FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

SDLE 2021/2022

MEIC - FEUP

Project 1

Reliable Pub/Sub Service

Group 11

Davide Castro - up201806512@edu.fe.up.pt
Henrique Ribeiro - up201806529@edu.fe.up.pt
João Costa - up201806560@edu.fe.up.pt
Tiago Silva - up201806516@edu.fe.up.pt

Index

1. Introduction

In this project, we were tasked with creating a reliable publish-subscribe service. A publisher should be able to publish new topics so that subscribes can get news about topics they subscribe to. The publishers and the subscribers are connected by a proxy that saves all new information about the topics and which subscriber is subscribed to each topic. Furthermore, this service should guarantee "exactly-once" delivery and handle communication failures and crashes.

This project was developed using Java 17. It relies on three main classes: **Subscriber**, **Publisher,** and **Proxy**.

2. API

Both the **Subscriber** and the **Publisher** extend the class **SocketHolder**. This class is responsible for every operation relating to their sockets, e.g.: creating the socket, connecting to the endpoint, setting up some socket options, and reconnecting in case of connection timeouts.

Subscriber

The **Subscriber** class exposes three API methods: **Get**, **Subscribe**, and **Unsubscribe**.

The **Get** method retrieves a topic update. It takes a topic as a parameter and blocks until an update is available for that topic. In case the user hasn't subscribed to the topic it asked for an update on, the method fails and returns an error. This method starts by sending a message to the **Proxy** asking for its next update. It then waits for a reply from the **Proxy**, which can be one of the following: no updates on the topic (try again); the user isn't a subscriber of the given topic (error); topic update (success). It should be noted that in case of connection timeouts, the client will attempt to reconnect with the server and try the request again.

The **Subscribe** method subscribes the user to a given topic. All topic updates published after the subscription processing will be received by the client (provided it calls **Get** enough times). The **Proxy** notifies the client on duplicated subscriptions (not considered an error, just a warning). This method blocks until the proxy finishes processing the request. In case of duplicated subscription to the same topic, the **Proxy** warns replies with an error message. The client application warns the user, but continues the program's execution.

The **Unsubscribe** method unsubscribes the user from a given topic. All pending updates for the user will be lost upon unsubscribing. The user receives an error message when the **Proxy** detects that he isn't subscribed to the given topic. This method blocks until the proxy finishes processing the request.

Both **Subscribe** and **Unsubscribe** are idempotent and because of this it is not necessary to send and receive **ACK** messages.

Publisher

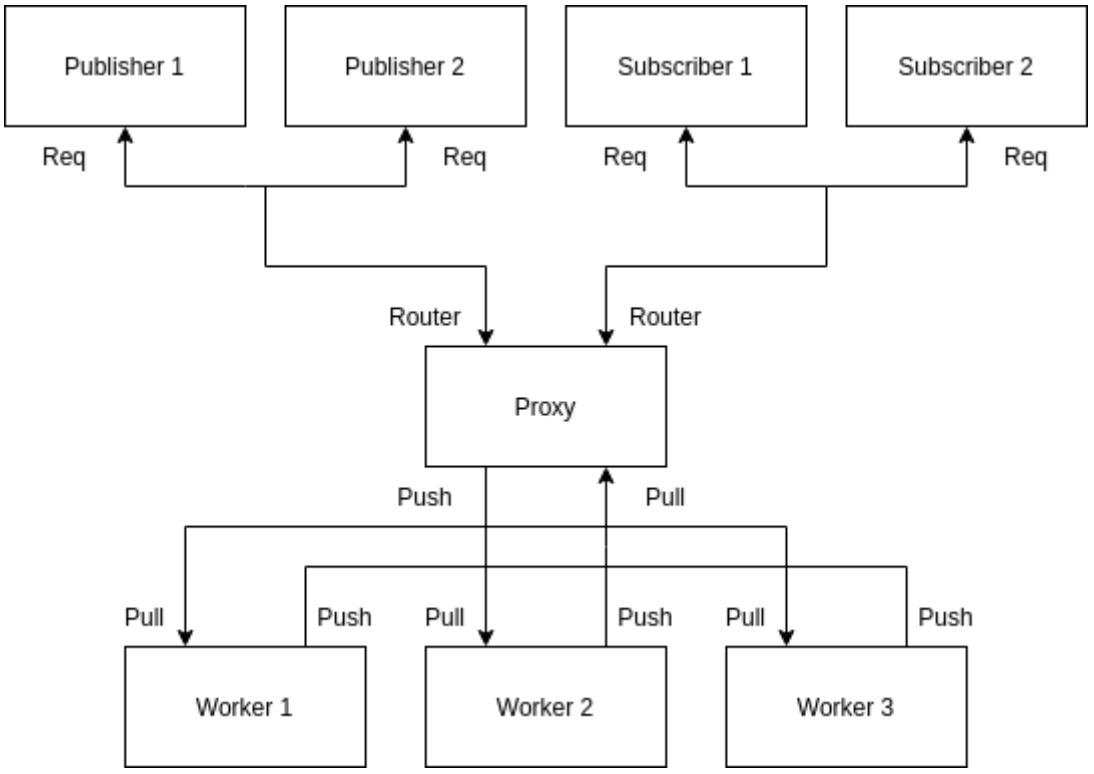
The purpose of the **Publisher** is to provide an API for the client to publish messages on a topic. These are the messages the subscribers to the given topic will eventually receive. For this purpose, the API exposes a **Put** method, whose arguments are the name of the topic and the content of the update. This method creates a message containing these parameters and sends it to the **Proxy**. It then waits for a reply. If it does not receive a success reply from the Proxy, the client will re-attempt to publish the content update.

3. Design choices

Proxy

The **Proxy** class handles the communication between the clients (publishers and subscribers), and manages the service's state.

The proxy handles incoming requests by polling its sockets. To make the management of the flow of received messages easier, the proxy listens on two ports: one for the publishers and another for the subscriber. It should be noted that this separation of sockets for each kind of request doesn't mean that each client can only perform one function. In fact, clients are identified by their **ID**, so each client can be publishing and subscribing to several topics. The processing of requests is delegated to *worker threads*: the proxy only receives and sends messages, leaving the rest to his workers.



Proxy workers

The **Proxy** handles a set of worker threads. These are initialized by the **Proxy**, each running an instance of the **ProxyWorker** class.

The **Proxy** uses a pair of Push/Pull sockets to communicate with these via ***inproc*** communication. The main advantage of these sockets is their internal queue, allowing for easy and lightweight load balancing: workers aren't required to send ready messages, like in the load balancing message broker example in the ZeroMQ guide.

These threads execute all the necessary actions needed to process the requests before sending the reply back to the **Proxy** via their own Push/Pull socket pair.

In order to gracefully close the application, the **Proxy** has a *control socket* that is used to send a *kill message* to itself (via ***inproc*** protocol). Upon receiving this message, the **Proxy** forwards it to its workers and waits for them to find the message before quitting.

Topic queue

In case the first node of the **Topic Queue** does not have anyone waiting to receive its information, that node is dropped, making the next one the queue's head.

This data structure is a form of simple single linked list. There's one queue for each topic. Each node in the queue represents an update on that topic. The queue holds a map that relates each subscriber with the next update they'll receive on that topic.

In order to free used space, the nodes of these queues hold a **reference count** that corresponds to the number of clients that will receive that update next. Whenever the **reference count** of the node at the head of the queue (the oldest update) reaches **0**, the node is dropped. The **reference count** can decrease when a user retrieves an update or unsubscribes from the topic.

Message handling

Subscribe message

Upon receiving a ***subscribe message*** from a client, the Proxy must add the client's identifier to the list of subscribers of the given topic. If there is no queue for that topic yet, a queue is created.

If the client is already subscribed to the given topic, a warning message is sent back to the client. Otherwise, the client will receive a *success reply*.

Unsubscribe message

When the **Proxy** receives an ***unsubscribe message*** from a client, it attempts to remove that client from the list of subscribers in the corresponding queue of the topic.

If there are no subscribers for that topic (the queue does not exist), or the client in question is not a subscriber to the topic, the **Proxy** sends an *error reply* to the client.

Get messages

Similar to *subscribe messages*, *get messages* create topic queues if the mentioned topic doesn't have a queue yet. These messages are used to retrieve content

updates from a given topic. These messages need to be acknowledged, so clients don't lose any of their messages.

Put

In the occurrence of a **Put** operation from a publisher client, the content is only inserted in the topic queue if the queue exists. This means that, in case there are no subscriptions to a given topic, the message is accepted, but the content is dropped.

4. Tradeoffs

Synchronization

Even though this is considered an antipattern by ZeroMQ, Java's synchronized statements and methods were used to synchronize the access to shared resources. The **Majordomo Pattern**, was considered as an alternative, but there would be tradeoffs. With this pattern, each **Worker** would be responsible for a set of topics. All requests relating to those topics would be forwarded to the correct **Worker**. The biggest tradeoff of this design would be the loan balancing between **Workers**: some topics can be a significantly more active than others, which may lead to situations where some **Workers** are constantly overwhelmed while others are free. By using synchronized methods, **Workers** may only block when accessing resources belonging to the same topic.

State keeping

Persistent storage is used to keep the program state. This ranges from active subscriptions to queued messages. Since access to permanent is slow, some optimizations are used to reduce the volume and frequency of these accesses.

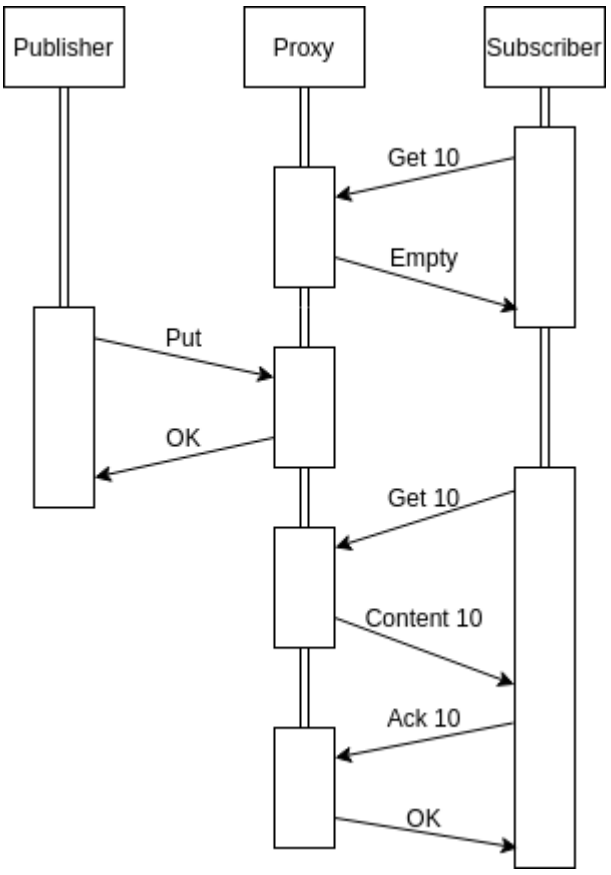
Every 50 messages, the current program state is saved to permanent storage. This means that in case of a sudden crash (that doesn't allow the exit handler of the program to save the current program state), some state will be kept without compromising the performance of the program too much.

The program state is saved to a directory containing one file for each topic. When backing-up the program state, only topics that have suffered an update are saved. This can reduce the volume of saved data significantly. To achieve this, the topic queue was made **Serializable.**

5. Failure model

Get message exactly-once assurance

To ensure the exactly-once policy, the **Subscriber** keeps a record of the last message ID it received for each topic. When fetching an update, the **Subscriber** informs the **Proxy** of the last update he received. Similarly, the **Proxy** keeps the ID of the last acknowledged message for each subscriber on each topic. This information is used by the subscriber to specify the next message expected, allowing the **Proxy** to recover from lost acknowledgement messages. The **Proxy** can also use this information to help clients recover from crashes by sending the ID of the current message on each reply.



Receive time-out

When a client process is receiving a message, a time-out is set. If this timeout is reached, the socket being used is destroyed, and the connection is restarted: a new socket is created and connected to the endpoint. This is done to prevent hanging in cases where the connection has been lost. The request is retransmitted after reconnecting.

Put message exactly-once assurance

To ensure only one **put** is done for each update, even in the presence of connection failures, a unique message ID is sent. This message ID is created by hashing (SHA-1 digest) the message's content and the current Unix timestamp. The last message ID, of the last update, of each publisher for each topic, is saved in the topic's queue. This ID is used by proxy to detect duplicate message publishing and to notify the

publisher. It should be noted that 2 updates can have the same content, but not the same timestamp for the same publisher (same message).

In case the new message ID is equal to the last one, an error reply is sent to the publisher. Upon receiving the error reply, the publisher skips checks how many time-outs occurred during message delivery. If there were time-outs during the delivery, the message is deemed delivered: the error occurred, because the publisher dropped a success reply. If there were no time-outs during the delivery, the publisher tries sending the message again with a different timestamp: in this case, it is assumed that the odd behavior is caused by either a process crash or a “too fast” publisher.