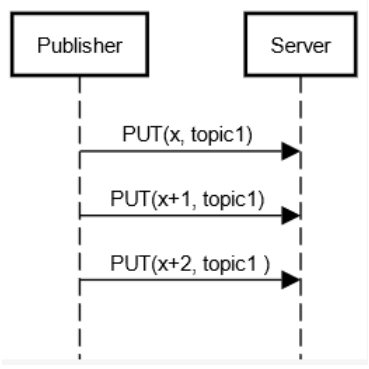

Figure 2: Publisher Server communication

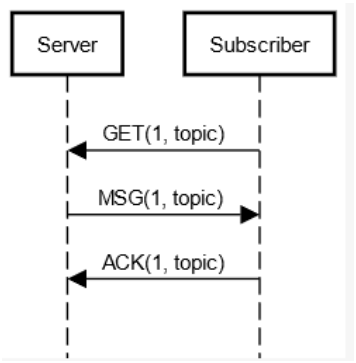

Figure 3: Server Subscriber communication

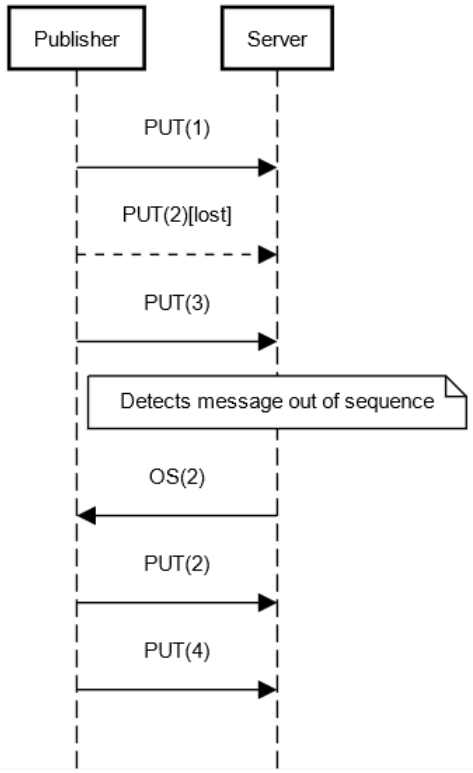

Figure 4: Publisher Server error

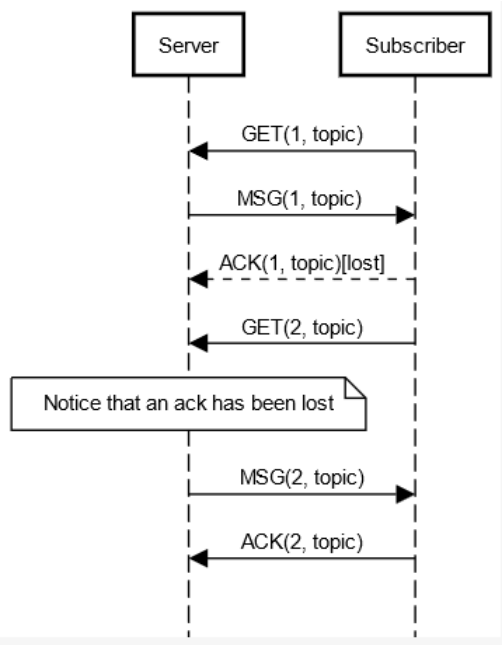

Figure 5: Server-Subscriber communication ack error

6.2 Source Code

Listing 6.1: Garbage Collector

```
def collect_garbage(self, topic: str) -> None:
        first_message = self.first_message(topic)
        last_message = self.last_message_received_by_all(topic)

        if last_message == -1:
            return
        if last_message > first_message:
            self.delete_messages_until(topic, last_message - 1)
```