Reliable Pub/Sub Service

1st project of the Large Scale Distributed Systems course

FEUP - Informatics and Computer Engineering

Class 2, Group 14

Ana Rita Ramada (up201904565)
José Frederico Rodrigues (up201807626)
José Pedro Ferreira (up201904515)
Lucas Calvet Santos (up201904517)

2022/23

Contents

1 Introduction

In this project, a reliable publish-subscribe service was developed. This service offers four operations:

- **Put**: to publish a message on a topic.
- **Get**: to consume a message from a topic. To perform this operation one must be subscribed to the topic in question.
- **Subscribe**: to subscribe a topic. This way, the new subscriber can consume future messages from the topic, by using the Get operation.
- **Unsubscribe**: to unsubscribe a topic.

In addition, the system should assure the following guarantees:

- **Durable Subscriptions**: a topic's subscriber should get all messages put on a topic, as long as it calls Get enough times, until it explicitly unsubscribes the topic. This should be guaranteed even if the subscriber runs intermittently.
- **Exactly-Once Delivery**: on successful return from Put on a topic, the service guarantees that the message will eventually be delivered "to all subscribers of that topic", as long as the subscribers keep calling Get. On successful return from Get on a topic, the service guarantees that the same message will not be returned again on a later call to Get by that subscriber.
- **Persistence**: the system should be able to recover from a crash and pick-up the information it held before the crash happened.

In order to accomplish the goals presented, we used Python's *zmq* library and the multitude of tools it offers.

2 General Architecture

The system we designed follows the **Lazy Pirate Pattern**, as it ensures the reliability using request-reply communications controlled by the client. The client sends requests to a message broker (the server), and has a timeout period and number of re-sends established. The communications follow this logic:

- Poll the Request socket and receive from it only when it's sure a reply has arrived
- Resend a request, if no reply has arrived within a timeout period.
- Abandon the transaction if there is still no reply after a certain amount of requests.

To ensure exactly-once delivery, the client-side has to keep some information, important for assuring the reliability of the service. The client keeps the number of times it has published in each topic, and, for each subscribed topic, the ID of the last received message. By having this information, the client never sends a Get request for the same message twice, because if a message is successfully received, the last message's ID for the topic in hand is updated. This ensures that a potential next Get command retrieves the next message and not the one just received. As for the Put operations, when publishing a message in a given topic, the client also provides the number of messages already posted on the said topic. This is useful when the communication fails and the client re-sends the request, because when this occurs the publisher's message count stays the same and the server can easily tell that the message is repeated, and never publish the same message twice.

The server, on the other hand, keeps for each topic:

- The list of active messages (ones that might still be requested by at least one current subscriber).
- The list of subscribers and, for each one, the ID of the latest message that is certain to have been received by the subscriber.
- The list of publishers and the number of messages that they have published in the topic.

This information complements the one kept by the client-side, and should match it. For this reason in case of mismatch of any of these parameters the server sends an error message along with its registered value (for example, in the Put operation, the publish counter).

3 Operations

As stated, four operations are offered by this server. In this next section, a more in-depth description is provided.

3.1 Get

The Get method retrieves a topic update. It takes three parameters: subscriber's ID, topic's ID and, to ensure no information is sent twice, the id of the message to be received. The server responds in one of three different ways. If all goes well, and there is a new message with the requested ID, it sends an acknowledge message with the requested content. If there is no new message, or if the client is not subscribed to the topic, it sends the corresponding error message.

3.1.1 Messages Format

Message	Sender	Format	Size
Request	Client	g subscriber_id topic_id message_id	4
Acknowledge	Server	a latest_topic_id message_text	3
Error Nonexistent Message	Server	e latest_message_id	2
Error Not Subscribed	Server	e ns	2

3.2 Put

The Put method is provided for the purpose of publishing messages on a given topic. With that in mind, the arguments of this operation are the IDs of the publisher and the topic, as well as the number of already published messages and the text message itself. Upon receiving a put message, if the server detects a mismatch on the published message count, an error is sent back to the publisher, otherwise, an acknowledge message is sent.

3.2.1 Messages Format

Message	Sender	Format	Size
Request	Client	p pub_id topic_id pub_count msg_text	5
Acknowledge	Server	a subscriber_count	2
Error Count Mismatch	Server	e publisher_count	2

3.3 Subscribe

The Subscribe method subscribes a user to a topic. This operation receives as arguments the IDs of the subscriber and the topic to subscribe. If the topic does not exist it is created by the server and subscribed. If the user is already subscribed to the topic an error message is sent, otherwise an acknowledge message is sent.

3.3.1 Messages Format

Message	Sender	Format	Size
Request	Client	s subscriber_id topic_id	3
Acknowledge	Server	a latest_topic_id	2
Error Already Subscribed	Server	e as	2

3.4 Unsubscribe

The Unsubscribe method unsubscribes the user from a given topic. All pending message updates will be lost upon unsubscribing. The client receives an error message when the server detects that they're not subscribed to that topic. Otherwise an acknowledge message will be sent by the server.

3.4.1 Messages Format

Message	Sender	Format	Size
Request	Client	u subscriber_id topic_id	3
Acknowledge	Server	a	1
Error Not Subscribed	Server	e ns	2

4 Durable Subscriptions

The service has to assure that a subscriber gets all messages put on a topic as long as it calls Get enough times, until it unsubscribes the topic in question. The server keeps for each topic the active subscribers and the last message delivered to them, only deleting them in case of unsubscription. As long as there is at least one subscriber that didn't consume a message, that message isn't deleted, and can be sent to the subscriber given that it calls Get enough times. The persistence mechanism explained below assures the durability of the subscription even if the client runs intermittently.

5 Exactly-Once Delivery

In order to achieve exactly-once delivery we need to assure that every message published will be delivered to every subscriber that calls Get enough times. We do this by only deleting messages that we know every subscriber has already received. This is assessed at each Get sent by subscribers, seeing as they always send the request with the next message ID they need. The server can assume that the client has the previous message to the one requested, because the client only updates the last message ID for each topic when it actually receives a valid message. So in each Get, the server updates the last message delivered to the subscriber in question and checks if there's now any message already sent to all subscribers. It's worth noting that when a new Subscribe request is received the server responds with the topic's latest message ID, and this is the value the new subscriber assumes as last received message ID.

In addition, we must also guarantee that a given subscriber never gets the same message following multiple Get requests. As we mentioned, the client only updates its last received message ID for each topic upon the reception of the acknowledgement with the requested message. And when this happens, a new Get will have the next ID. This way, it's impossible to receive a message and send a Get request for the same ID. In addition, Put operations are never processed twice, because the publishing counters of both the broker and the publisher must be coherent.

6 Persistence

In order to keep the program's state, persistent storage is used. The information is being saved in *JSON* files, one for the server and one for each client, containing the subscribed topics, the id of the last received message for each topic as well as the publishing count. The server's file contains all existing topics. For each topic it has all the subscribers' id with the id of the last sent message to each one, and all the publishers along with the respective publishing counts. All active messages are being saved, meaning, messages that have not yet been sent to all subscribers.

Upon execution of both the server and clients the files are read. The files are then updated with every change made.

7 Conclusion

To conclude, this project was a great way to gain deeper knowledge on the communication with sockets, specifically on the *ZeroMQ* tool, that ended up being very intuitive. By building this reliable publisher/subscriber service from scratch, it was possible to analyse not only the implemented solution, but also other several possible architectures, understand the pros and cons of each and eventually reach the conclusion that no service is perfect. The only way to overcome these imperfections is to identify the possible problems that they entail, and if avoidance is not an option, try to minimize their effects.

8 References

Reliable request-reply patterns. ZeroMQ Guide. (2021, June 17). Retrieved October 3, 2022, from https://zguide.zeromq.org/docs/chapter4/