U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Implementation of a Reliable Pub/Sub Service

Project 1 Report

Eduardo Correia, up201806433
Ricardo Fontão, up201806317
João Cardoso, up201806531
David R Ferreira, up202102686

Large Scale Distributed Systems
Master in Informatics and Computing Engineering

Porto, November 2021

1 Introduction

Today, distributed systems involve thousands of entities whose location and behaviour can change through their lifespan. This large scale settings motivate the search for flexible communication models and systems that complies to the dynamic and decoupled nature of applications (Liu & Plale, 2003).

The publish/subscribe pattern is claimed to contribute to a loosely coupled form of interaction that is required in those large scale settings (Eugster *et al.*, 2003). In brief, subscribers can express their interest in an event (or groups of events) and are notified of any event generated by publishers which matches their subscription.

In this document, we will describe the design and implementation of a reliable publish-subscribe service with *Exactly-once* message delivery in mind.

Our implementation was done in Kotlin and since it compiles to Java bytecode, we used the JeroMQ library. As a build tool we used Maven.

2 Design

We designed our system in such a way that an exactly-once delivery is guaranteed. ZeroMQ already guarantees at least once delivery, as long as the system is up, so we had to ensure our service delivered messages at most once.

Starting by describing the general architecture of our implementation, it is divided in 3 major elements:

- **Broker** - The proxy responsible for coordinating the storage and delivering of messages
- **Publisher** - Sends messages to the broker that are then delivered to subscribers
- **Subscriber** - Subscribes or unsubscribes to topics and then gets the topics' messages from the broker

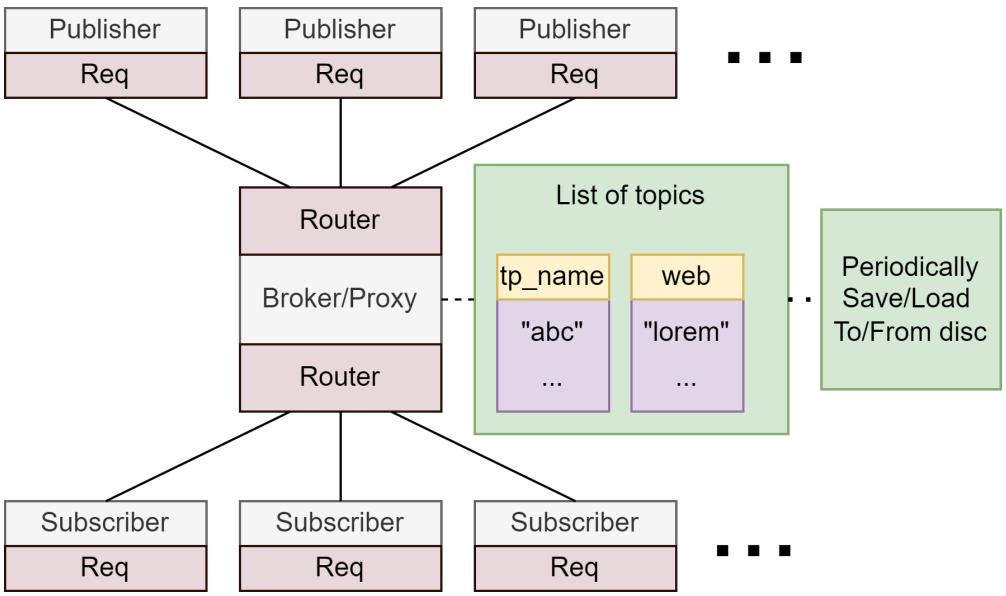

Figure 1: Overview of the architecture implemented.

Each subscriber and each publisher use a `REQ` socket and the broker uses two `ROUTER` sockets, one to deal with subscribers and another to deal with publishers.

All communication is done using those sockets. By using the JeroMQ library, the exchange of multi-part messages becomes very easy by using the `ZMsg` class. This easily allows to send multi-part messages and handle them as if they were single part ones. With that said, all communication is done using this `ZMsg` class.

The first message sent by each subscriber contains the string `"SUBSCRIBE"`, the topic to subscribe and its ID. When the broker receives this message, it processes the request and replies with message containing `"SUBSCRIBED"`.

After this the subscriber can now request messages with `get()`. The message is similar to the subscribe one, only changing `"SUBSCRIBE"` to `"GET"`.

After processing the request, the broker can respond with the desired content, `"Not_Subscribed"` in case the subscribe is not yet subscribed to the desired topic or `"Empty_Topic"`. If the response is the latter then that means no new messages were available. In this case the subscriber will retry the request until it gets a positive response.

After receiving the number of messages chosen from the topic, the subscribers proceed to unsubscribe from the topic, using the same message, only changing `"SUBSCRIBE"` to `"UNSUBSCRIBE"`. This message does not have a response from the broker.

2.1 Topics

To store each topic's messages, a custom data structure was implemented, a single *linked list*. The list includes a *head*, which points to the earliest message which has not been received by a subscriber and a *tail* which points to the most recent message.

Each element of this list represents a message and contains:

- **next** - A reference to the next element
- **data** - A string which contains the content of the message
- **subCounter** - Number of subscribers who have this message as their next
- **subList** - A list containing the ids of the subscribers who have this message as their next

Whenever a subscriber subscribes, the broker adds it to the current front of the corresponding topic's message list.

Whenever there is a `get()` request, there is an attempt to update the *head*. To update the *head*, the value of `subCounter` is checked. If it is bigger than 0 then no changes are made. If its value is 0, then no one else needs to received that specific message and it can be moved forward.

At this stage, we advance the head while there are still nodes with the `subCounter` equal to 0. This should only happen if a subscriber that is behind in the message queue unsubscribes. In most cases the *head* is only shifted to the next element.

This approach works well, because, when references are lost to the older elements, these will later be collected by Java's Garbage Collector, making this very simple to manage.

In addition to this linked list we map the subscriber IDs to a specific node of the linked list, or to null, if it already received all its messages.

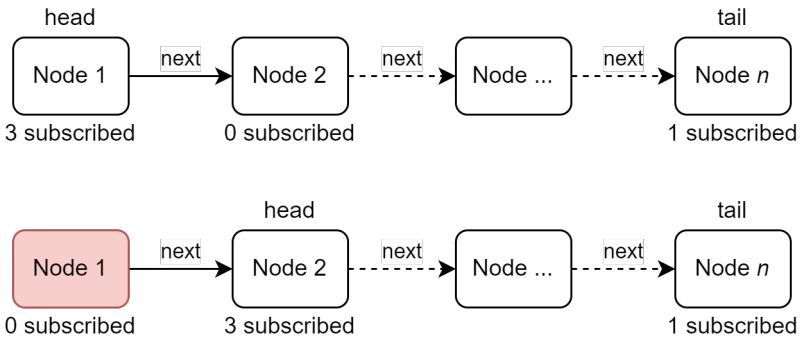

Figure 2: Example of the custom data structure implemented in order to store topic's messages.

3 Persisting Data

Every few `get/put` operations (20, chosen to couple failure prevention and efficiency), the broker saves its internal state to disk. To serialize the data, the Java `java.io.Serializable` interface was used. However, the data cannot be serialized as is due to Java crashing when the number of nodes on the linked list is too high. To circumvent this problem, when serializing the data, it is inserted into an auxiliary data structure.

That data structure is a `MutableMap<String, Pair<Map<String, List<String>>, List<String>>`.

- `String`: Topic name
- `Map<String, List<String>>`: Map containing each message in the topic. Keys are each message' data and values are a list of every subscriber whose next message is theirs.
- `List<String>`: List containing the subscribers who have already received every message in the topic.

When the `Broker` class is instantiated, it looks for a file that contains its state. If found, the broker reads it and restores the internal state of the previous one.

4 Implementation Details/Notable Decisions

If a subscriber re-subscribes without unsubscribing first, that subscription request is answered normally. However, the broker handles it slightly differently, as, instead of moving a subscriber to the end of the message list (to the most recent message), it keeps it in the same message.

This approach was chosen taking into account two main factors: The handout mentioned that if a subscriber calls `get()` enough times then it should get all messages sent by the publisher. This means that, in case of a double subscription without unsubscribing, then the subscriber should still receive all messages.

In this implementation, subscribers, publishers and the broker are all single threaded. The primary reason for this choice was that scalability wasn't our top priority, but rather reliability. With that in mind, a multi-thread implementation of the broker isn't impossible. Our implementation, however, made the introduction of threads not very beneficial, since we were using one single central data structure. This means that, on the processing of a request, the entire structure would have to be protected by a single lock, so that its consistency can be maintained. So, in the case of using multiple threads, only one at a time could enter the lock, making it basically single threaded, defeating most of its purpose.

A topic's queue/class object is only created when a subscriber subscribes to it. Any messages put into a queue without anyone subscribed would never be used, so not only are the messages ignored, but their topic isn't created either. To be noted that the publisher is never notified if this situations occurs, since this is a trait of the decoupled nature of the service. The subscriber never needs to know if someone is receiving the messages.

The broker does not save its state at every `get/put` request, since that would slow the service too much for it to be usable. The two main approaches to be considered is saving the state based on a request interval or based on a time interval.

The chosen approach ended up being the one where the state is saved based on a request interval of, for example, 20 requests. However, it should be noted that, due to our policy on `get()` requests (retry if no new messages on topic), only `get()` requests which return a topic's message count towards the interval, so that requests that don't necessarily change the state of the broker don't trigger unnecessary saves.

4.1 Failure cases

4.1.1 Broker crash (and subsequent reboot)

Since we're not saving the state for every request, the at most once requirement may not be satisfied when the system goes down. For instance, if a subscriber executed several `get()` operations that were not saved to disk and the broker disconnects, then it will not keep track of those occurrences when booting up and will consider that that subscriber did not perform such operations.

One way to mitigate this would be to save for every request, which still wouldn't fully ensure exactly-once delivery in case the broker crashes between sending the message and saving the topic data. Another way would be saving the state based on a time interval (the lower the interval the better).

In addition, if a subscriber unsubscribes and the broker crashes before it saves again, that subscriber's topic data remains in the broker's memory until the subscriber resubscribes and unsubscribes.

Because the broker doesn't retry its `put()` request, and therefore doesn't have a retry limit or timeout, if the broker crashes while processing a publisher's `put()` request before sending a reply, the publisher holds forever. The broker works as it should, even if that publisher never returns. This issue isn't present in subscribers, since they have a retry limit.

5 Conclusion

Message-Oriented Middleware can be used to fulfill the demands of modern systems with ever-increasing scales by providing a flexible communication between different entities.

The work described on this paper has shown how we can use the publish/subscribe pattern to reach this decoupled system and how we can do it by guaranteeing that each message is delivered exactly once, by filtering duplicated ones. In our implementation, the *broker* verifies if a *subscriber* has already received that message before sending it. As mentioned earlier, this "Exactly-once" guarantee implies the use of some kind of storage, which has some trade-offs, including possible performance and storage issues. Because of that, this guarantee should only be applied when it is strictly necessary to avoid duplicated messages.

6 References

Liu, Y. & Plale, B. (2003). Survey of Publish Subscribe Event Systems.

Eugster, P., Felber, P. A., Guerraoui, R., & Kermarrec, A. (2003). The Many Faces of Publish/Subscribe. ACM Computing Surveys.