Assignment 1 – Reliable Pub/Sub Service

André Pereira<up201905650>
Beatriz Aguiar<up201906230>
João Marinho<up201905952>
Margarida Vieira<up201907907>

T4G12

Master in Informatics and Computing Engineering
Large Scale Distributed Systems

October 22, 2022

Table of Contents

1 Introduction **3**

2 Assumptions **3**

3 Design **3**

3.1 Client & Server 4

3.2 Broker 4

3.3 Data Structures 4

3.4 Data Storage 5

4 Protocol Implementation **5**

4.1 SUB 5

4.2 UNSUB 5

4.3 PUT 6

4.4 GET 6

5 Corner Cases **7**

6 Difficulties **7**

7 Conclusion **8**

8 Project Contributions **8**

References **9**

1 Introduction

In this assignment, we were given the task to design and implement a reliable publish-subscribe service supporting four types of operations - the publication and consumption of messages, the subscription of topics, as well as the closing of such subscriptions.

Hereupon, a publisher should be able to publish new topics so that subscribers can get messages regarding the topics they have chosen to subscribe to. Both the publishers and the subscribers, also referred to as servers and clients, respectively, interact with a broker that serves as an intermediary between the first two.

Furthermore, this service should guarantee **exactly-once** delivery of messages and handle communication failures and possible crashes.

This project was developed using the *Rust* programming language [1] on top of *libzmq* [2], a minimalist message oriented library, and was divided into five source files: *client.rs*, *server.rs*, *broker.rs*, *storage.rs* and *utils.rs*.

2 Assumptions

The following assumptions were made to guarantee the correct execution of the service calls:

1. File system is trusted, so the created files must not be tampered with.
2. There are no issues related to memory usage, so the program must be able to create as many files as needed.
3. All the service components behave as expected, so none of them has a byzantine behaviour.

3 Design

Our approach was based on an existing reliable Request-Reply pattern, the *Lazy Pirate* [3], which allows a service to attend multiple requests from different clients in a reliable manner.

The project's architecture is organized into three major entities: the subscribers (clients), the publishers (servers), and a service (broker).

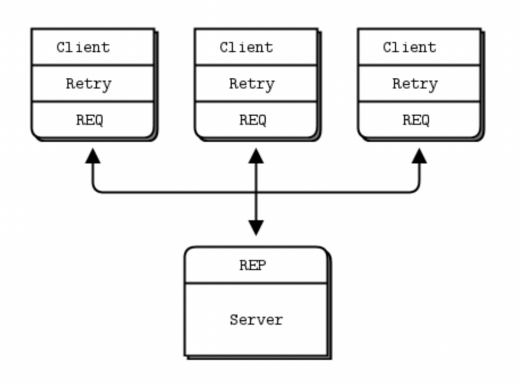

Figure 1. Lazy pirate.

3.1 Client & Server

These individuals interact with the service similarly. Despite having independent ports, from the service's perspective, they act as if they were mainly users and differ only in the types of requests they execute.

GET, SUB and UNSUB are performed by the client, while PUT is carried out by the server.

3.2 Broker

The third and final intervening developed in this project is the broker. It offers the above-stated operations, GET and PUT, besides supporting two other system functionalities, SUB (subscribe) and UNSUB (unsubscribe) to a specific topic.

After receiving each request, the service must then attend to the following while keeping a persistent state of the clients and messages available to the current environment.

3.3 Data Structures

Regarding data structures, there are essentially two to focus on - the ***struct Topic***, which keeps information about each topic, and the ***struct Storage***, which stores all the topics available in the service.

The first, *struct Topic*, has three attributes - *clients*: a HashMap which maps each client's encoded ID to their current message index; *messages*: a vector containing each message filename (a timestamp), its content and number of possible readers; *decreaser*: a number that stores the number of messages removed from the *messages* vector, necessary to retrieve the correct message upon a GET request.

As for the second, *struct Storage*, it has only one attribute, *topics*, a HashMap which maps each topic's encoded ID to the respective *Topic* struct.

3.4 Data Storage

Concerning how information is kept in persistent memory, we must describe the chosen directory structure, the file naming techniques and how data is updated in each file.

Much like our data structures, we decided to partition the Storage directory into several directories, each representing a topic. Clients and messages are stored as files in the topic client's and topic messages' directories, respectively. The decreaser value, explained further, is stored in the deacreser.txt file in the topic's root folder.

The topic and clients' directory/file names are given by their respective IDs, after a "base64" encoding, so that no strange characters cause errors upon the file creation. While the message's filename is set on the timestamp (milliseconds since the Epoch) upon receiving a PUT request. It is worth mentioning that in order to prevent the creation of equivalent files, a new timestamp is measured if an already existing file has the same filename.

Ultimately, the process of updating information occurs at the file level, i.e., whenever a message is read, or a client requests a new message, the outcome is preserved in the respective intervenient file.

4 Protocol Implementation

4.1 SUB

SUB <id> <topic>

The SUB operation subscribes a client to a specified topic.

In case the topic does not exist, the latter is created and the client id is added to the topic - stored in both persistent and volatile storage.

Initially, a topic has an empty message queue and, therefore, the client's next message is of index 0. This information is sent alongside the acknowledgement response message, being of utter importance for guaranteeing the **exactly-once** message delivery, further explained in the GET request.

In the case that the topic already exists, the client id is stored. However, since the client is only able to read messages inserted after his subscription, the message's index number returned is the number of current messages in the topic's queue.

An error message is sent when the client is already subscribed and the request is ignored.

4.2 UNSUB

UNSUB <id> <topic>

The UNSUB operation unsubscribes a client from a specified topic.

For a matter of efficiency, the service does not store clientless topics nor messages without possible readers. Hence, upon receiving an UNSUB and removing the client from the topic, the service proceeds to verify and act on these situations.

If the client was the last topic's subscriber, the theme is fully erased. For the other situation, an example, rather than an explanation, will be carried out for simplicity purposes:

A topic has 3 messages. Client 1 subscribed from the beginning, and client 2 subscribed before message 3.

If client 1 unsubscribes the topic, the service verifies all messages from the client's index and decreases by one the number of possible readers. In this case, message 1 and message 2 are erased from the topic, being left with 0 readers.

Nevertheless, there is another detail to be taken into consideration: upon a message removal, the client message's indexes and the message's vector size become inconsistent. The importance of the topic's decreaser arises in this situation. The decreaser variable is increased by one for every message removed.

Following the previous example, the topic has currently 1 message, decreaser of 2, and 1 subscriber, whose index is 2, which would result in an *index out of range* error. However, the use of a decreaser to access the messages vector solves the problem. If the client requested a message, the service would access index 0 (client's index - decreaser), returning the only message in the topic. Again and further detailed in the GET operation, this message would be posteriorly removed as there are no readers left.

4.3 PUT

PUT <topic> <message>

The PUT operation inserts a message into a specified topic.

If the topic exists, the message tuple is assembled and stored in memory. The topic's current number of subscribers corresponds to the number of possible readers placed within the message. In agreement with what was previously indicated, this information detail is crucial for the service to know if a message can be erased.

If, on the other hand, the topic does not exist, the message is discarded and an error message indicating that the topic does not exist is sent.

4.4 GET

GET <id> <topic>

The GET operation retrieves the subscriber's next message to consume from a specified topic.

Both the service and the clients are aware of the message index a subscriber is on. Upon receiving a message, the client increases its index by one and stores it, always keeping a record of the messages read. From the user perspective, the idea of an index does not exist, the program is in charge of reading the client's index from the respective file and sending it in the GET message in the format **GET <id> <topic> <index>**

The service side is trickier, the latter has no certainty if the client received the message, so it only updates the subscriber's index once he gets a GET request with a higher index than the one stored. In such case, it compares the received index with its own. If indexes match, the message is retrieved, otherwise, if the received index is higher than the known index, the service updates all data related to the known index message - decreases the possible readers by one - and retrieves the following message.

To illustrate, let's assume both service and subscriber have index 0 and the subscriber sends a GET request. As the indexes match, the service returns the message and the subscriber updates its data. In the subscriber's next GET request, the client's index is 1 and the service's is 0. This difference will notify the service that the client has already read the message 0.

This is essentially how our program guarantees the **exactly-once** message delivery.

Upon decreasing the possible number of readers of a message, the service proceeds to verify if the message was left with no readers and acts similar to when the only reader unsubscribes the topic - mentioned in the SUB operation.

In case the topic does not exist, or the client is not subscribed to the topic, an error message is sent.

When a client is subscribed to the topic and there are no messages left to read, a different message of type *WAIT* is sent for the client to keep requesting.

5 Corner Cases

Our program may only fail if the mentioned assumptions are not assured. This is a very common problem in programs trying to implement a reliable **exactly-once** semantics.

Furthermore, the message expiration case is yet another instance which can not be covered by our application.

6 Difficulties

Throughout the project's development, multiple factors caused our solution to be remodelled, nevertheless, we were capable of building a successful product.

The first is related to the programming language, since there were no restrictions for the used language, our group decided to explore a new soaring programming language, *Rust*. This decision hindered the initial development since any of us were quite experienced with the chosen language. Something that has proven to be really useful to overcome this difficulty were the ZMQ examples translated to *Rust* [4].

Another difficulty appeared regarding our initial interpretation of the project specification. At first, we thought that to develop the project we would need to use multiple parallel threads so that upon the reception of a request the work could be distributed. This proved to be unnecessary since a single process was capable of attending to all requests.

7 Conclusion

The main objective of this project was to implement a reliable and fault-tolerant publish/subscribe service with durable subscriptions.

Despite having already implemented reliable communications between two processes in previous projects, this one has given us some completely new experiences, especially regarding the usage of the ZeroMQ messaging library. Since ZeroMQ provides us with a set of functions, that hide the work needed to send messages through the transport layer, we could focus on the faulty cases that our program could arrive at.

The decision to use Rust proved to be positive, resulting in the deepening of our programming language knowledge.

8 Project Contributions

André Pereira - 25%

Beatriz Aguiar - 25%

João Marinho - 25%

Margarida Vieira - 25%

References

[1] “The rust programming language.” [Online]. Available: https://doc.rust-lang.org/book/

[2] “Zeromq.” [Online]. Available: https://zeromq.org/get-started/

[3] “Reliable request-reply patterns,” Jun 2021. [Online]. Available: https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern

[4] “Zeromq rust github.” [Online]. Available: https://github.com/erickt/rust-zmq