Reliable Publish-Subscribe Service
Large Scale Distributed Systems Report

Marcelo Couto[1] Andŕe Santos[2] Francisco Oliveira[3] Miguel Amorim[4]
up201906086@up.pt[1], up201907879@up.pt[2], up201907361@up.pt[3], up201907756@up.pt[4]
Faculty of Engineering of the University of Porto
Porto, Portugal

October, 2022

Abstract

This report will describe the implementation details of the project developed in the scope of the Large Scale Distributed Systems course of M.EIC, FEUP. The project focuses on building a reliant publish-subscribe service that ensures exactly once delivery of messages/posts using the *ZeroMQ*[1] library.

Keywords distributed systems, ZeroMQ, publish-subscribe

1 Introduction

The project proposed in SDLE class consists of a reliable publish-subscribe service that should come in the form of a library which provides 4 functions:

subscribe allows the user to subscribe to a topic

unsubscribe allows the user to unsubscribe to the topic he was subscribed

get allows a subscriber to fetch a message from a topic

put allows the user to publish a message to a topic

The system developed should guarantee exactly-once delivery of its messages, except for some rare error conditions, which means, quoting the project's statement:

- 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

In addition the project should be built using the *libzmq* library, from *ZeroMQ*. We chose to use the *Rust*[2] programming language to build our system, as it is well known for its security, reliability, speed and developer satisfaction, as well as having a libzmq binding available.

2 Project Structure

The project is divided into 3 main modules:

lib contains the message formats, subscriber and publisher context data and the functions to be offered to the user (get, put, subscribe and unsubscribe)

broker contains the code for the broker that will store and handle the published messages and deliver them to the correct subscribers

client contains the code for various scenarios of clients using our library

[1] https://zeromq.org/

[2] https://www.rust-lang.org/

3 Implementation Details

3.1 ZeroMQ Messaging Pattern

Even though *libzmq* has a Publisher-Subscriber pattern available, we decided to use Request-Reply pattern for the following reason: the *ZeroMQ* pub-sub pattern does not allow for replies on a get or put request and does not allow for the control of the message queues, which would make it impossible for us to control the exactly once factor. The reason for this is that the pub-sub pattern does not guarantee the arrival of the messages in certain scenarios, such as, quoting the documentation, "when a network failure occurs, or just if the subscriber or network can't keep up with the publisher"[1, Chapter 5]. This choice, of course, meant more implementation work, but it was a necessary sacrifice.

3.2 Message Structure

All messages are sent in the form of a serialized *Rust struct* - **Message**. This serialization step is done using a library called *Bson*[3].

```
struct Message {
    message_type: String,
    payload: Bson
}
```

The message struct is composed by a message type and a payload as depicted above. The payload is a serialized struct, also using *Bson*. This serialized struct is of a different type, depending on the type of message.

3.2.1 Get

```
pub struct Request {
    pub sub_id: String,
    pub topic: String
}
pub struct Reply {
    pub sub_id: String,
    pub message_no: u64,
    pub broker_id: String,
    pub payload: Vec<u8>,
}
```

[3]https://docs.rs/bson/latest/bson/

As we can see, the request is composed by a sub id and a topic, meaning the subscriber's id and the topic that he is subscribed to. The reply is composed also with the sub id, with a message number, that identifies the message, the broker's id and the payload.

Furthermore, on this requests we have also acknowledge requests and replies.

```
pub struct Ack {
    pub sub_id: String,
    pub message_no: u64
}
pub struct AckReply {
    pub sub_id: String,
    pub message_no: u64
}
```

As we can see above, the requests and the replies are composed by a sub id and a message's number.

3.2.2 Put

```
pub struct Request {
    pub pub_id: String,
    pub topic: String,
    pub message_uuid: String,
    pub payload: Vec<u8>
}
pub struct Reply {
    pub message_uuid: String,
    pub topic: String,
    pub broker_id: String
}
```

A Put request is composed by a pub's id, meaning the publisher's id, the topic, a message's uuid, that identifies the message and the payload of the message. Meanwhile, a reply is composed by a message's uuid, the topic and broker's id.

3.2.3 Subscribe

```
pub struct Request {
    pub sub_id: String,
    pub topic: String
}
pub struct Reply {
    pub sub_id: String,
    pub topic: String,
    pub broker_id: String,
    pub post_offset: u64
}
```

As listed above, a subscribe request is composed by the sub's id and the topic that he wants to subscribe to. A subscribe reply is composed by also the sub's id and the topic, and the broker's id and a post offset.

3.2.4 Unsubscribe

```
pub struct Request {
    pub sub_id: String
}
pub struct Reply {
    pub sub_id: String,
    pub broker_id: String
}
```

As we can see above, the structure of an unsubscribe request is the sub's id. As about the reply, it is composed by the sub's id and the broker's id.

3.3 Broker

The broker process is responsible for storing all the posts coming from publishers and for answering all the requests from subscribers. In order to do so, all the information inherent to the requests must be stored in appropriate data structures.

```
enum SubscriberStatus {
    WaitingAck,
    WaitingGet
}

struct SubscriberData {
    topic: String,
    status: SubscriberStatus,
    last_read_post: u64
}

struct TopicData {
    posts: HashMap<String, Vec<u8>>,
    post_counter: u64
}

struct BrokerState {
    broker_uuid: String,
    subs: HashMap<String, SubscriberData>,
    topics: HashMap<String, TopicData>,
    received_uuids: HashSet<String>,
}
```

The Broker's three main data structures, as shown in the pseudo-code above:

topics An *Hashmap* containing information on every topic (posts, number of last published message, etc.). Each value of the *hashmap* is a struct containing another hashmap, which links the number of each post to its payload (the number is in String format because Bson did not allow u64 to be the key for a map when serializing the map to save in a file)

subs An *Hashmap* containing information on every subscriber (topic it is subscribed to, last_read_post, etc.)

received_uuids An *Hashset* that stores the unique ids of the publisher's messages, to ensure a message is not sent twice

The usage of *hashmaps* and *hashsets* is justified by the temporal efficiency they bring to the table:

- You can get to the particular post in compile time using its topic and post number through the two nested *hashmaps* in *TOPICS* and *TopicData.posts*
- You can retrieve data from the desired user using the *hashmap* in constant time
- You can check if a UUID already exists in the *hashset* in constant time

The processing of the incoming requests is done in a simple manner, as depicted in diagram...

3.4 Publisher and Subscriber Contexts

In order to make the state of a user well defined, we created two structs that holds such data for each type of user (Subscribers and Publishers):

```
struct PublisherContext {
    pub_id: String,
    known_broker_id: Option<String>,
}
```

```
struct SubscriberContext {
    sub_id: String,
    topic: String,
    known_broker_id: Option<String>,
    next_post_no: u64
}
```

- Each of them have an **ID**. The subscriber needs an ID in order to subscribe a topic. However, both subscriber and publisher require a string ID so that the data can persist the lifetime of the client program if needed.
- Both structs contain a field alluding to the **broker ID**, for the case that the broker may have gone down and all the save data was eliminated
- The subscriber only subscribes a topic, hence this information is also useful
- The subscriber context needs to hold the number of the post it is expecting to receive, so as to ensure exactly-once message delivery.

3.5 Persistence of Data in Files

In order to insure the persistence of data in the event that something crashes or fails, the usage of files is essential so we can recover and bring the broker up in the same conditions as the previous state. Because of that, the solution was storing the 3 main data structures previously mentioned in 3 separate files. This process was executed using thread pools, so that the process stale time would be reduced and the effort split. The chosen time between backups was 5 seconds, which is a balanced option, since it isn't a short interval but isn't a huge gap either, finding a middle between performance and reliability. Every time the broker connection is raised, it tries to recover a possible previous state. In the absence of any files, it assumes that no information was stored so it starts from zero.

3.6 Error Handling

An efficient error handling method needs to evaluate swiftly if any failures occurred. Therefore, the solution was to integrate the error type in the message struct, as shown in the previous section. This way, if any error happens to come up, a message is sent with the pre-defined error type as a replacement for the pretended message(the client was already waiting for a response). On the receiver side, the reader parses the struct and when it catches an error, it parses its payload so that the right error is handled. A struct containing all error types and own solutions is defined.

4 Reliability and Exactly-Once Delivery

5 Conclusion

All things considered, the goal of developing a reliable publish-subscribe service using the ZeroMQ library, while guaranteeing the 'exactly once' message delivery, can be considered a success. The latter specification can be considered one of the most challenging matters in the assignment, since it implied the use of different detail specifications and thought process to insure it. One of the things that ended up being really positive was the chosen programming language. In the beginning, the decision of writing the code in Rust did not carry many confidence, given the learning curve and the low knowledge about it from everyone. However, after some time, this choice proved to be valuable since many of the tasks would be much more difficult in other languages, which eased the work a lot. After running different tests, the program seems to answer well to the various edge cases following the demanded requirements. In the end, the whole project was concluded successfully, and the system could be used reliably and fault-proof.

References

(1) Hintjens, P. ZeroMQ Guide https://zguide.zeromq.org/, (accessed: 17.10.2022).