U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Large Scale Distributed Systems

Project 1 - Reliable Pub/Sub Service

Class: 7 Group: 13

26 de November de 2021

António Cadilha da Cunha Bezerra **up201806854**@fe.up.pt
Catarina Justo dos Santos Fernandes **up201806610**@fe.up.pt
Gonçalo de Batalhão Alves **up201806451**@fe.up.pt
Pedro Jorge Fonseca Seixas **up201806227**@fe.up.pt

Overview

The goal of this project was to develop a reliable publish-subscribe service. To create this service we implemented a server, a client, and a provider, which handles the communication between the last two. The server is the main part of the service, managing the requests made by the clients, which can be publishers, subscribers, or both.

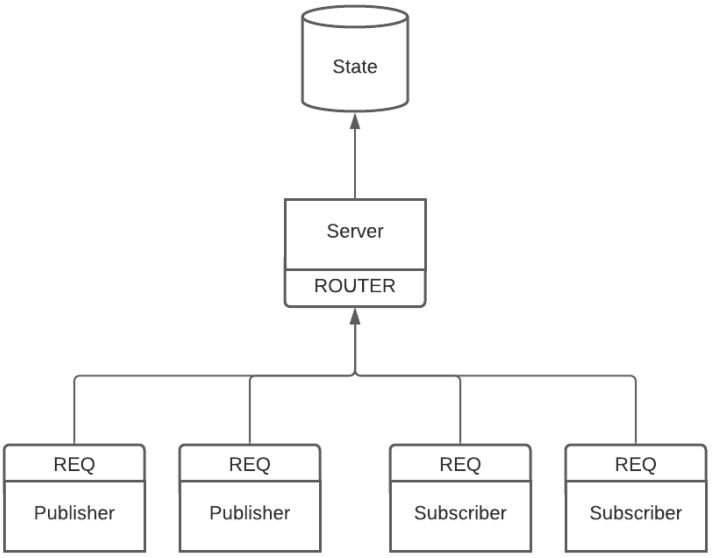

Figure 1: Service Architecture

The server keeps information on the currently existing topics, their subscribers, and the messages that each one of them needs to receive. The server saves the state in non-volatile memory at a fixed delay, in order to reduce the message losses when a crash occurs.

The client uses the provider to connect and communicate with the server. A client can **subscribe** and **unsubscribe** from a topic. From the moment the client subscribes to a topic, the server starts to save the messages put on that topic that need to be sent to the client. All subscriptions are **durable**, which means that, after subscribing to a topic, the only way that that subscription will end is by explicitly sending an unsubscribe message. This ensures that a topic's subscriber should get all messages put on a topic, as long as it calls *get()* enough times. Publishers can publish a message to a topic by using the *put()* call and subscribers should call *get* whenever they want to receive a message from a topic.

The service guarantees an "exactly-once" delivery in the presence of most types of communication failures or process crashes, except in some isolated extreme circumstances. Whenever a successful call to *put* is made, it is guaranteed that if the subscribers call *get* enough times, they will receive exactly once the message and they will never receive a duplicate one.

Implementation

Provider

In order for the client to communicate with the server, the **Provider** was created with multiple functionalities:

- **connect(id:** *byte[]***)** - Initiates a new *zmq::Context*, creates a **ZMQ_REQ** socket with identity *id*, and connects it to the server. When connecting to the server, an "ONLINE" message is sent to the server, for it to know that a new client has just become online. The purpose of this message is explained later in the Fault-Tolerance section.
- **subscribe(socket:** *zmq::Socket***, topic:** *String***)** - Subscribes to a topic. The function builds the message with the format: "SUB [*topic*]", sends it to the server, and waits for its response. This call also creates a topic if it does not exist already.
- **unsubscribe(socket:** *zmq::Socket***, topic:** *String***)** - Unsubscribes from a topic. The function builds the message with the format: "UNSUB [*topic*]", sends it to the server, and waits for its response.
- **get(socket:** *zmq::Socket***, topic:** *String***)** - Get a message from a topic. The function builds the message with the format: "GET [*topic*]", sends it to the server, and waits for its response. There are three possible responses:
  - "NF" - When the topic does not exist in the server;
  - "NS" - When the topic exists in the server but the client is not subscribed to it;
  - "OK *message*" - When the message arrived successfully.
- **put(socket:** *zmq::Socket***, topic:** *String*, **value:** *String***)** - Publish a message in a topic. The function builds the message with the format: "PUT [*topic*] *message*", sends it to the server, and waits for its response. If the topic does not exist, the server will just silently discard the message, since there are no subscribers to receive it.

The main function of the **Provider** is to isolate every communication from the client. From the client's perspective, all he has to do is to call the function corresponding to the action he wants, with the necessary arguments, and the provider handles all the message exchanges with the client.

Sockets

The sockets used either in subscribers or in publishers were **ZMQ_REQ** sockets, and the server used a **ZMQ_ROUTER** socket.

After trying out different architectures with different sockets we concluded that these kinds of sockets would be the ones that would facilitate our work in order to develop a service that guarantees that every message is delivered to every subscriber exactly once, assuming the subscriber calls **get()** enough times.

ZMQ has built-in publisher-subscriber pattern-oriented sockets, **PUB**, **SUB, XPUB,** and **XSUB.** After testing with these sockets, we found it hard to ensure that subscriptions were durable, as the handout suggests, since every time the publisher or the subscriber crashed, the socket would unsubscribe the topics that were supposed to be kept subscribed. Because of this, we opted for the above-mentioned sockets.

In the documentation of the **ZMQ API** the following paragraph can be found that pertains to **ZMQ_REQ**: "*If no services are available, then any send operation on the socket shall block until at least one service becomes available. The REQ socket shall not discard messages.*". This ensures the **durability** of our service, since either a **put(), get(), subscribe(),** or **unsubscribe()** will block the **pub/sub** when the service is down. When the server gets back online it will receive all these pending requests.

We also used **ZMQ_ROUTER** since this socket not only allows for a large number of connections to clients, using **polling**, but it also allows us to know when a client is down, based on the throw of an exception, using the flag **ZMQ_ROUTER_MANDATORY.**

Data Structures

The data structures used in the service were two HashMaps, one for the topics and all their respective information, and another for the pending requests.

The topics HashMap has an entry for each topic, using the topic string as the key. The value is another HashMap, where the key is the *id* of the subscriber and the value is a queue of messages to be sent to that subscriber. The structure is as follows:

```
{
      topic1: {
            sub1: [Message1, Message2, Message3, ...]
            sub2: [Message1, Message2, Message3, ...]
            ...
      }
      ...
}
```

The pending requests HashMap has an entry for each topic that has pending requests. These pending requests represent the subscribers that are blocked in a *get* call to that specific topic. When a publisher puts a message on that topic, the server will send that message to every subscriber that is waiting for a message in the pending requests hashmap. The hashmap has the following structure:

```
{
      topic1: [sub1, sub2, sub3, ...]
      topic2: [sub1, sub2, sub3, ...]
      ...
}
```

Concurrency Design

In our service, as we will explain later in this report, we used a single-threaded server to handle the requests with the objective of keeping the order of the requests. Besides that, we have another thread that is not responsible for handling requests and whose task is only to save the state to non-volatile memory. The main objective of this implementation is to reduce the losses when a server crash occurs.

We have implemented our data structures inside a Mutex so that when the server is saving the data structures to a file, it is not being changed, which could cause message losses or inconsistencies.

Scalability

In terms of scalability, our service is not the easiest to scale due to the fact that we developed a more reliability-focused service. We opted for a single server implementation, which is not the best choice when concerning scalability but has helped us significantly ensure the “exactly-once” delivery of every message. The single server also allowed the service to maintain a request order, which was crucial to define which were the subscribers of a topic whenever a message was put to that topic.

Fault-tolerance

The service aims to guarantee fault-tolerance for every situation that might occur. This means that when, either the server or client, goes down before, during, or after sending a message, there will not be any message losses. To achieve this, it was key to choose the correct implementation features and also consider some extra precautions. Some of the most relevant cases are:

1. Client crashes after sending a message, and before receiving the server reply

This situation was handled with an option on the ***ZMQ_ROUTER*** socket. When sending messages a ***ZMQ_ROUTER*** socket uses the first part of the message to determine the routing id of the client the message shall be routed to. If the peer does not exist anymore (as if it had crashed), when the option ***SET_ROUTER_MANDATORY*** is set to 1, the socket fails, returning an error message which is handled by the server.

2. Client crashes while blocked in a GET call, and is restarted before there is any PUT call to that topic

As we previously demonstrated, in order to implement a blocking GET call, we needed to create a data structure to keep the pending requests that should be handled whenever a PUT call was made to a topic. If for some reason, the client that had a pending request crashed, and was back online, a PUT call would make the server send that message to the subscriber who had a pending request, but as he wasn't expecting to read any message from the server, the message would be lost. To forearm this situation, the *ONLINE* message was implemented. With this message, when a client goes online, the server knows about it and checks if it had any pending request to that client, and, if so, it silently discards that request.

3. Server crashes before the client sends a request

With the use of ***ZMQ_REQ*** sockets, if no services are available, then any send operation on the socket shall block until at least one service becomes available, so, when the server restarts, it will handle the request.

4. Communication channel fails

By using TCP channels to communicate, if, for any reason, there is an error in the transport layer, an exception will be thrown and caught by the server or the client.

Trade-offs

For the server, although the best performance could be achieved through multithreading for handling requests, we have implemented the server in a single thread for one main reason: to keep the requests in the order that they have arrived. This order helped us to define which clients were, or weren't, subscribed to a topic when a message was sent to that topic. Another reason that made us choose a single-threaded request handling server was the fact that every request needed to use the HashMap, whether to delete or add new entries, which meant that each request could also only be handled after the one that was currently being handled. The differences in performance were not significant enough to make us change our implementation.

With the server only handling one request at a time and being that request blocking, if the server crashed in the middle of the communication, we haven't found a way to warn the client that the server was down, so it will just stay blocked until termination. We needed to keep this trade-off in order to avoid duplicated messages. If we implemented a time-out in the client that would retry to send the message if it did not receive a reply, there would be a situation where the server would handle the request but crashed before sending the response and, after restarting, would receive the same request again, and handle it again, resulting in duplicated requests.