Reliable Publish-Subscribe Service

SDLE Project 2021/2022

This project was developed by T4-G17 from the MEIC course in FEUP:

Carlos Lousada (up201806302@up.pt)
José Maçães (up201806622@up.pt)
Mariana Ramos (up201806869@up.pt)
Tomás Mendes (up201806522@up.pt)

Table of Contents

Overview and project goals

The main goal of this report is to describe the design and implementation of our Reliable Publish-Subscribe Service.

All the required operations are supported by our service:

- put() - A Publisher publishes a message on a topic
- get() - A Subscriber consumes a message from a topic
- subscribe() - A Subscriber subscribes to a topic
- unsubscribe() - A Subscriber unsubscribes from a topic

Additionally, we implemented a new operation to see all the topics a subscriber has subscribed to. The usage of these operations are described in the README.md file for testing purposes.

Furthermore, **"Exactly-once"** delivery is guaranteed in the presence of communication failures or process crashes, using the Request-Reply pattern to exchange Acknowledgement messages and

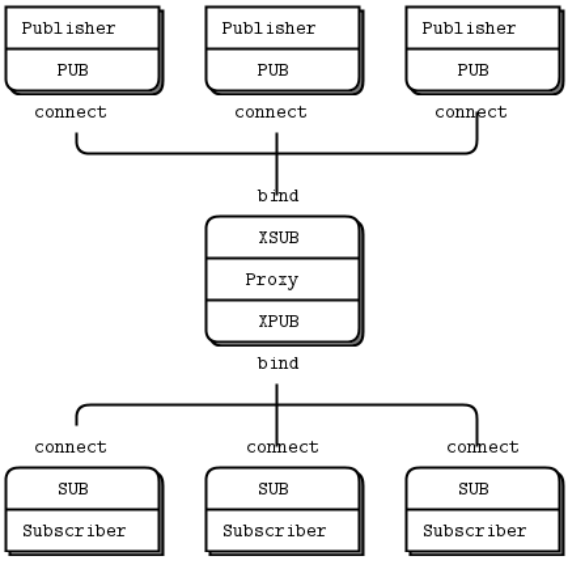
lost data. We also handled the multiple sockets using the **zmq_poller()** and made sure to register which messages the subscribers had already received, avoiding duplicated data. Additionally, we implemented persistent **storage** that allows the program to recover even if the intermediary server (proxy) fails. All of these features are explained in detail in the following sections along with the communication patterns used.

Lastly, since it was not imposed any programming language to develop this project, we chose Java and used its techonology of Remote Method Invocation (RMI).

ZeroMQ

Pub-Sub Network with a Proxy - *XPUB/XSUB*

One of the requirements of this project was to use the **ZeroMQ** library, which already implements a quality solution for the Publish-Subscribe problem, connecting a set of publishers to a set of subscribers and facilitating data distribution.
In order to allow multiple publishers and subscribers, we used a pattern that included an intermediary: a static point in the network to which all other nodes connect.
That being said, this was our base architecture:



However, this does not guarantee **exactly-once** delivery.

Request-Reply pattern - *REQ/REP*

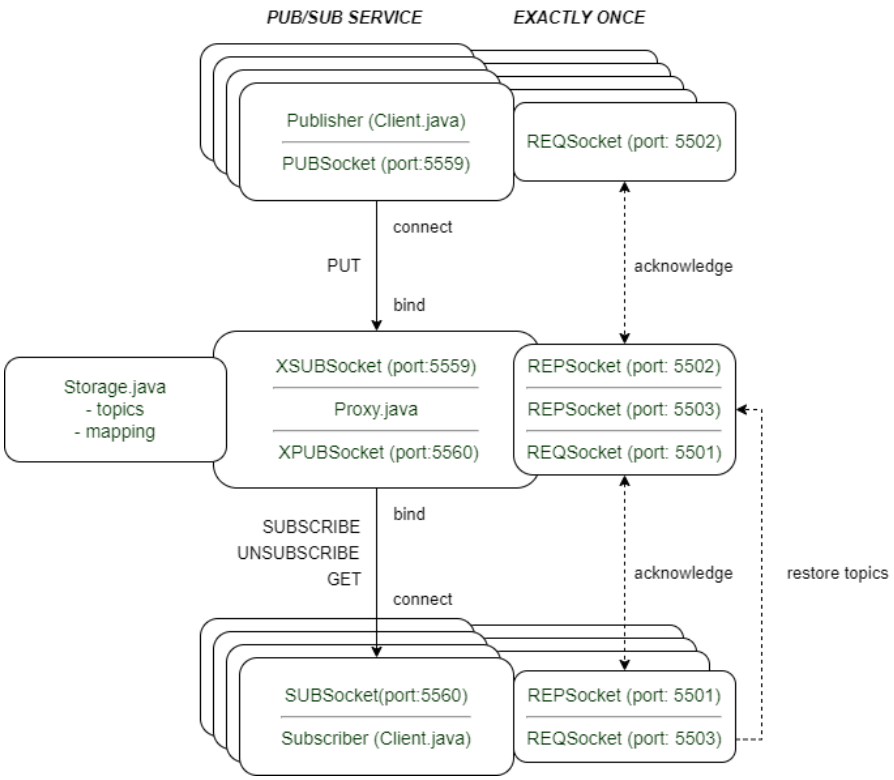
In the presence of communication failures or process crashes, there was no guarantee that the messages were already delivered or would eventually be delivered. By implementing the **Request-Reply** pattern, we opened new connections that allowed acknowledgement messages to be exchanged between the server and its clients. We also used this pattern to recover potentially lost data on client crash (such as the topics that he had subscribed), information which is stored in the server.

ZMQ_Poll

In order to read from multiple sockets all at once and deal with the blocking aspect of these patterns, we used **zmq_poll()**. This way, we poll for activity on every socket. We also use a **periodical handler** for message synchronization (especially in *REQ/REP* communication), implementing timeouts (default 5000ms), and number of tries (default 3 tries).

Architecture design and implementation aspects

The image below summarises the architecture implemented.



Proxy and Connections

The proxy is an intermediary that allows for several Publishers and Subscribers to communicate. It stores all the information using a nested class **Storage.java**. This class contains two *ConcurrentHashMaps*: *"topics"* with all the topics and corresponding messages, and *"mapping"*, containing the subscribers' ids and an index that indicates what was the last message that the subscriber received (using get) on a specific topic.

As described in the previous section, the Proxy implements two main sockets: the **XSUB**, to receive all messages from the publishers (put operation) and **XPUB** to forward those messages to the Subscribers (get operation) according to their subscriptions.

The server (proxy) starts by binding every used socket to its respective address. Afterward, the clients (publishers and subscribers) can start connecting to the already bound sockets. Each service then creates a *ZMQ-poller*, allowing them to read data from multiple sockets all at once in an efficient manner.

Additionally, we also implemented three **Request/Reply** sockets in the Proxy that are mainly used for confirmation messages (crash detection) and data retransmission (previously subscribed topics).

The socket in port 5501 is used by the Subscribers to confirm to the Proxy that they received the message requested by a get operation correctly. On the other hand, the socket in port 5502 allows the Proxy to confirm to the Publishers that it processed their messages correctly after a put operation, and will deliver them to the clients that are subscribed to that topic. Finally, the socket in port 5503 was created when the need to restore previously subscribed topics arose. This way, when a subscriber crashes or shuts down, if it ends up being restored, it will send a request asking the server to send the topics that it had previously subscribed to.

These messages are explained in detail in the Communication Protocols section.

Publisher and Subscriber

The Publisher contains one *PUB* socket, to communicate with the Proxy, more precisely to send the **PUT** message and publish new messages. The *REQ* socket, as already described, is used for dealing with acknowledgment messages.

The subscriber has two lists. One for all the subscribed topics and one for the identifiers of the messages already received. Similar to the Publisher, it uses the *SUB* socket, as part of the basic PUB/SUB implementation to **SUBSCRIBE** and **UNSUBSCRIBE** to topics as well as **GET** messages from subscribed topics. The
REP socket is used to deal with acknowledgment messages and, finally, the *REQ* socket, is used to request the proxy for previously subscribed topics in the eventuality of a crash.

Communication Protocols

The *libzmq* core library has already implemented two APIs to deal with message exchanges. We used the *ZFrame* class that provides methods to process messages sent by a *SUB*.subscribe and *SUB*.unsubscribe, such as *ZFrame.recvFrame()* and *frame.getData()*. A 'frame' corresponds to one underlying *zmq_msg_t* in the *libzmq* code. All the data is processed in Strings that are converted to byte arrays.

Communication between Publisher and Proxy

The communication between the Publisher and the Proxy allows for publishers to publish - **put()** - messages on topics.

First, the publisher sends the message frame *"<topic>//<message>"* to the proxy using the *PUB* Socket.
Afterward, it sends another message *"<pubId>"*, through the *REQ* socket, asking for a confirmation that the message was added correctly. Using a zmq_poll(), it waits for the reply of the Proxy that, if everything works fine, will re-send *"<pubId>"* through the *REP* Socket.

Communication between Subscriber and Proxy

The Subscriber and the Proxy communicate so that subscribers **subscribe** and **unsubscribe** to topics, and consume - using **get()** - messages from subscribed topics.

- Subscribe

The subscriber starts by sending a subscribe ZFrame through the *SUB* Socket with the following content: *"<subID>//<topic>"*. This operation uses the method subscribe().
After this, the Proxy sends the message *"<subId>//<topic>//1"* through the *XPUB* Socket confirming the subscription was successful.

- Unsubscribe

Similar to the subscribe operation, the subscriber sends an unsubscribe ZFrame through the *SUB* Socket with the following format *"<subID>//<topic>"*.

- Get

This operation runs a little differently. Since we wanted to take the maximum advantage of the *PUB/SUB* pattern provided by Zeromq, we decided to use the *SUB* socket, but with a 'get' flag. This way the subscriber sends a subscribe ZFrame similar to the one used in the Subscribe operation, but with *"//get"* in the end.
The Proxy then sends *"<subId>//<topic>//<message>"* through the *XPUB* socket.
After that comes the confirmation: the proxy, using the *REQ* socket, sends *"subId"*. And the Subscriber will reply re-sending *"subId"* using the REP socket.

Further Reliability

Storage

We had into consideration the possibility of the **proxy failing**.

In order to save on disk all the topics, messages, and subscribers, we implemented a **Storage class**. This class contains two data structures with the information mentioned and implements methods that are then responsible to save them in the file *"storage/data.ser"*. At first, we thought of calling these methods every ten seconds. However, upon further thought, we decided to also write to the file after every subscriber operation, guaranteeing the proper fault tolerance.

When the proxy restarts, it reads its previous state from the file and no information is lost.

Restore topics

Additionaly, we had into account the possibility of a **subscriber failing** for an indefinite period.

Every time a subscriber is initialized it connects to a *REQ* socket already bounded by the proxy. Through this socket, the subscriber sends a request message to the proxy asking for the topics that it had already subscribed to. This message only contains the id of the subscriber.

Subsequently, the proxy will reply, using the *REP* socket, with a list of all the topics the subscriber had subscribed to. This list can be empty.

The subscriber will read the reply and do a local **subscribe** operation to every element of the list.

However, unlike a normal **subscribe**, this operation will not reset the index that points to the last message read by that subscriber (using get).

Garbage Collection

Finally, we implemented a **garbage collection** method that allows us to have a more efficient program.
This way, our garbage collection implementation runs in a different thread every 10 seconds. It verifies if a certain topic has no subscribers left and deletes it in afirmative case. Furthermore, removes messages that were already consumed by all the subscribers of a specific topic (updating **topics** - Concurrent HashMap) and also modifies the index that indicates what was the last message that the subscriber received (using get) on a specific topic (updating **mapping** - Concurrent HashMap).

Conclusion

In conclusion, we were able to fulfill all the requirements specified and think of new solutions for the problem at hand. In a single server implementation of the service, we were able to make all the operations work as expected, even in the presence of faults in the server or clients, guaranteeing durable subscriptions.

Furthermore, we used several *ZeroMQ* patterns and APIs. Not only the *XPUB/XSUB* but also *REQ/REP* to guarantee the **exactly-once** delivery. Actually, in the event of faults in a publisher-proxy or subscriber-proxy communication, we introduced acknowledgement communication in order to never violate the exactly-once delivery. Additionally, we were able to obtain durable subscriptions by storing information regarding subscriptions in the proxy. To prevent this information from being lost, it is also stored periodically in disk memory.

Finally, during the development of this project, we noticed advantages and disadvantages in the decisions made. For example: since we were using Java we had to implement *RMI*, which took some extra time. However, the trade-off is that it made it easier to implement and see the communication between subscribers and publishers, making it very transparent. Besides that, the implementation of *XPUB/XSUB* is not thought out to be reliable. Bearing this in mind, we were forced to use an additional communication service - *REQ/REP*. However, libzmq has a lot of underline code that we cannot change.