Reliable Pub/Sub Messaging Service

1st project of the Large Scale Distributed Systems course

Master in Informatics and Computing Engineering

Beatriz Santos, (up201906888@edu.fe.up.pt)
João Barra, (up201808910@edu.fe.up.pt)
Maria Carneiro, (up201907726@edu.fe.up.pt)
Sérgio da Gama, (up201906690@edu.fe.up.pt)

Class 5, Group 12

2022/23

Contents

1 Introduction

The objective of this work is to develop a reliable publisher/subscriber service for asynchronous communication in a distributed system. This system consists of a set of distinct entities that communicate with each other by exchanging messages, considering that entities may not be available simultaneously at all times. This fact arises some relevant difficulties to ensure reliability, namely in the fault-tolerance and in the order of delivery of messages. To handle the 'unavailability' of this scenario, there was a need to save the messages in a queue until the receiver is available to receive them. The work was developed in Python.

2 Messaging Service

Throughout the development of the messaging system, we took advantage of the *ZMQ framework* sockets of type *ZMQ.REQ* and *ZMQ.REP*, which are the implementations of the Request-Reply Pattern. This type of socket allows more flexibility over the design of our solution, as they represent the simplest way of communication, the client/server model.

We choose to make a ”parent” class (Client) that is divided into two types:

- Publisher: Has only the ability to publish a message on an existing topic.
- Subscriber: Has the ability to subscribe to a topic (even create it, if this one doesn't exist), unsubscribe to the same topic, or get the messages published to that topic.

2.1 Messages format

As we know, the server and the client communicate by exchanging messages. In order to fulfil the client's requests, the server has to receive useful information and vice-versa.

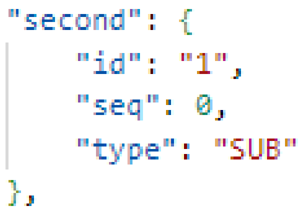
The format chosen for the messages is a **Python dictionary**.

2.1.1 Requests Format

Requests to the server can be made from two clients with different purposes. For this reason, the message that the server receives differs if the client is a Publisher or Subscriber and has the following fields:

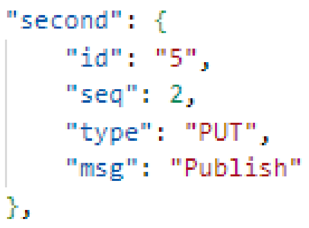
Topic Name: (***holds the name of the topic***)
{
'id' (***id of the client***)
'seq', (***sequence number to maintain synchronisation and keep track of the messages***)
'type', (***type of request. Can only be one of the following: SUB, UNSUB, GET OR PUT***)
'msg', (***only used in the Publisher-Server communication and holds the message that is going to be published in the topic***)
}

The Request messages's Figure below contains an example of what a message from a Subscriber and a Publisher might look like.



```
"second": {
    "id": "1",
    "seq": 0,
    "type": "SUB"
},
```

(a) Subscriber



```
"second": {
    "id": "5",
    "seq": 2,
    "type": "PUT",
    "msg": "Publish"
},
```

(b) Publisher

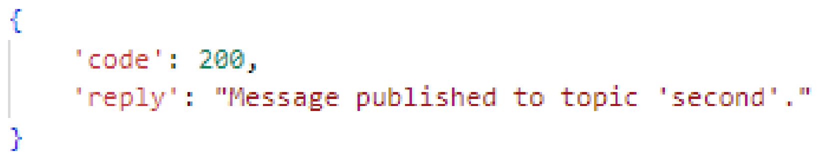
Figure 1: Request messages

2.1.2 Responses Format

The server replies to the two different clients with the same message fields:

{
'code', (***Code number that indicates if the request was successful or not***)
'reply', (***Message for the user to know what happened***)
}

The Reply message's Figure below contains an example of what a message from a Server might look like.



```
{
    'code': 200,
    'reply': "Message published to topic 'second'."
}
```

Figure 2: Reply message

2.2 Operations

2.2.1 Put

A publisher can put a certain message in a specific topic by running the following command:

```
Publisher.py <port> <id> <topic> <message>
```

Since the publisher's only action is `PUT`, there is no need to specify it as an argument.

After the publisher instance is created, the client sends a `PUT` request to the server, in the Requests Format, with the message clearly associated to the topic it wants to publish in.

The server gets this request and parses it to guarantee that the message is stored correctly in its topics object. The sequence number received represents the amount of messages the client has already sent to the server to process, related to the topic specified. There are three possible outcomes when publishing a message:

- `404`: The topic the publisher wants to publish in hasn't been created by any subscriber, so the message isn't published.
- `500`: The topic the publisher wants to publish exists and the client have already published on that topic, but the sequence numbers from the server and the client don't match, implying some internal server error occurred, so the message isn't published since it could break the 'exactly-once' guarantee (further explanation in section Tradeoffs)
- `200`: The topic the publisher wants to publish exists, the client have already published on that topic and their sequence numbers match, so the message is published.

The topic object is then updated with all the new information and the response codes and acknowledgement messages are sent back to the publisher. After that, the sequence number is increased on the server's side and the server's files, containing all the topics, are updated with the new information side using Persistent storage.

The client receives the reply message, and, after processing it, if the message was published, the sequence number of the client is increased, to state that one more message was processed. After acknowledging that, this information is updated on the client's side using Persistent storage.

2.2.2 Get

A subscriber can get a certain message for a given topic by running the following command:

```
Subscriber.py <port> <id> GET <topic>
```

After the subscriber instance is created, the client sends a `GET` request to the server, in the Requests Format.

The server gets this request and parses it to guarantee that the message is retrieved correctly. The sequence number represents the amount of messages the client has already consumed and that they received. There are five possible outcomes when getting a message from the topics queue:

- `404`: The topic the subscriber wants to get a message from doesn't exist, so no message is retrieved.
- `403`: The topic the subscriber wants to get a message from exists, but the subscriber isn't subscribed to that topic, so it can receive any messages that belong to it.
- `500`: The topic the subscriber wants to get a message from exists and the client is subscribed to that topic, but the sequence numbers from the server and the client don't match, implying some internal server error occurred, so the message isn't retrieved since it could break the 'exactly-once' guarantee (further explanation in section Tradeoffs)
- `204`: The topic the subscriber wants to get a message from exists, the client is subscribed to that topic and the sequence numbers of the server and the client match, but the amount of messages from that topic is equal or inferior to the sequence number, meaning that the subscriber has already received all the messages for that topic and that there are no more new messages for them to receive.
- `200`: The topic the subscriber wants to get a message from exists, the client is subscribed to that topic, the sequence numbers of the server and the client match and the amount of messages from that topic is superior to the sequence number, meaning that there are new messages that the subscriber can receive.

The topic object is then updated with all the new information and the response codes and acknowledgement messages are sent back to the subscriber. After that, the sequence number is increased on the server's side and the server's files, containing all the topics, are updated with the new information side using Persistent storage.

The client receives the reply message, and, after processing it, if the get operation was successful (200 or 403), their internal logs are updated with the new sequence number.

2.2.3 Sub

A subscriber can subscribe to a certain topic of messages by running the following command:

```
Subscriber.py <port> <id> SUB <topic>
```

After the subscriber instance is issued, this client sends a `SUB` request to the server, in the Requests Format, specifying the topic it wants to subscribe to.

The server receives this request and parses it to guarantee that the message is retrieved correctly and handles it accordingly. There are two possible scenarios when subscribing to a topic queue:

- `200`: The topic the subscriber wants to subscribe doesn't exist and, thus, a topic is created in the server's topic list, the client becomes a subscriber, and a list of messages and a list of publishers associated with that specific topic are created. Or in case the topic existed and the client isn't subscribed to that topic, the client becomes a subscriber to that topic.

- `304`: The subscriber wants to subscribe to a topic already subscribed by himself. In this scenario, nothing is modified.

The topic object is then updated with all the new information and the response codes and acknowledgement messages are sent back to the publisher. Then, the subscriber client receives a response with the server's status. If the operation was successful, the subscriber's topic list is updated with the topic subscribed and the subscriber's id. And on the side of the server, the client is added as a subscriber to the topic. Finally, the server's and client's files are updated with the new information side using Persistent storage.

2.2.4 Unsub

A subscriber can unsubscribe to a certain topic of messages by running the following command (similar to the `SUB` operation command):

```
Subscriber.py <port> <id> UNSUB <topic>
```

When the subscriber instance is created, the client sends a `UNSUB` request to the server, in the Requests Format, specifying the topic it wants to unsubscribe to.

The server receives this request and parses it to guarantee that the message is retrieved correctly and handles it accordingly. There are three possible scenarios when unsubscribing to a topic queue:

- `404`: The subscriber wants to unsubscribe from a topic that doesn't exist. In this scenario, nothing is modified.
- `403`: The subscriber wants to unsubscribe from a topic that exists but that wasn't subscribed to by him. Nothing is modified.
- `200`: The topic the subscriber wants to unsubscribe from exists, and the client is a subscriber of it. The client is popped out of the topic's subscribers list (if the list of subscribers becomes empty the topic is popped out of the list too).

The topic object is then updated with all the new information and the response codes and acknowledgement messages are sent back to the publisher. Afterwards, the subscriber client receives the response and processes it. If successful (error code 200), it updates the subscriber's information, if not (error codes 403 or 404), it only prints information about what happened. On the server's side, the topic's subscriber list is updated. Finally, the server's and client's files are updated with the new information side using Persistent storage.

3 Fault-tolerance

3.1 Threads

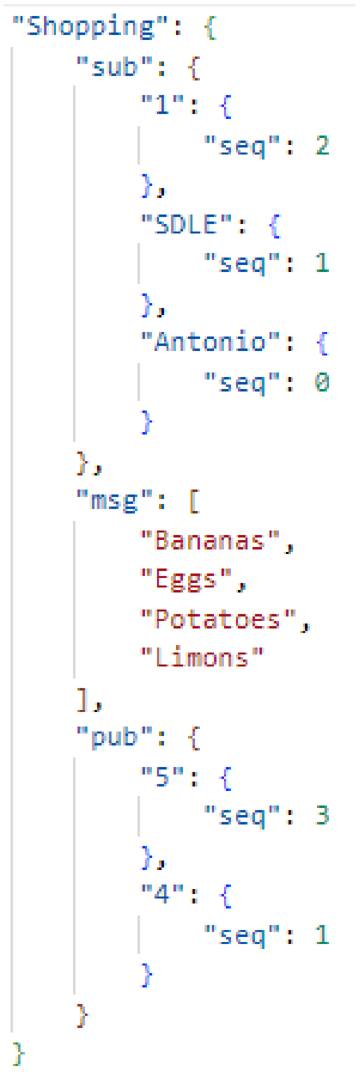
The server is working as a 'middleware' between the entities involved, namely the clients (Subscribers and Publishers), recording all the info related to them. So, to manage all the requests made to the server by the clients, the main server thread listens to requests, delegating jobs to several threads. Each thread is responsible to handle one of the server's operations (`PUT`, `GET`, `SUB` and `UNSUB`). The main thread, then, waits for the completion of the threads, sending the generated responses back to each one of the clients issuing the requests.

3.2 Persistent storage

Persistent storage is any data storage device that retains data once the power source to it is switched off [2]. In this way, in order to guarantee that the server does not lose the information related to the topics, such as the subscribers, the messages published, etc., the received information is stored in a *JSON* file after the received information has been processed.

This information is stored per topic. This was done in order to make it easier to search for topics, each one containing the respective Subscribers and Publishers, as well as the client's sequence number associated with the topic, and it also contains, of course, the published messages.

The Figure 3 bellow is an example of the information that a topic can store.



```
"Shopping": {
    "sub": {
        "1": {
            "seq": 2
        },
        "SDLE": {
            "seq": 1
        },
        "Antonio": {
            "seq": 0
        }
    },
    "msg": [
        "Bananas",
        "Eggs",
        "Potatoes",
        "Limons"
    ],
    "pub": {
        "5": {
            "seq": 3
        },
        "4": {
            "seq": 1
        }
    }
}
```

Figure 3: Topic storage

In order to know the last action performed by a client, the message that they send to the server is also saved in a *JSON* file with the format seen in Request messages's Figure.

3.3 Polling

In order to have another layer of fault tolerance, it was implemented the Lazy Pirate Pirate Pattern [3]. Indeed, without this mechanism, a server crash would permanently break the clients that were waiting for this first one to respond. This situation happened, because of the nature of the REQ/REP socket used, which blocks until input is, in fact, received.

Therefore, aiming to prevent this client's blockage, it was implemented said pattern, which uses polling, with a given timeout, to check if the server is still alive or not. If it verifies that the server is offline, it retries to send the desired request and polls for data with the same timeout. This process is then repeated a fixed number of times, which is defined in our code by the macro *REQUEST_RETRIES* to 3, in the *client.py* file.n The timeout is 2.5 seconds, which is also defined in the same file, but in the macro *REQUEST_TIMEOUT*.

3.4 Tradeoffs

As mentioned before, the case where sequence numbers between the server and the client don't match (500), represents a trade-off we had to consider when developing our system. Since the cause for such an error would imply that the sequence number was either altered on the server side or on the client side, that would mean that the "exactly-once" principle was compromised. We chose to maintain that principle at all costs, prioritizing reliability instead of availability, so when such an error occurs, our system doesn't resend the request (for example, a subscriber making a `GET` request that has mismatched sequence numbers, can only unsubscribe from that topic later on, since *get* and *subscribe* operations require equality of sequence numbers). More details about the corner cases that might cause this can be found in section Rare Circumstances.

4 Exactly-once

4.1 Guarantees

In the development of the messaging system we prioritize this property over the rest. That said, we guarantee exactly-once with the implementation of a sequence number in all the methods that work with actual published messages, *get* and *put*, similar to what is used in Apache Kafka [1].

This sequence number states until what message, from the server queue, a given subscriber has already consumed, in the case of *GET* request. On the other hand, when a publisher makes a put request, its sequence number states how many messages the publisher has already published. When the server receives one of these requests it updates the sequence number of the respective client in order to validate subsequent requests made by that same client.

Thus, if the publisher tries to publish, or the subscriber tries to get, the same message twice, they won't be able to, because the sequence number the server is expecting from them, will be different. Guaranteeing that the subscriber is only able to get a particular message once.

4.2 Rare Circumstances

This chapter describes some of the rare circumstances that might eradicate our messaging system and break the exactly-once paradigm. All the cases below can be verified in the *scripts/error_scenarios* folder, where there is a script representing each one of said cases.

4.2.1 Server Crash

One of these cases is when the server crashes after receiving and processing a request, but before saving the data into files, persistently. This will put the system in an invalid state, because, when it goes back online it will accept, for example, subscription requests that were already made, or even worse, get and put requests that were already made. Hence, by accepting these duplicated requests, it is no longer guaranteed the exactly-once property.

4.2.2 Data Integrity

In fact, we take data integrity for granted, but files stay many times corrupted and it is not despicable at all. This data corruption can lead to the alteration of the sequence number, in any of the points of communication, the server or the client. Therefore, when this happens, the subscriber is no longer able to receive requests, because there is no way the server can guarantee the exactly-once property. For example, if a subscriber gets the message with sequence number 3 and then its sequence number is corrupted and tries to get the message with the sequence number 2, the server can't deliver it, because the client has already received it. When this situation occurs to the subscriber, it can only perform the unsubscribe operation, which will reset their sequence number.

4.2.3 CPU Overload

Finally, when it is performed a bad load balance, the server might end up in a CPU overload situation. When this happens after receiving a request, but before sending the reply, the server will stay in an invalid state. This happens when the client waiting time for the server's response is smaller than the time the server takes to start sending the reply and the server will stay permanently blocked in that incomplete send.

5 Conclusion

During the development of the project we were able to learn more about many concepts related to large-scale distributed systems, such as messaging-oriented middleware, client/server processing and pub/sub design patterns, as well as discovering and using a powerful library, *ZeroMQ*. Understanding and exploring the challenges in building a simple pub/sub system made us aware of the complexity and efficiency of many of the messaging services we use in our daily lives. Overall, this project had a positive impact on us as it allowed us to further deepen our knowledge to hopefully become better engineers.

6 References

[1] Confluent. Exactly-once semantics are possible: Here's how kafka does it. Confluent, available at https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/, 2022. last accessed on october 2022.

[2] netapp. What is persistent storage? NetApp, available at https://www.netapp.com/data-management/max-memory-accelerated-data/persistent-storage/, 2022. last accessed on october 2022.

[3] ZMQ. Client-side reliability (lazy pirate pattern). ZMQ Guide, available at https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern, 2022. last accessed on october 2022.