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:

meic-mq library that 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 Messages

All the four functions execute similar actions:

1. Send the request
2. Receive the reply
3. Analyze the reply and take action accordingly

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 {
    msg_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>,
}
```

The reply message contains the message number so that a confirmation on the correct order of messages can be performed inside the library's functions.

The broker id exists solely to inform the user that the broker could not recover its state in a crash.

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
}
```

The goal of the *Ack* message is to guarantee that a given subscriber does not miss a message for a network failure, as will be described in the 4 section. The *AckReply* only exists due to the obligation of the broker to reply to a request per ZeroMQ Req-Rep pattern.

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
}
```

The message uuid is used to ensure a given message is not sent twice to the broker. This mechanism will be better explained in the 4 section.

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

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
}
```

The post_offset represents the current post counter on the topic the subscriber subscribed.

3.2.4 Unsubscribe

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

3.2.5 Error

```
enum BrokerErrorType {
    SubscriberNotRegistered,
    SubscriberAlreadyRegistered,
    InhexistantTopic,
    DuplicateMessage,
    TopicMismatch,
    AckMessageMismatch,
    UnknownMessage,
    NoPostsInTopic,
    NotExpectingAck
}

struct BrokerErrorMessage {
    error_type: BrokerErrorType,
    broker_id: String,
    description: String
}
```

The error message is used for when the request made to the broker has generated any problem. These errors can be both internal errors, which will be treated by the library functions, or user errors. In the latter case, the error messages are returned to the user directly through the return value of the library's functions.

3.3 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
}
```

While the *SubscriberContext* subscriber id and the number of the next post should be saved in the same structure and were the primary motivation for these, the rest of the data is merely to improve user experience and allow for a similar interface for each of the cases, subscriber or publisher.

3.4 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 Fig. 1.

4 Reliability and Exactly-Once Delivery

To ensure our service is reliable and the posts/messages are delivered exactly-once to each subscriber (in the conditions stated in the project's summary), we had to make a multitude different choices when designing our system. The problems we addressed in order to ensure these conditions were:

1. A given post was sent twice and saved duplicated in the broker
2. A subscriber did not receive the message sent by the broker
3. The broker crashes and looses its data

4.1 Duplicated Post

If a given put request was sent twice and no duplicate post verification was performed, the subscribers would eventually receive the same post twice, violating the exactly-once rule. Checking for similarities in the request payload was not a solution either, as some scenarios require messages with the same payload, like weather forecasts. For this reason, we came up with a system involving UUIDs (Universal Unique Identifier). With this system in practice, the exchange of messages occurs like this

1. Each put request is assigned an UUID as of its creation
2. Upon arrival of the request on the broker, the latter checks if there is any UUID equal to the one received on the *received_uuids hashset*
3. In case there is, the request is rejected and an error message is sent to the publisher accordingly

This solution proved to be quite simple and efficient, having only the down side of the necessity of storage of more data, the UUIDs.

4.2 Subscriber Not Receiving Get Reply

If a subscriber would not receive a message, there had to be a way for the broker to know and resend him the message. Our service uses post numeration and Ack messages to solve this.

- The numeration is used for the confirmation of the correct order of messages on arrival to the subscriber:
    1. the subscriber first receives the number of the last post that occurred before he subscribed to that topic, in order to register which was the first message this susbcriber was to receive. This information is stored both in broker (*Subscriber Data* struct) and client (*SubscriberContext* struct)
    2. each time a get request is issued and the reply arrives, an ack is sent to the broker
    3. only on the arrival of the Ack request and only if the broker is waiting for it (*SubscriberData.status*) will the Broker acknowledge that the subscriber received the post and raise the value for the post counter for that subscriber.
    4. only after this process does the subscriber check if the number of the message is correct. If the number is lower than expected, the subscriber keeps issuing get requests until it is satisfied.

This implementation of the get obviously decreases the efficiency of our system, given the Ack implies the double the travel time of messages and the repeated get requests are terribly inefficient. Even so, it does the job we wanted it to.

Note that all these mechanisms occur inside the library functions and do not require any end-user intervention.

4.3 The Broker Crashes

In order to insure the persistence of data in the event that the broker process crashes, the usage of files is essential so we can recover and bring the broker up in the same conditions as the previous state.

The solution we came up with was storing the 3 main data structures used to store the data of the broker in a file every time a change occurred. Even though this will hurt our broker's performance, we decided it was the only way to minimize the chance that a post successfully made from put would never get to a certain subscriber. This was achieved by serializing the *BrokerState* struct, and saving the result on a file (Save State step in Fig. 1).

The *broker_uuid* variable existed only to communicate a subscriber issuing a get request that the broker could not recover its previous state from a crash.

4.4 Rare Circumstances

There are a few situations where the reliability of our service is at risk:

- The broker crashes and the files with its state are wiped out - Given this situation, the previously published messages are completely lost, and all the system can do is inform the end-user through the brokerid, which is a uuid that only changes when such circumstances occur.
- The broker crashes while saving the data in files - Given this situation, the service heals but some messages can be lost forever

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. After running different tests, the program seems to answer well to the various edge cases following the demanded requirements.

One of the things that ended up being really positive was the chosen programming language. There was, at first, some doubt on this choice, given the overhead caused by the learning period of the language. However, after some time, this choice proved due to the quality of the language's compiler, making many of the tasks much easier than they would have been in other languages.

In the end, the whole project was concluded successfully, and the system could be used reliably.

References

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

A Broker Activity Diagram

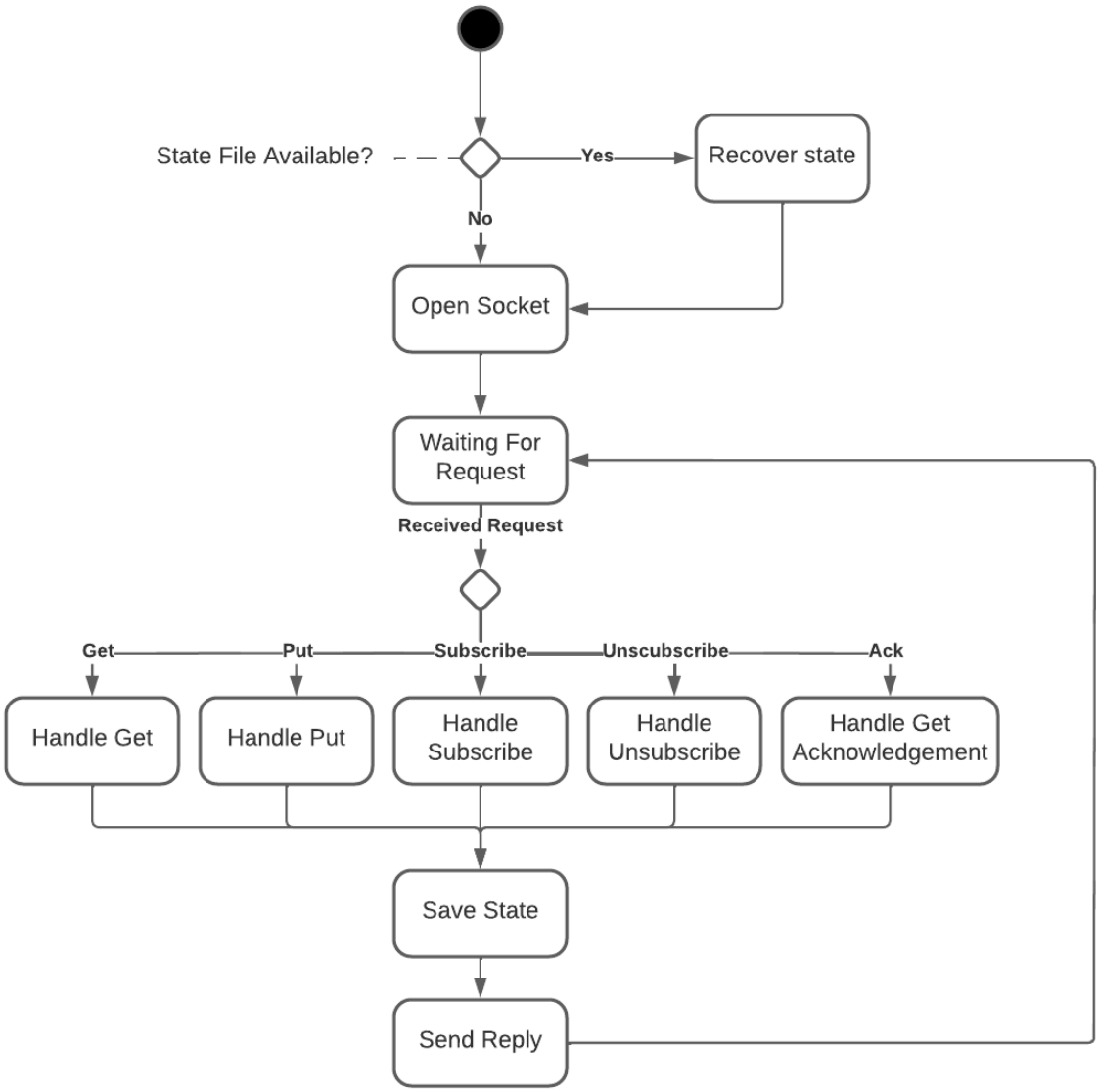

Figure 1: Broker Activity Diagram