U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Large Scale Distributed Systems

Reliable Pub/Sub Service

1MEIC03 - GRUPO 12

Caio Nogueira – up201806218@fe.up.pt

Diogo Almeida – up201806630@fe.up.pt

Nuno Oliveira – up201806525@fe.up.pt

Pedro Queirós – up201806329@fe.up.pt

Index

Introduction

The main objective of this project was to develop a dependable service that could handle a subscription based message delivery system. This means that the service would get messages from both publisher and subscriber entities to either publish or retrieve a message, respectively, having these operations fault-tolerable. In this document, we present and characterize in detail our approach to the problem, as well as the implementation and possible trade offs of our solution.

Architecture and Design

The implemented service provides four operations that must be fault-tolerable: *Put*, *Get*, *Subscribe* and *Unsubscribe*. The first two operations create and retrieve a message from a topic, respectively. The other two operations are self-explanatory, one allows a client to subscribe and the other to unsubscribe to a topic. With this in mind and the requirements presented in the project specification, we used a REQ/ROUTER architecture. The ROUTER socket type tracks all the connections it has, and tells the caller about their identification. This identification is transmitted in the frame of each message.

The service binds 2 sockets ("*publisher*" and "*subscriber*"), which are used for communication between publishers and subscribers, respectively. In the server loop, the server polls these 2 sockets. By doing this, the service accepts connections in a single queue.

The publishers connect to the "*publisher*" socket, in order to send the *put* message. Since publishers don't have an explicit identification, we use an arbitrary string (randomly generated) to send in the message frame as the connection's identifier. On the server's end, after processing the *put* operation, the service sends the reply to the publisher, indicating success (or not).

The subscribers connect to the "*subscriber"* socket to send requests (get, *subscribe* and *unsubscribe*). The *subscribe* and *unsubscribe* requests work similarly to the *put* request: upon receiving *subscribe* or *unsubscribe,* the server handles them (operating in the proper data structures) and returns a response, indicating that the service received the request and changed its internal state (or not) accordingly. The *get* call is a little more complex than the rest. After receiving the response from the server, the client needs to confirm having received that same message (to prevent losing messages in case of a crash on the server side). The confirmation (*ACK*) is done using a PUSH/PULL communication. These socket types are ideal in this scenario, since we need unidirectional communication (the client sends a message to the server and doesn't expect to receive a response).

Exactly Once

As specified, the program should guarantee "exactly-once" delivery, even in the presence of failures or crashes. This design ensures that a message is delivered to a client once, and only once.

To accomplish this, we used built-in Python data structures - lists, dictionaries and tuples - which fortunately are also *thread-safe*. Each message is given a sequence number (which is calculated by the previous sequence number for the message's topic +1). The sequence numbers are saved in the dictionary *sequence_number*. As for the clients, their information is stored in a nested dictionary: for each client, we need to save the information about the topics for which he is subscribed. Additionally, for every topic, we need to know the sequence number of the next message to be read by that particular client from that topic. Finally, the messages are stored in a list where each element contains the message's content, the sequence number and respective topic.

Subscriptions' durability

The project specification stated that the subscriptions must be durable, which means that they exist until explicitly deleted. In the context of the problem, a client who subscribes to a topic, must remain subscribed to that topic until he calls the *unsubscribe* method.

Our implementation meets the requirements, as the subscriptions are stored in the *clients_idx* dictionary. However, the messages don't need to be durable: they are only kept if there is a client currently subscribed to the respective topic and can be deleted from the system by the garbage collector method (if the message was already read by all subscribed clients).

Concurrency

Since the requests are *polled* in a single thread, the service allows concurrent requests to be sent by different clients. These requests are put on a queue. However, if the concurrent requests are coming from the same client, the program will crash, as the processes are *binding* to the same socket.

Garbage collector

In order to avoid the risk of running out of memory, every message that has already been delivered to all the subscribers can be removed from the server. To do this a thread is created to remove the messages that are no longer needed and every five seconds this same action is executed. The same logic is applied for the topics: the topics without subscribers are removed by the same thread. This thread that executes the garbage collector function, allows the system to have a higher scalability.

Reliability and fault tolerance

One of the requirements for the project is the reliability of the system and its ability to recover from crashes, either from the client side and the server side. The architecture that was chosen for the project helps to accomplish that because every request has a response from the server. However, it also needs confirmation that the client received the message in the get operation, something that couldn't be done only with the REQ/ROUTER implementation. To complement that, the client responds with an ACK message, after receiving the response of the request, to notify the server that the response was indeed delivered. Furthermore, the socket for this communication has a timeout of 2 seconds to receive the response. In the case that the response doesn't come in time (client crashed), the server will try to send that message up to five times. After those tries, the server will stop trying to send the message and will restore the previous state, assuming that the message was not delivered.

The state of the server is saved every time a request is received, which allows it to reload its state in the occurrence of a crash. This way, the information is always saved and when the server restarts it can continue its normal functionalities. The state is saved in a .json file every time a poll action takes place. If the server crashes, the client that was communicating with it will receive a timeout message, indicating the crash. Any client that tries to make a request to the server while it is down will receive the same timeout message.

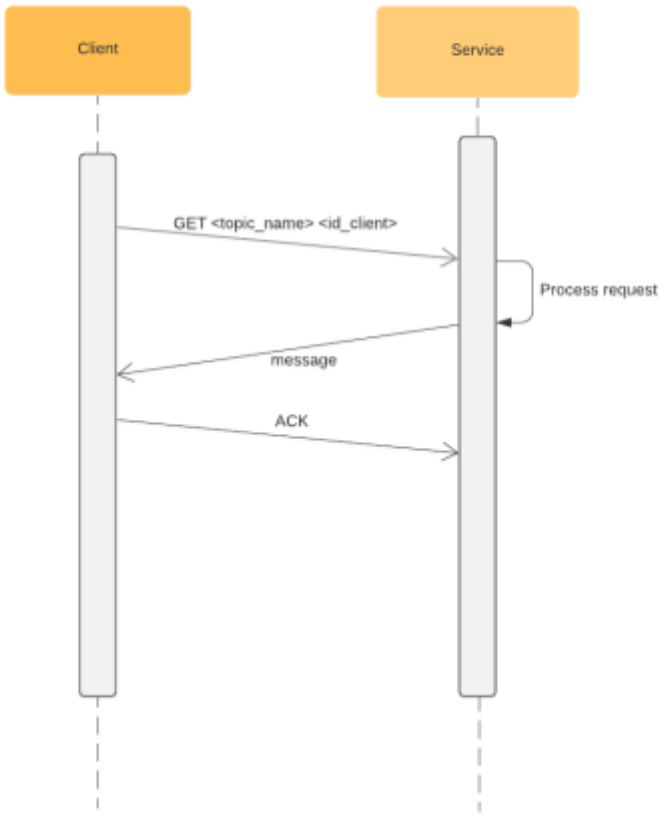

Figure 1 – Get Communication Protocol

Pros and Cons

The implementation adopted for this project, as any other, has advantages and disadvantages:

Starting with the advantages, the design used for the communication allowed to implement the required methods without exchanging a large amount of messages per request. This is a very important point, since the server reads and operates on the data structures in a *single-threaded* queue - the less time each operation takes to get done the better. The *send* calls to the different sockets are also done with the *DONTWAIT* flag activated, which speeds the polling iterations, as the service does not wait for a possibly crashed client.

On the other hand, we consider that our implementation doesn't take advantage of the *SUB* and *PUB* sockets, that could be used in this scenario where there is one end trying to publish messages and another subscribing to topics and receiving these messages. In addition to this, since the only operation that involves an ACKNOWLEDGEMENT is the *get* call, in the rest of the operations, the server cannot know if the clients have indeed received the return of the request or the message got lost in the socket. Furthermore, the fact that the requests are *stateless* means that the subscribers do not know to which topics they are subscribed: that information is only kept by the server, which saves it in non-volatile memory.

Conclusions

The developed program successfully implements all the goals and specifications necessary for the project, presenting one of the many possible approaches for the architecture: REQ/ROUTER.

There are other possible patterns that could be applied to this problem's context. However, this was the chosen approach and the one the group thinks it's better for the final purpose, with its advantages greatly outweighing the disadvantages