U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática
e Computação
2022/2023 | 1º Semestre

Project 1 - Reliable Pub/Sub Service

Large Scale Distributed Systems

Bruno Rosendo | up201906334@up.pt
Henrique Nunes | up201906852@fe.up.pt
João Mesquita | up201906682@up.pt
Rui Alves | up201905853@up.pt

21/10/2022

Index

Introduction

In the context of the Large Scale Distributed Systems course, this project's goal is to build a reliable publish-subscribe service for message exchange with **"exactly-once"** delivery, implemented on top of the ZeroMQ minimalistic library which serves as the basis for the socket communication between server and clients. The required API specification has the following structure:

- **put(**topic, message**):** publish a message on a given topic.
- **get(**topic**):** consume a message from a given topic.
- **subscribe(**topic**):** subscribe to a topic.
- **unsubscribe(**topic**):** unsubscribe from a topic.

As for the technology choices, the group chose to implement the service using the C++ programming language.

Implementation Details

The implementation of this service consists of an architectural design with a broker (a single server that deals with all the requests) and clients that can be publishers, subscribers, or both over time. The service implementation was based on a reliable request-reply pattern named Lazy Pattern. Using this pattern, rather than simply blocking when the client sends a request to the broker, we:

- 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.
- Establish a new socket connection when retrying to send the request a pre-defined number of times.

Through this socket connection, the client sends requests to the broker and waits for its response, with either the acknowledgment of the request or the payload desired (in the case of a *get* request). The broker is designed to persist in its execution, except in crash situations, and keeps replying every time a client request is received. On the other hand, the client is designed to only make one request (with possible retransmissions) and end its process when the reply arrives.

Both the client and the broker are designed to be fault-tolerant. This way, they restore their state after a crash occurs by saving the necessary information in persistent storage. This mechanism is explained in detail in the sections below.

Poller and Event-Based Architecture:

The architecture of the system is event-based in two ways:

- A central server that listens to messages from clients (the events). After receiving each message, the server processes it and sends a reply synchronously. In this case, the use of ZeroMQ's sockets gives us the guarantee of eventually processing every received message instead of losing some while processing another. Otherwise, we'd need a complex multi-threaded solution to have these guarantees.
- After sending their request, the clients make use of the ZeroMQ's Poller to poll from the used socket(s) and only continue execution after receiving the server's reply or after a predefined timeout, inherently an event-based approach, In the case of a timeout, the client reconnects to a new socket and tries to send the message again.

The system is built on top of the ZeroMQ library and is mainly inspired by two known patterns: REQ/REP, used for simple interchange of messages between the clients and the broker, and Lazy Pirate, used as a simple and reliable way of ensuring the communication between those entities. Even though the use of this library brings many advantages (reduction of complexity, reliable and well-tested sockets), it is always a tradeoff. Since the service is dependent on an external library, we end up losing some degree of control over it. In this case, the group concluded that the advantages outweigh the disadvantages by a considerable amount.

Command Line Interface (Client):

To properly showcase the capabilities of the system, a command line interface was developed. It incorporates the client specified in the previous sections and provides its operations by the use of different commands.

The CLI is used with the format "client_exec <operation> <clientId> <topicId> [message]". The possible operations are *put*, *get*, *sub*, and *unsub*, which correspond to the actions in the API specification. The "message" argument is only used in *put* operations. If the arguments contain more than one word, they must be written between quotation marks (e.g. client_exec put 123 "Cooking Recipes" "Carbonara Spaghetti").

Data Structures:

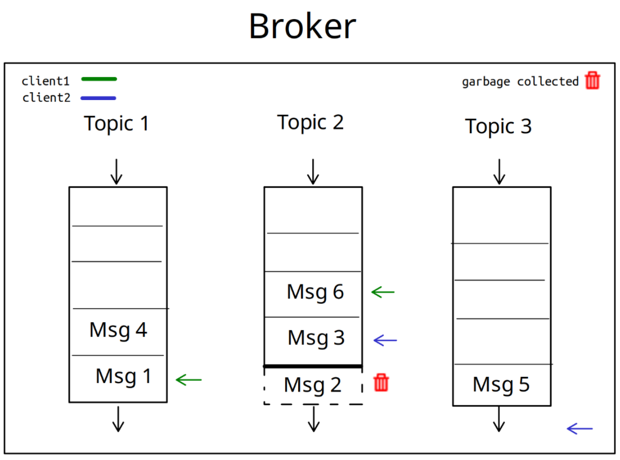
The broker side of the service requires keeping track of the information published as well as the updates triggered by the events/requests received over time. To guarantee consistency and reliability across crashes, the data structure used must be well-designed to store and provide information efficiently.

With this in mind, the main structure, responsible for storing each topic's messages and the information related to its subscribers, is a C++ map where the keys are the topics' IDs and their value is a pair containing a deque as the first element and a map as the second one. The deque stores the messages and respective IDs (another pair). The map is responsible for the subscribers' information, by storing their IDs as a key that matches a pair containing the last *get* message's ID of that topic and the index of the last message retrieved from the deque.

The map is used to access a specific topic's information in O(1) time, significantly improving the performance of the system since this is an operation done in all requests. Regarding the messages, FIFO management is expected and the deque ensures this order. At the same time, it allows for the access of the messages' information in O(1) time as well, given that the broker knows the last index accessed for each pair *(client, topic)*. It also makes it easy to retrieve the next message to be sent to the client.

Apart from the main structure, an auxiliary data container stores all the message IDs derived from *get* operations inside a set, for each topic. This improves the efficiency of the broker, by validating the uniqueness of an arriving message through a search in the set. Finally, the last structure is another map, but this time responsible for storing the message IDs of the last *subscribe* and *unsubscribe* operations for each client and topic. Hence, enabling the server to detect *sub/unsub* repeated messages and ensure "exactly-once" delivery.

The figure below partially represents the broker's data structure content after successfully processing all the requests on the right, in the displayed order. The arrows represent the last index accessed for each client on that topic.



```
 1. client sub client1 topic1
 2. client put clientX topic1 "Msg 1"
 3. client sub client2 topic2
 4. client put clientX topic2 "Msg 2"
 5. client get client2 topic2
 6. client get client1 topic1
 7. client put clientX topic2 "Msg 3"
 8. client put clientX topic1 "Msg 4"
 9. client sub client1 topic2
10. client sub client2 topic3
11. client get client2 topic2
12. client put clientX topic3 "Msg 5"
13. client put clientX topic2 "Msg 6"
14. client get client1 topic2
```

Figure 1: Flow of the broker's data structure in a usual execution

As shown in the illustration above, the server's implementation contains garbage collection. Thus, deleting irrelevant messages and topics. In the case of messages, the cleaning occurs upon a get operation, by checking who is the subscriber to the topic with the oldest messages yet to be read in the queue and preserving the appropriate message index. Then, every message with a lower index is removed. Note that the messages are deleted only when the broker is certain they are not useful. That is why in the image above the *Msg1* of *topic1* is not deleted after the *client1* performs a *get* operation since the broker doesn't have confirmation that its response was handed to the client. Therefore, it would only remove *Msg1* if it received another *get* from *client1* with a different message ID.

Unsubscribe events trigger the garbage collection of *topics*. After removing the client from the list of subscribers, the broker verifies if the updated list is empty and, if so, deletes the respective topic.

Messages and exactly-once guarantee:

This service must guarantee "exactly-once" delivery for the client's requests. This demand was accomplished by having a specific message format described below and saving the global state of the service and information about the requests in persistent storage.

Each request is sent with a payload containing the message ID (a hash generated with a nanosecond timestamp and the rest of the payload), the operation (sub, unsub, get, put), the client's ID, the topic's name, and the content of the message in case of the *put* operation. Newlines separate these 4/5 attributes.

In the case of a *put* operation, the client sends a message to the broker containing the information previously described, who then processes it by adding the new message to the specified topic and saving the ID of the request received. After that, the broker sends an acknowledgment reply and the client understands that the operation was successful. If for some reason the client doesn't receive this reply, it will resend the message and the broker identifies potential duplicates by using its ID. The *sub* and *unsub* operations work similarly.

In the case of a *get* operation, the method used has some specialties. The client sends a request to the broker and waits for its reply that contains the last non-seen message about the specified topic. If there are no new messages on that topic, a failure message is sent. Since the broker saves the ID of the last request it replied to, it can identify if the client is repeating the request (some error might have occurred and the message should be resent). This way, the broker guarantees that the same message will not be read more or less than once to that subscriber.

Failure Model

The system developed is designed to be fault tolerant. To do so, both the client and the broker make use of persistent storage to keep the information updated across the lifetime of each one and accordingly to the events that occur. This leads to improved reliability of the system, where the broker and the clients can recover from crashes in situations like:

- Before or after a topic is created
- Before or after a client is subscribed
- Before or after messages are sent to a topic

To ensure the "exactly-once" delivery in the presence of failures, such as crashes and network failures, the client stores the ID of the request it sends before doing so. The storage of each type of operation is independent of each other, to make sure they are resent after the client recovers from a crash. This behavior only happens in case the client crashes before receiving the reply from the broker. After that, the client doesn't need to store any information and it must be deleted so that the request is never resent to the broker again.

However, no system is perfect and this one doesn't cover all of the corner cases or extreme failures. For example, the persistence of the broker and client will not hold if the disk (or another storage device) malfunctions, hence breaking most of the present fault tolerance. Another situation, even if very unlikely, would be in a case where the broker is extremely flooded with requests and doesn't manage to reply before a client's timeout (potentially delaying the service until it recovers, similarly to a DoS attack).

Conclusions

This report describes the work done to implement a reliable publish/subscribe service which provides arbitrary topics and messages to be used by its clients. We understood how it's possible to guarantee that a message is delivered exactly one time and how to increase fault tolerance to a pleasing degree, even if it's not possible to cover every single scenario. The developed system achieves this by making use of ZeroMQ's REQ/REP sockets and the persistent storage of both the clients and the broker. As always. these mechanisms have their tradeoffs which we considered to be worth it in the context of this application.

To ease the test and use of the service, a CLI application was built to represent the specified API in the form of simple commands that can be executed in a terminal. The users can use this interface to seemingly interact with the system.