FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

SDLE

Assignment 1

Reliable Pub/Sub Service

Advisor: Pedro Souto

PORTO, OCTOBER 2022

Contents

1 Introduction

A reliable publisher/subscriber service was developed with resource to the ZeroMQ [2] library, using a binding for C++ (cppzmq [1]). The operations available are *put*, *get*, *subscribe* and *unsubscribe*. Subscriptions are persistent (remain until *unsubscribe* is called) and the service guarantees **exactly once** delivery. Topics are created by subscribing to a non-existing one and are deleted when the last subscriber unsubscribes.

2 Design

Two entities were developed: client and server. Clients can be both publishers and subscribers of the same and different topics, and the server functions as a broker between different clients. Communication follows the request-reply pattern and is done via ZeroMQ sockets. The server socket is of type REP and the client is of type REQ.

Client requests follow the following pattern:

- Put: PUT <client ID> <topic> <content>
- Get: GET <client ID> <topic> <ID of the last message received>
- Subscribe: SUB <client ID> <topic>
- Unsubscribe: UNSUB <client ID> <topic>

Success replies from the server follow the following pattern:

- Put: PUT <client ID> <topic> <content>
- Get: GET <client ID> <topic> <message ID> <content>
- Subscribe: SUB <client ID> <topic> | RESUB <client ID> <topic>
- Unsubscribe: UNSUB <client ID> <topic>

2.1 Data Structures

The server program stores a map from a topic name to a topic object. These topic objects are from the class Topic and store a message queue for every client that is subscribed to it. Messages are also objects from the class Message which contain the message ID, as well as its content. Topic names and message IDs are unique (within their class).

When a new message is added to a topic (via *put*), it is pushed to the queue of every subscriber of that topic. These queues are created when a client subscribes for the first time, and deleted when they unsubscribe.

3 Exactly-once delivery guarantee

As mentioned above, when calling the *get* instruction, the client has to pass as a parameter the ID of the last message received from a *get*. This allows the server to know whether their previous reply to a *get* from this client was received or lost. To allow for exactly-once delivery, the message at the front of a subscribers' queue is **only** removed if its ID matches the ID of the last message received by the client. While they are different, the message at the front of the queue is sent but remains in place.

In order for this guarantee to work after a crash or end of execution on the client side, the ID of the last message received from every subscribed topic needs to be stored on disk and updated after every reply from a *get*.

4 Durable subscriptions

Subscriptions are only terminated when a client explicitly unsubscribes. Subscriptions are marked by the presence (or absence) of a clients' queue in a given topic. If their queue exists, they are subscribed.

This is all stored on the server side, which means that regardless of the state of the client, the state of their subscriptions remain the same until further requests are sent to the server.

5 Problem of indefinite wait on the client side

To avoid indefinite wait after a *recv* call, a thread is started before calling the function *recv*. The new thread sleeps on a condition variable for 5 seconds and if it wakes up via timeout, it wakes up the main thread by shutting down the context. This means that an exception is thrown in the main thread whenever a timeout happens, so it has to be caught and dealt with. In our code, we simply print an error message.

6 Crash Resistance

The case of the server crashing might be unlikely, but a crash-safe mechanism was implemented nonetheless. When booting the server for the first time, a folder named Storage/ is created, which is used to store on disk the data structures present in memory. After rebooting, the contents of the folder are loaded into memory in order for the server to recover its previous state.

7 Issues/Vulnerabilities

In case several clients communicate concurrently with the server, the service performance is not the best since multithreading is not used in the server. This work focused on reliability rather than performance.

There might be concurrency issues in the client when the main thread wakes up from *recv* and at the same time the concurrent thread wakes up from their sleep via timeout.

References

[1] C++ ZeroMQ binding (cppzmq). https://github.com/zeromq/cppzmq/. Last accessed: 2022-10-14.

[2] ZeroMQ. https://zeromq.org/. Last accessed: 2022-10-14.

[3] ØMQ - The Guide. https://zguide.zeromq.org/. Last accessed: 2022-10-20.