Faculty of engineering

University of Porto

Master in Informatics and Computing Engineering

Reliable Pub/Sub Service

Large Scale Distributed Systems

Group members:

Bruno Campos Gomes up201906401

Fernando Luis Santos Rego up201905951

José Pedro Abreu Silva up201904775

Rui Pedro Mendes Moreira up201906355

Academic Year 2022/2023

Porto, 23th October 2022

1 Introduction

The goal of this project was to create a reliable publish-subscribe service that should guarantee ”exactly-once” delivery. The service offered 4 operations:

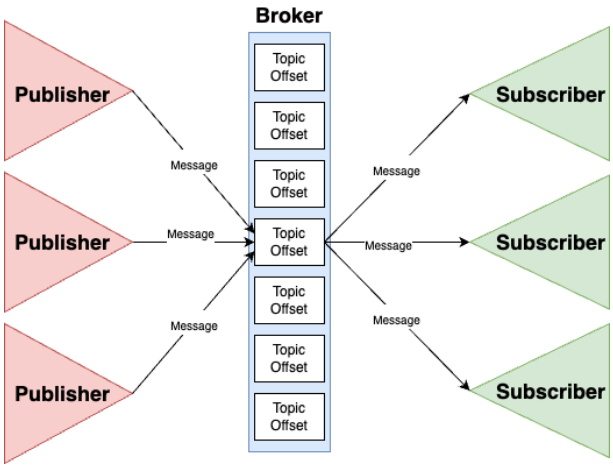
- **SUBSCRIBE** - Operation to subscribe a topic.
- **UNSUBSCRIBE** - Operation to unsubscribe a topic.
- **PUT** - Operation to publish a message in a specific topic.
- **GET** - Operation to consume a message from the subscribed topics.

Each client can subscribe to multiple topics, and the client should get all messages put on a topic, as long as it calls **GET** enough times and, in case of unsubscribing the topic with messages yet to read, we assumed that this messages are lost.

Every **PUT** operation publishes a message to a topic’s queue, so that every subscribed client can get them via a **GET** operation.

The main concept of the service was trivial to implement, however the ’exactly-once’ delivery is the tricky part that requires some planning and in this report, we explain the service’s architecture, implementation, and robustness of our fault tolerance to ensure that.

2 Design



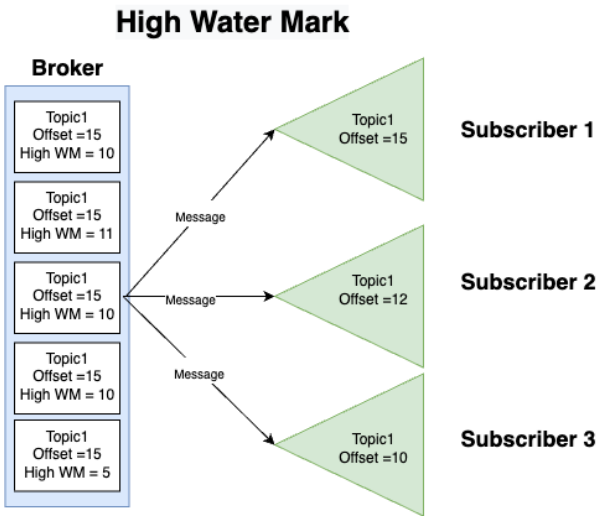
The design is based on three main components, **Subscriber**, **Publisher**, **Broker**.

- **Subscriber** - Sends **GET**, **SUBSCRIBE** and **UNSUBSCRIBE** to the **broker**. Is responsible for subscribing or unsubscribing a topic and consume the messages published in the topic.
- **Publisher** - Sends **PUT** requests to the **broker**. Is in charge of providing all the messages to the topics.
- **Broker** - Handles all operations and is in charge of maintaining the state.

In order to achieve 'exactly-once' delivery, we made the decision of having a Server/Broker system running in a single process, which made the service less scalable and efficient, however, the 'exactly-once' delivery was ensured. The clients created for this project, are the Subscriber and Publisher. These **clients** are executed only to perform a single operation, but to ensure the 'exactly-once' delivery, we keep the client state for each topic in a file related to each **client**. This approach was prioritized because the **clients** are the only components that in a normal situation perform request operations, contrariwise to the **broker**, that only replies to those requests. So there was no use in maintaining the **clients** alive when there were no operations to be executed.

2.1 High-Water Mark

In addition we decided that it would be a good optimization and principle to have a **High-Water Mark** feature, that is responsible to clear messages from a **Topic** that were already consumed and that will never be available to any **client** subscribed to that **topic**.



3 Implementation

This project was developed in Java with the aid of ZeroMQ's library with **REQ** and **REP** sockets. We used the Lazy Pirate pattern to implement the request-reply with 'exactly-once' delivery. To ensure the 'exactly-once' delivery both client and server are keeping state in order to maintain the order of the messages and to prevent the **client** to lose some message.

3.1 Broker

The **broker** consists in a structure containing topics, that can be updated by receiving and processing the **clients** requests already mentioned, and is responsible keeping the correct state to ensure fault-tolerance in case of some client is not up to date.

3.1.1 Topic

```
1  public class Topic {
2      private String topicName;
3
4      private long offset;
5
6      /** Contains all the uniqueIds of the messages **/
7      private HashMap<Long, String> messageUIds;
8
9      /** Contains the topic messages with the 'offset' associated to
           each message **/
10     private HashMap<Long, String> topicMessages;
11
12     /** Contains all the uniqueIds of the clients **/
13     private HashMap<String, Long> clientIDs;
14 }
```

3.2 Subscriber

The **subscriber** handles all the **GET**, **SUBSCRIBE** and **UNSUBSCRIBE** requests, and is in charge of storing the it's own **offsets** and receiving the desired messages.

3.2.1 Get Request

```
public class GetMessage extends Message {
    private final static String CLIENT_DIR_PATH = "./client/";
    private final String clientId;
    private final String topic;
    private final long offset;
}
```

3.2.2 Subscribe Request

```
public class SubscribeMessage extends Message {
    private final String client_id;
    private final String topic;

}
```

3.2.3 Unsubscribe Request

```
public class UnsubscribeMessage extends Message {
    private final String clientId;
    private final String topic;

}
```

3.3 Publisher

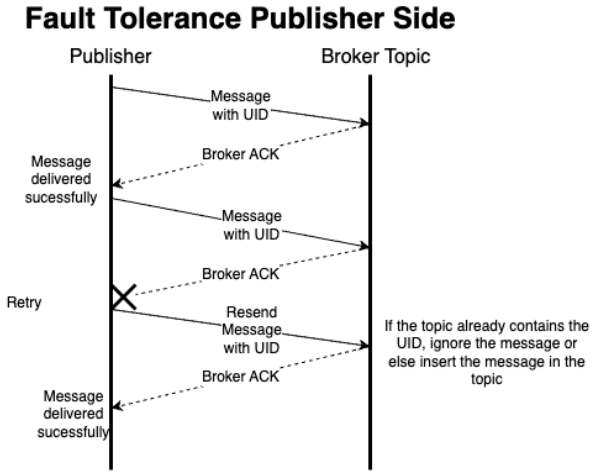
The **Publisher** is in charge of sending messages to the respective topics. Every instance of a Publisher generates a **UID** that is composed of a timestamp and the message's content to ensure no duplicate messages are processed in the **broker**.

3.3.1 Put Request

```
1 public class PutMessage extends Message {
2     private final String message;
3     private final String topic;
4
5     private final String messageUID;
6 }
```

4 Fault Tolerance

4.1 Publisher Client Fault Tolerance



The strategy that was adopted to achieve the 'exactly-once' scenario, when the Publisher client performs a **PUT** operation was to create a **Unique ID** for the message being delivered by the **client** to the **broker** and storing the **Unique ID** along with message content inside the topic.

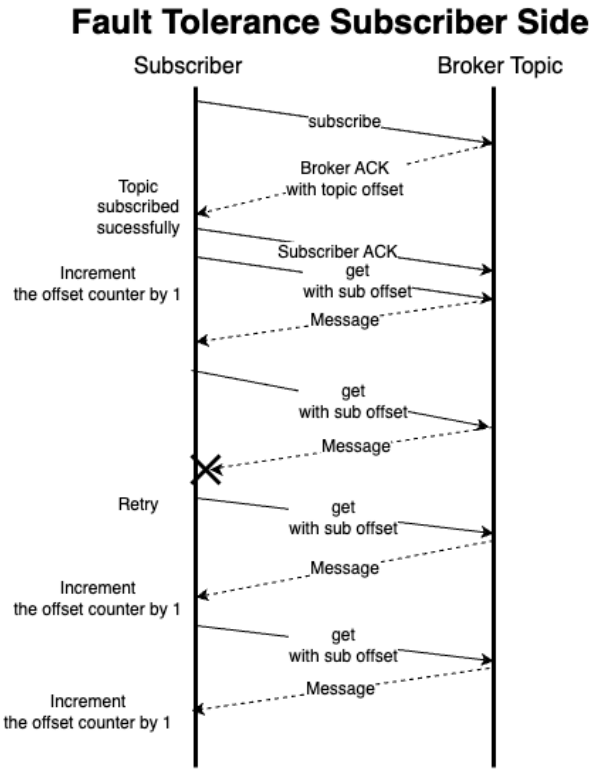
We can **Resend** the message but if the topic already contains the message being resent ignores it, and correctly acknowledges the **client**.

In a failure scenario, where:

- The **Broker** received the message, but fails to send the acknowledgement to the **client**.
- The **Broker** receives the message, but the acknowledgement didn't reach the **Client**.

In an effort to avoid **hash collisions**, when we attempt to send the same message content to the same **Broker Topic** in different requests, we added the milliseconds since Epoch to the string before hashing, with an eye towards adding an entropy source to avoid these collisions.

4.2 Subscriber Client Fault Tolerance



The strategy adopted to achieve the 'exactly-once' scenario when the **Subscribed client** performs a **GET** operation was to have an incremental counter that we call **offset** that keeps track of what the last message received by the **client** was. Therefore we are also able to synchronize this **offset** on a specific topic in the **broker** side.

If the **client** doesn't receive the message sent by **broker**, it performs 3 retries to get the message. In this case, we don't need to acknowledge the **broker**, mainly because once the **client** receives the message the **offset** is incremented and synchronized.

4.3 Broker Fault Tolerance

Since it wasn't the main focus of this assignment, we only have one **broker** available, and if it fails, the **clients** are unable to connect to it. Hence, within the scope of this assignment, we

only implemented a system that reconstructs the **Broker** to a previous state, moments before a failure.

In order to have a well-designed fault tolerance system for the **broker**, we would have to have many different **broker** instances and have replication and leader election, amongst them. In spite of the previously mentioned argument, this wasn't implemented.

5 Failure Scenarios Behaviour

5.1 Subscriber Client

5.1.1 Crashes after GET, not receiving the broker's message

If the **client** fails after the **GET** request, since the **Offset** is not updated, when the **client** recovers, the next **GET** request will retrieve the message that was not delivered before and then update the **Offset**.

5.1.2 Crashes after SUBSCRIBE, not receiving ACK

If the client fails after the **SUBSCRIBE** request, but the **broker** successfully managed to subscribe the **client** to the specific topic, if we attempt to subscribe the **client** again, the **broker** replies with **'Client is already subscribed to the topic'** in order to the client update its state.

5.1.3 Crashes after UNSUBSCRIBE, not receiving ACK

If the client fails after the **UNSUBSCRIBE** request, but the **broker** successfully managed to subscribe the **client** to the specific topic, if we attempt to subscribe the **client** again, the **broker** replies with **'Client is already unsubscribed to the topic'** in order to the client update its state.

5.2 Broker

5.2.1 Crashes after GET, not replying ACK

If the **broker** crashes after receiving a **GET** request, the **client** will try again until the **broker** replies, or will abort if it exceeds 3 tries. It will return the correct message, because it will return the one with corresponding from the **GET** request's **offset**.

5.2.2 Crashes after PUT, not replying ACK

If the **broker** crashes after receiving a **PUT** request, the **client** will try again until the broker replies, or will abort if it exceeds 3 tries. The following **PUT** requests won't be have an effect on the **broker**'s state, and it will work as if only one request was sent.

5.2.3 Crashes after SUBSCRIBE, not replying ACK

If the **broker** crashes after receiving a **SUBSCRIBE** request, the **client** will try again until the **broker** replies, or will abort if it exceeds 3 tries. The following **SUBSCRIBE** requests won't be have an effect on the **broker**'s state, and it will work as if only one request was sent.

5.2.4 Crashes after UNSUBSCRIBE, not replying ACK

If the **broker** crashes after receiving a **UNSUBSCRIBE** request, the **client** will try again until the **broker** replies, or will abort if it exceeds 3 tries. The following **UNSUBSCRIBE** requests won't be have an effect on the **broker**'s state, and it will work as if only one request was sent.

6 Conclusion

To finish, all the goals of this project were concluded with success:

- The **publisher** is able to use *put()* in order to publish a message in a specific topic.
- The **subscriber** is able to use *subscribe()* in order to subscribe a certain topic.
- The **subscriber** is able to use *get()* in order to consume a message from the subscribed topic.
- The **subscriber** is able to use *unsubscribe()* in order to unsubscribe a certain topic.
- The **'exactly-once'** delivery was ensured.
- All the normal possible failure scenarios are well handled but there may be rare circumstances that the system may fail.

References

[1] Zhang N., "Exactly-once" Semantics across Multiple Kafka Instances Is Possible , Jan 2021.

[2] Narkhede N., Exactly-Once Semantics Are Possible: Here's How Kafka Does It , Jan 2017.

[3] Baeldung, Exactly Once Processing in Kafka with Java , Aug 2021.

[4] Baeldung, Gson Deserialization Cookbook , Jun 2022.