Reliable Pub/Sub Service

Implementation in Python + ZeroMQ

Duarte Sardão*, Henrique Sousa †, Mateus Silva‡ and Melissa Silva §

*FEUP. *up201905497@up.pt*
†FEUP. *up201906681@up.pt*
‡FEUP. *up201906232@up.pt*
§FEUP. *up201905076@up.pt*

I. Introduction

The following report provides insight on work for the first project of Large Scale Distributed Systems (*SDLE*). This report in different sections to better showcase the different facets of the group's implementation.
The group's main and only objective is to end finish the project in possession of a reliable application for the intended purpose, along with understandable (inline documentation) and report.

II. Problem Description

The proposed problem involved a service that offers two simple operations: *put()* and *get()*, that deal with arbitrary byte sequences (messages). Those two operations are joined by *subscribe()* and *unsubscribe()*, the first being used for the implicit creation of a topic.
A topic is an identifiable message queue, for which all subscriptions are durable and should provide all of its messages as long as a client calls *get()* enough times. A subscription must be independent of the lifetime of the process that makes said subscription.
The service should guarantee *exactly-once* delivery, in the presence of failures or process crashes, outside of some rare circumstances. The *exactly-once* guarantee means:

- On successful return from *put()* on a topic, the service guarantees the message will eventually be delivered to all subscribers on that topic as long as they 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 *get()* by that subscriber.

III. Implementation

A. The Lazy Pirate Pattern

Considering the different approaches and patterns seen during practical classes, a lot of possibilities were available. The group decided to go with the ***Lazy Pirate Pattern***, having client-side reliability in mind.
As per the ZeroMQ documentation [1], the pattern uses polling instead of a blocking receive. Generally, the pattern:

- Polls the *REQ* socket and receives from it only when it's sure a reply has arrived;
- Resends a request whenever no reply arrives within a timeout period;
- Abandons a transaction whenever a sequence of several requests yields no replies.

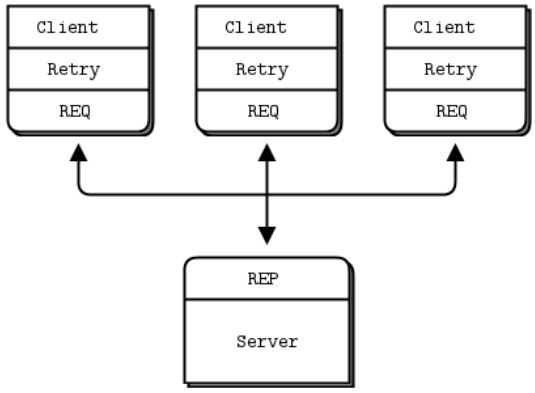

Fig. 1: Lazy Pirate Pattern Diagram.

The suggested brute force solution of closing and reopening the *REQ* socket after an error was used in the implementation.

B. Operations

The problem was approached using object-oriented programming. This is quite easy to do in Python. For better understanding of this, the report includes an UML diagram (see **Fig. 2**) that can be analyzed while reading the following sections.
All implemented solutions defaulted to the simplest data structures and implementations: lists, strings and integer variables.

1) Subscribe

Starting off, it's checked if a **Client** is already subscribed to the topic they are trying to subscribe to - if this is the case, a message that points to this occurrence is printed. Otherwise, a Subscribe Message is created that is sent to the Server (using *send(message)*). This process follows the **Lazy Pirate Pattern** - there is an attempt to send the message and should it be unsuccessful, three other attempts are made (*REQUEST_RETRIES*).

In **Server**, it's checked if the identified topic by the Client making the subscribe exists, if not, the topic is implicitly created and the subscription process occurs in *Topic*. It's in that class that it's checked if the Client isn't already subscribed to a topic - in case it's not, a new instance of *ClientStatus* is created. This is a class that merely couples the client's ID and the ID of the latest message from the topic.
After a successful run of this process, the execution goes back to **Server** where an *ACK_SUBSCRIBE* message is sent, acknowledging the receiving of the subscribe message from the Client to show that the process was successful.

2) Unsubscribe

During an unsubscribe process, the **Client** first removes the topic from its own list of subscriptions.
Following this, an Unsubscribe Message is created and sent to the Server, once again following the **Lazy Pirate Pattern**.
In **Server**, the identified topic subject to the operation has its *unsubscribe(client_id)* method called. If successful, an *ACK_UNSUBSCRIBE* message is created and sent back to the Client.
A check for topic deletion is also performed in Server: if a topic has no subscribers, it is automatically deleted and this is reflected in the stable storage files.

3) Put

The **Client** creates a Put Message that it intends to put on a topic - this is done by getting the message to the Server.
The **Server** receives the messages, checking the already created topics. If it finds that the topic hasn't been created yet, it's created implicitly.
In either case, the new message is added to the respective topic. The message number is also stored, to proceed with its deletion once every Client has received the message.

4) Get

When a **Client** calls the get method, it itself checks if it's trying to get a message from a topic they aren't subscribed to. If that is the case, a fitting message is printed.
A Get Message is sent to the Server, once again using the **Lazy Pirate Pattern**. The **Server** then checks if the topic exists and assuming it does, the *get(topic_id, message_id)* method from Topic is called.
Topic checks if the identified message requested wasn't already delivered, if it has, due to the the *exactly-once* restriction the implementation must oblige, the message won't be delivered to the Client - they'll get a *NACK_GET* message, to point out the occurrence.
In case this doesn't happen, the ID of the latest message delivered to the Client is updated (within **ClientStatus**) and the message is delivered to them.

C. Fault Tolerance

To test and assure fault tolerance in case of a client or server crash, the group decided to save data (using functions of prefix *save*) that is created in execution time: list of topics and their subscribers in CSV files. These were manipulated using Pandas.
The information in these files is loaded to the program (using function *load()*) every time it starts, serving as a baseline for every time the program is executed.
This process makes sure that no data is lost should a crash occur and allows all services to recover easily.

1) Exactly-once

Exactly-once delivery is guaranteed in two ways:

- On a successful return from *put()* on a Topic, the service guarantees the message will eventually be delivered to "all subscribers on that topic", as long as the subscribers keep calling *get()*;
  - As soon as the Server receives a message, the message will be saved with an unique ID in stable storage. Therefore, a subscriber calling *get()* will be able to go through all messages of a Topic until the received put message is consumed;
- On a successful return from *get()* in Topic, the service guarantees that the same message will not be returned again on a later call to *get()* by the subscriber;
  - As soon as the Client gets a message yielded by *get()*, said occurrence will be stored in stable storage by saving the received message number. Therefore, the Server will never yield a certain message to a Client a second time, since it won't ask for it;

The only possible scenario that might make this approach fail is when a crash occurs only after the message is received yet before it's stored. To make this as unlikely as possible, the implementation stores data as soon as possible after a message being received.

IV. Conclusion

Throughout work for this project, it was possible to learn how to effectively use and apply ZeroMQ in the development of a distributed system.
The abstractions provided by ZeroMQ are incredibly powerful and simplify a lot of the work that would have been required otherwise. As such, the work's efforts were redirected to the implementation of protocols and our already covered strategies, namely for the operations and our fault tolerance measures.
The final result is a simple yet powerful system that abides by the provided specifications and operations.

References

[1] Pieter Hintjens. Chapter 4 - reliable request-reply patterns. https://zguide.zeromq.org/docs/chapter4/. [Online; accessed 09-10-2022].

Appendix

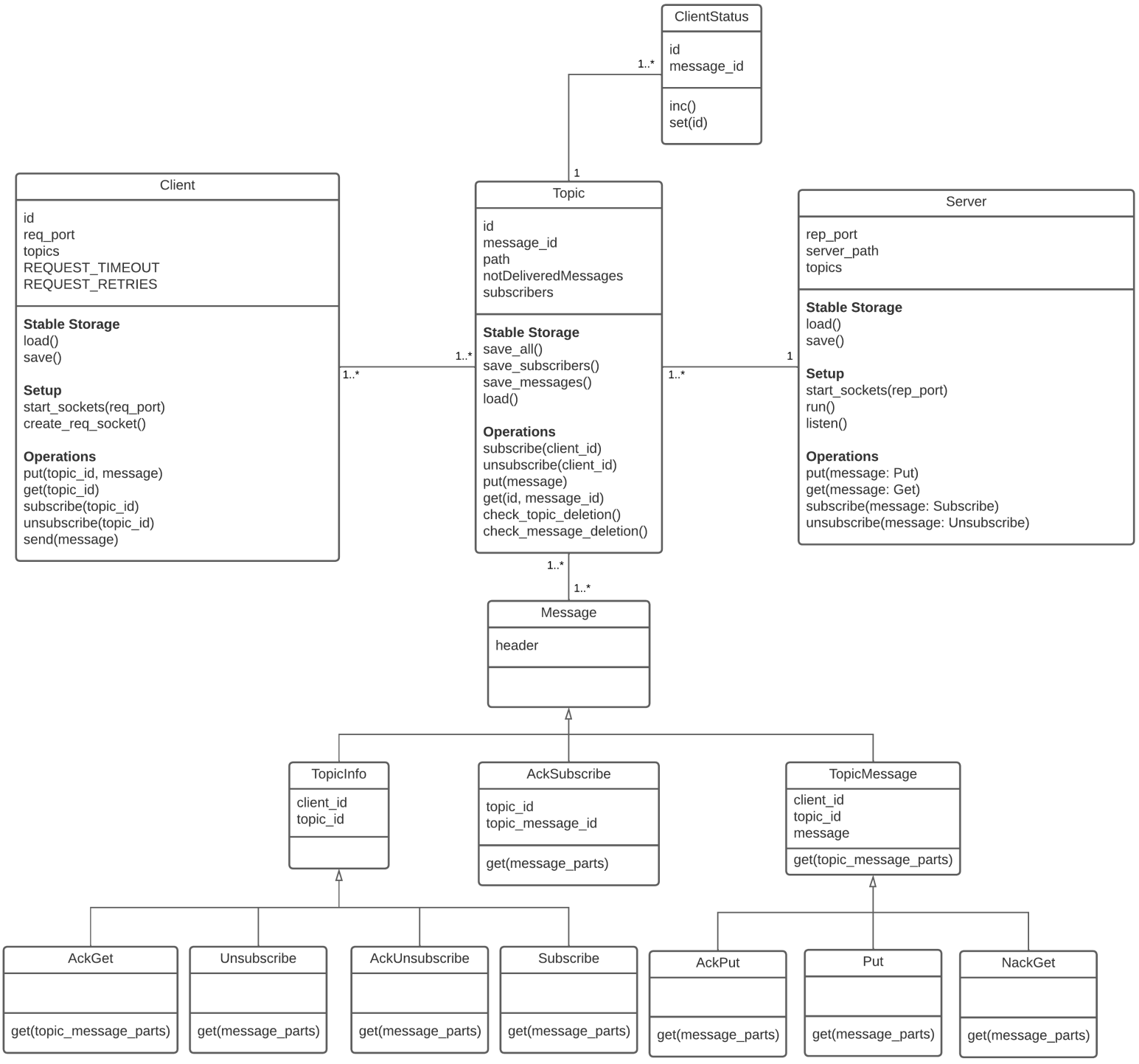

Fig. 2: Implementation API UML Diagram.