SDLE 1st Assignment - Reliable Pub/Sub Service

Afonso Monteiro - up201907284@edu.up.pt
Miguel Freitas - up201906159@edu.fe.up.pt
Joana Mesquita - up201907878@edu.fe.up.pt
Carolina Figueira - up201906845@edu.fe.up.pt

October 23, 2022

Abstract

This report aims to illustrate the work done in order to design and implement a reliable publish-subscribe service application, therefore furthering the knowledge in Large Scale Distributed Systems.

The project consists solely on the development of a publish-subscribe application that contains 4 operations:

- put() - to publish a message on a topic;
- get() - to consume a message from a topic;
- subscribe() - to subscribe a topic;
- unsubscribe() - to unsubscribe a topic;

To clarify, messages are arbitrary byte sequences and topics are identified by an arbitrary string.

Contents

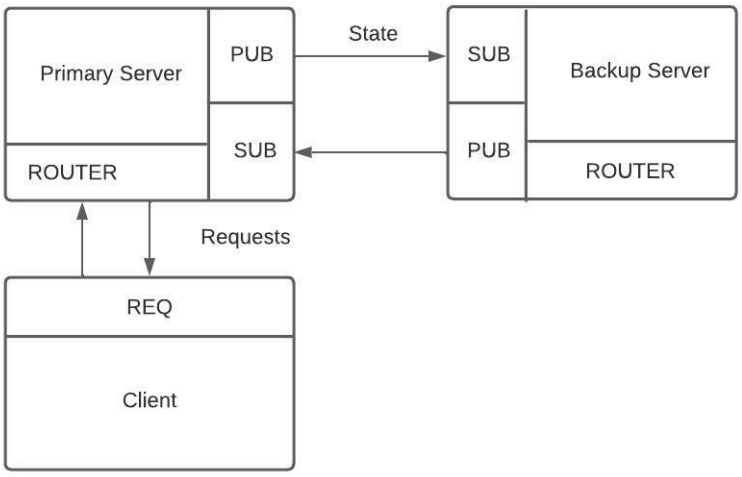

Figure 1: Scheme of our publish subscribe service

1 Application design

The application consists of a publish-subscribe service implementing a Binary Star Pattern meaning that not one but two servers are used in order to accept requests and avoid fail states.

Both servers actively poll for messages from clients on their respective ROUTER sockets and the clients send requests via REQ socket. Initially the servers also had a REP sockets, however this meant that they had to process each individual client request completely before processing the next which was quite inefficient.

1.1 Client reliability

Client reliability is defined on the lazy_pirate.py file. Essentially, all clients always know the PRIMARY and BACKUP servers' endpoints. Upon being called by the test_client to perform a request, a client will first attempt to send that request to the PRIMARY server and poll for an answer for a specified time duration. Should it not receive an answer in that time it will print a warning to the console informing that it could not connect to the PRIMARY server and it will try its luck with the BACKUP server:

```
[WARNING]: no response from server, failing over
```

If the BACKUP server is silent as well, the server tries sending the request to the PRIMARY server again. It continues alternating between the two servers until one of them replies back. The file is named lazy_pirate.py

because we this behaviour is a modified lazy pirate pattern described in the Z Guide[2]

1.2 Data permanence

To guarantee data permanence a server starts a thread that writes to permanent memory every five seconds if the server is currently the active one in the pair. This location is currently hard-coded as **topics.json** . Topic info is stored in the following format:

```
1    {"topics": [{"topicName": "1111", "subscribers": [], "
     messages": [
2 {"messageID": "ea17a9247a5c4749a6339670d4f08ab0", "
     messageContent": "1111",
3 "subscribersToRead": []},
4  {"messageID": "1dc6ebf4f9184c208b041aee6da39604", "
     messageContent": "1111",
5  "subscribersToRead": []},
6 {"messageID": "85a2722a0d924140a3a86ae3c6492ee0", "
     messageContent": "1111",
7 "subscribersToRead": []}]}
8 ]
9 }
```

In which subscribersToRead is always a subset of that message's topic's subscribers, representing what subscribers have not yet called GET on that message. Older messages are always at the beginning of the list and the unique message id is generated on the server side.

1.3 Server reliability

Server reliability is assured by the aforementioned Binary Star Pattern .[1]

In the case of server malfunction, we assure the existence of two different endpoints: one PRIMARY and one BACKUP. This pattern ensures that, should the PRIMARY server fail, the BACKUP server takes over while it is unavailable.

The state machine in figure 2 is an accurate description of how the servers work. The following terminology is required to understand the diagram:

- **Primary:** The server that is normally or initially active.
- **Backup:** The server that is normally passive. It will become active if and when the primary server disappears from the network, and when client applications ask the backup server to connect.
- **Active:** The server that accepts client connections. There is at most one active server.
- **Passive:** The server that takes over if the active disappears. Note that when a Binary Star pair is running normally, the primary server

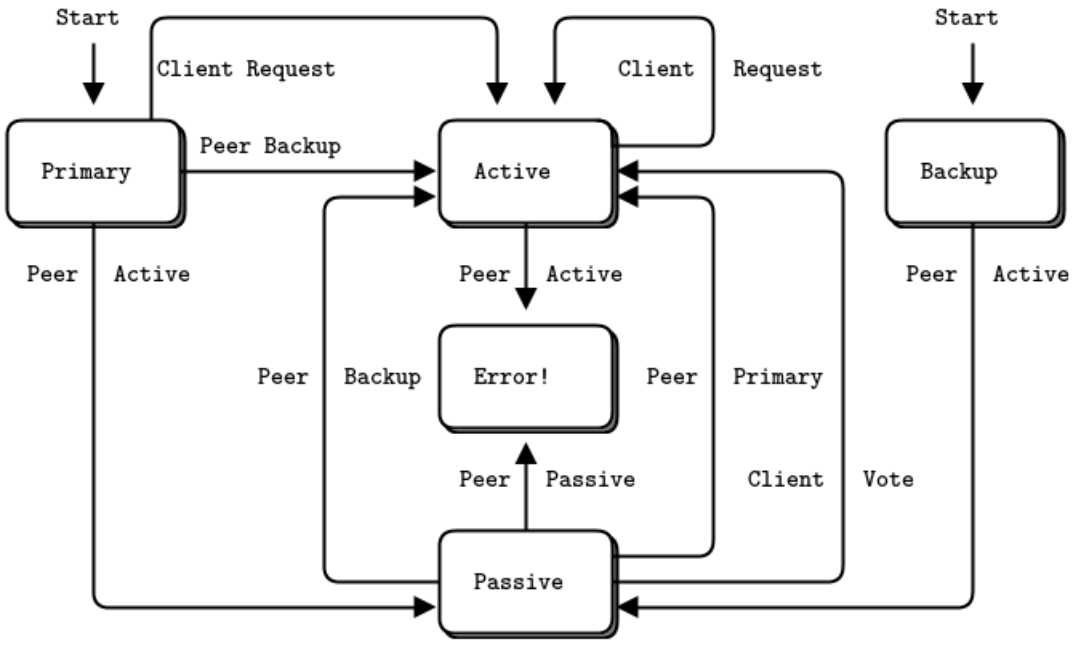

Figure 2: Binary Star state machine

is active, and the backup is passive. When a failover has happened, the roles are switched.

Due to the fact that clients have no awareness of which server is currently active, they will always try to send their request first to the PRIMARY server before failing over to the BACKUP. Because ZMQ's ROUTER sockets queue messages even if they're not received immediately, once the PRIMARY server is restored it will process all requests it "missed".

Meanwhile once the BACKUP server receives its first request from the client, signaling that the primary server is down and that it is now the active server, it will read the current topics' information from the **topics.json** file.

1.4 How servers keep track of each other - HEARTBEAT

Heartbeating was used in order for the servers to be able to keep track of which one of them is the current active one, using their respective PUB and SUB sockets to keep track of their peer's status. However, servers must be cautious not to overdo these messages since this could cause them to take over for the other, when the it just needed some time to recover.

This also helps prevent Split-brain syndrome, which occurs when different parts of a cluster think they are active at the same time.

1.5 Failover/recovery

We could have chosen to implement the Binary Star pattern in such way that once the PRIMARY server is restored, the BACKUP becomes passive again. However, we chose to follow ZMQ's recommendations and make it so that you must also reset the BACKUP server, otherwise you get a BStarState exception.

This choice was made because as we understood it is good a practice for the problem at hand. If this was a more complex application, once a server went down for any reason, you'd want recovery to be a manual operation, since if it was automatic this would create uncertainty for system administrators, who can no longer be sure which server is in charge without double-checking, at a time when the absolute first priority should be certainty so the issue that caused the crash can be identified and fixed.[1]

1.5.1 Proper way to trigger recovery

- The operators restart the primary server and fix whatever problems were causing it to disappear from the network.
- The operators stop the backup server at a moment when it will cause minimal disruption to applications.
- When applications have reconnected to the primary server, the operators restart the backup server.

1.6 Some trade-offs of the Binary Star pattern

- **Use of many resources:** Having a backup server that's taking up resources while not doing anything most of the time can be costly.
- **Only one backup :** This pattern allows for only one backup server.

1.7 Possible breach of the "exactly once" principle

Because the PRIMARY server will only write to permanent memory every five seconds (for efficiency's sake) and the only way the BACKUP server has to keep up with state is that very same file, there is always the possibility that the PRIMARY server processes some requests from clients and updates its internal state accordingly but dies before actually updating the file. The BACKUP thus takes over and uses a stale state to process the following requests. However, once the PRIMARY server is fixed and takes over again, because of ZMQ's queues, the state should once again be up to date.

This issue could be "fixed" by having the two servers writing the topics information directly to the same database instead of storing the state internally and then permanently every X seconds. The server could still die

before performing said write in this situation, but this would make it less likely.

2 Operations

As stated in the abstract, the application revolves around four main operations.

2.1 PUT

```
PUT <topic_name> <message>
```

The put operation, represented by the PUT_TOPIC function, takes the message and attempts to publish it on the given topic, passing through the broker so it can keep track of the operations made, using the handle_PUT function.

2.2 GET

```
GET FROM <client_port> ON TOPIC : <topic_name>
```

The get operation takes the topic to retrieve information from as a parameter. It starts by checking if the client doing the request is subscribed to that topic, exiting the call if it is not. If the client is subscribed, it will try to send the request message, also through the broker using the handle_GET function.

2.3 SUBSCRIBE

```
SUBSCRIBE <topic_name> FROM SUBSCRIBER <client_port>
```

The topic subscription is done with the subscribe_a_topic function on the client.py file. Using the topic as parameter, it will attempt to subscribe a client to a topic through the broker (handle_SUBSCRIBE function). If it is successful, it will update the subscribed topics of that client.

2.4 UNSUBSCRIBE

```
UNSUBSCRIBE <topic_name> FROM SUBSCRIBER <client_port>
```

Unsubscribing from a topic is done with the unsubscribe_a_topic function on the client.py file. Using the topic as parameter, it will attempt to unsubscribe a client to a topic through the broker (handle_UNSUBSCRIBE function). If it is successful, it will update the subscribed topics of that client.

As it preforms the operations listed, the broker is also saves the information by writing to a file, to maintain reliability.

2.5 BROKER RESPONSES

To guarantee that messages aren't lost somewhere between the client and broker some messages of acknowledgement are sent form the broker to the server.

A successful response from the server will look like:

```
ACK [SUCCESS]: <operation_name> is DONE
```

While an unsuccessful one will look like such:

```
ACK [FAILURE]: <failure_message>
```

3 Conclusion

In conclusion, we believe that we were successful in our implementation of the publish-subscribe service application, on top of a ZeroMQ library.

It gave us the opportunity to better understand the polling mechanism and the challenges presented when dealing with multi-threading programming, as well as different possible solutions and its trade-offs.

Therefore, we feel confident on the product developed and are ready to tackle the next challenge.

Bibliography

[1] ZMQ. *High-Availability Pair (Binary Star Pattern)*. URL: https://zguide.zeromq.org/docs/chapter4/#High-Availability-Pair-Binary-Star-Pattern.

[2] ZMQ. *Reliable Request Reply Patterns*. URL: https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern.