Reliable Pub-Sub Service

Large Scale Distributed Systems

Miguel Rodrigues, Rita Mendes, Tiago Silva, Tiago Rodrigues

Faculty of Engineering of the University of Porto

October, 2022

Contents

Chapter 1

Introduction

Within the scope of the curricular unit of Large Scale Distributed Systems, we were requested to develop a reliable Pub/Sub service on top of ZeroMQ [1] - an open-source universal messaging library.

The goal here is to design a simple protocol that is the skeleton of our developed service. This service should offer a simple API and a set of guarantees, particularly fault tolerance from the different components of this service - which will be detailed in this report.

This report will first outline what our service should do and what behaviour we should expect from it. After that, we will discuss the underlying architecture highlighting several design details. And finally, in the last chapter, we will describe our protocol and showcase advantages and trade-offs, outlining scenarios where our service cannot sustain the guarantees specified in the service's specification.

Chapter 2

Reliable Pub/Sub Service

Our service's goal is to provide a reliable publisher/subscriber message pattern. It offers a pretty straightforward set of operations which will be described ahead.

2.1 What is Pub/Sub

In the pub/sub messaging, all the information follows a single direction: from the publisher to the subscribers. This means that only publishers are allowed to send messages, and subscribers are only allowed to receive them. Additionally, subscribers can filter the messages from the publisher based on a topic of their interest, and such filtering occurs by subscribing to one or more topics, if necessary.

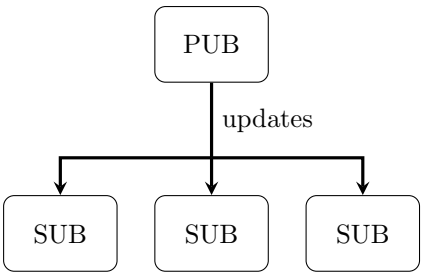

Figure 2.1: Communication flow with the Pub/Sub messaging pattern

ZeroMQ provides this messaging pattern out-of-box with the *PUB* and *SUB* socket types. However, the main requirement of this service is to be able to ensure reliability - something that ZeroMQ does not provide to their users and which is always a very difficult challenge to get done in the right way.

In the next subsections, we will discuss what is our understanding of reliability and how we tackled the challenges that emerge in the development process.

2.2 Languages and Tools

One of the requirements for this service was that it must be implemented on top of ZeroMQ. Even though we were able to choose any language that has bindings with libzmq, the fact that ZeroMQ, itself, is written in C++ forces us to take a critical decision that has a strong impact on the service's architectural design - the choice of a programming language.

The final decision was to use Rust due to its ease of use, especially thanks to the Cargo package manager, and features that enforce safety and correctness. Another point in favour of Rust was the fact that the zmq crate follows the C API closely, which allowed us to port the examples from the guide (written in C) without a significant effort.

2.3 Service's Components

The developed service is very simple and easy to understand as it is only composed of three distinct programs - the *publisher*, the *subscriber*, and the *broker*.

2.3.1 Broker

The main program is the *broker*. The *broker* is responsible for managing all the service's states, hence, a big part of the challenges were in the *broker*'s domain.

That means that both the *publisher* and the *subscriber* are completely stateless, and they must communicate with the *broker* to fulfill their operations. Since the *broker* manages all the state and it might fail, we found that was critical to provide some kind of persistence to the *broker*'s state to recover from any sort of failure or crash.

2.3.2 Publisher

The simplest of the three components is the *publisher*. This program performs the `put()` operation described in the service's specification, i.e. publishes a message on a given topic.

Essentially, it acts as a producer of messages that will, eventually, be delivered to all interested consumers.

2.3.3 Subscriber

The remaining component is the *subscriber*, which performs all the remaining operations defined in the service's specification.

It acts as a consumer, as opposed to the publisher, and it may manifest interest and disinterest in certain messages via the `subscribe()` and `unsubscribe()` operations.

On the other hand, with the `get()` operation, the *subscriber* can consume messages from a topic to which it is subscribed.

Chapter 3

Protocol

In this chapter, we will discuss our interpretation of the requirements present in the service's specification and how such interpretation materialized into a protocol capable of offering reliability in the context of durable subscriptions and exactly-once delivery.

3.1 Durable Subscriptions

Durable subscription is a concept that arises in JMS (Java Messaging Service) [2]. JMS specifies an API (for Java applications) that defines a MOM (Message-Oriented Middleware) that allows independent programs to communicate with each other by exchanging messages.

With that in mind, JMS states that durable subscriptions should outlive the subscriber's process lifetime. This means that even if a subscriber runs intermittently, a subscription only finishes when an explicit call to `unsubscribe()` is done.

3.2 Exactly-Once Semantics

Exactly-once means exactly what it suggests - messages must be delivered only once, i.e. without existing repeated messages.

With that in mind, another important aspect to recall is that, according to JMS, while a subscription is alive, all the messages in a topic should be delivered to a subscriber, as long as it keeps calling `get()` enough times. This is crucial because our protocol should ensure that those messages are not delivered more than once.

With all of this, these are the following guarantees that our protocol must follow:

1. 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()`
2. on successful return from `get()` on a topic, the service guarantees that the same message will not be returned on a later call to `get()` by that subscriber

Note that these guarantees only apply if the respective operation succeed.

3.3 Design

The protocol's design is perhaps the most challenging part of the service implementation. It has to have a lot into consideration, ranging from the message queue service - in this case, ZeroMQ - to the guarantees it is expected to provide.

Despite being a PUB/SUB service, it must guarantee reliability, which always makes things more complex, so we have implemented the protocol based on the REQ/REP messaging pattern. Also, we have taken advantage of ZeroMQ's buffering on each socket, which allows us to use ZeroMQ's polling mechanism and deal with all requests in an event-based approach.

As previously mentioned, we have a central *broker* that manages all the state. This means, that whenever the broker is down all the communications between the publisher and subscribers are ceased.

3.3.1 Publisher's End

On the *publisher* end, our protocol is very simple, i.e. a `put()` returns successfully when we are sure that the broker dealt with the incoming message. Since *publishers* do not create topics, their messages can be stored - when the topic exists - or discarded. Regardless of the message's destination, the *broker* returns an acknowledgement back to the *publisher*.

3.3.2 Subscriber's End

The *subscriber*'s end is more complex as there are more operations to be managed.

For the `subscribe()` and `unsubscribe()` operations the *broker* maintains a map that stores the list of subscribers for each topic. Whenever a request of that kind arrives, it inserts or deletes the subscriber accordingly. For instance, when a `unsubscribe()` operation occurs, all pending messages on a given topic associated to a given *subscriber* are dropped as well. Both operations are completed with success when an acknowledgment, from the *broker*, arrives.

On the other side, the `get()` operation is where a big part of the subscriber's complexity resides. First, the *subscriber* requests the next message on a given topic, to which the *broker* replies with the message itself. However, this is not enough, the *subscriber* must tell the *broker* that it received the message, so it sends back an acknowledgment to the *broker*. This last acknowledgment tells to *broker* that the *subscriber* received the message, so can to update the list containing the recipients whose a given message was not delivered yet. When the update is over, the *broker* acknowledges back the *subscriber*. When all the recipients received received a message, the broker can perform garbage collection. Recall that, for a `get()` to succeed, all this exchanging of messages has to occur without any errors.

3.3.3 Protocol's Visualization

The following figure describes how the distinct components of our service interact with each other, i.e. how the described protocol works.

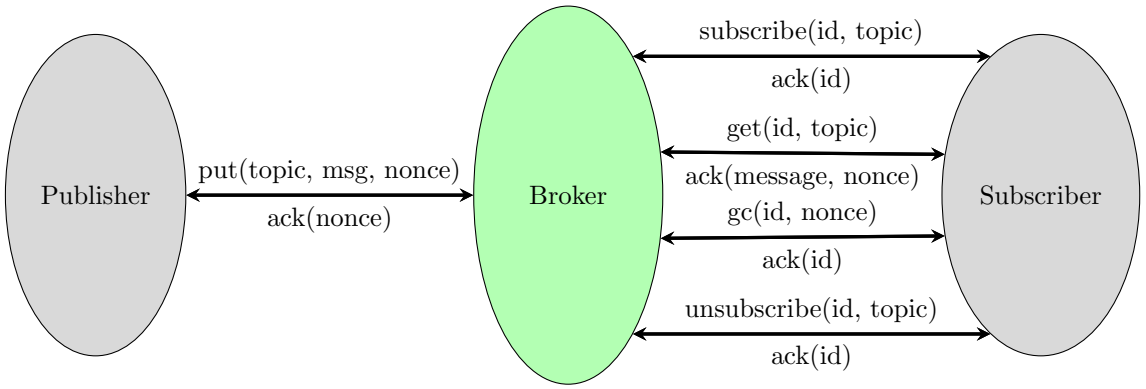

Figure 3.1: Messages exchanged in the protocol's execution

3.3.4 Failing Scenarios

Failing scenarios can be defined as scenarios where exactly-once delivery is not accomplished. Moreover, in the *Exactly-Once Semantics* we have stated that guarantees can only be achieved when the operations succeed, otherwise they are dropped.

With that said, the only scenario where we cannot ensure the expected delivery semantics occurs when the service's state persistence goes wrong.

Chapter 4

Conclusion

Before starting the development process, topics such as Pub-Sub, reliability, ZeroMQ, and Rust were a bit cloudy in everyone's minds.

With this project, these subjects became clearer, and while there is always room for improvement, it is safe to say that our implementation of a reliable Pub/Sub service is pretty decent. With that in mind and with the conclusion of this assignment, we can affirm that our knowledge about message queues, but especially about messaging patterns, increased substantially, thus making this assignment's outcome a very positive one.

Bibliography

[1] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. Oreilly and Associate Series, O'Reilly Media, Incorporated, 2013.

[2] J. Juneau and T. Telang, "Jakarta messaging," in *Java EE to Jakarta EE 10 Recipes*, pp. 491–510, Springer, 2022.