Reliable Publish/Subscribe Service
Large Scale Distributed Systems

João Baltazar
up201905616@up.pt
Faculty of Engineering of the University of Porto
Porto, Portugal

Luís Lucas
up201904624@up.pt
Faculty of Engineering of the University of Porto
Porto, Portugal

Nuno Costa
up201906272@up.pt
Faculty of Engineering of the University of Porto
Porto, Portugal

Pedro Nunes
up201905396@up.pt
Faculty of Engineering of the University of Porto
Porto, Portugal

Abstract

A publish/subscribe service is one of many possible messaging patterns typically used as a part of a message-oriented middleware (MOM) in distributed systems. This project's goal is to implement a reliable version of this service, using the *ZeroMQ* library.

The goal was achieved and the system meets the specifications. In this report, we showcase our implementation of the service using *Python*, while carefully explaining its reliability and failure scenarios and discussing possible improvements.

Implementing this service also prompted discussion, allowing for a deeper understanding of some difficulties in these types of systems such as the impossibility of perfect *exactly-once* delivery, information durability and server resilience and availability.

1 Introduction

In this project we aim to develop a publish/subscribe service using the *ZeroMQ* [8] library - a minimalistic message oriented library.

This service should allow a client to *subscribe* and *unsubscribe* a given topic, which is an arbitrary string. It should persist such subscriptions.

It should also focus on guaranteeing an *exactly-once* delivery, except in *rare circumstances*. There are no restrictions on the message order - this means that the message order for two clients subscribed to the same topic might be different. The only condition that must be respected is that both receive the same messages, as long as they are subscribed to that topic in the same time frame and call *get()* enough times.

All of these scenarios are discussed and taken into account, as well as many others that are essential to a service like this one. The following sections showcase our solution and explain some design choices at a theoretical and practical level.

2 Solution Overview

For communication between client and server to be achieved, a ZMQ_REQ socket is created client-side. The request to the server is sent making use of the auxiliary functions defined in the *clientreqs* file. There, all possible services available to the user are contemplated in the form of classes (put, get, subscribe, unsubscribe). After a request, the client will poll the response from the server with timeout. If no response is obtained, the client will follow the same protocol with the backup server.

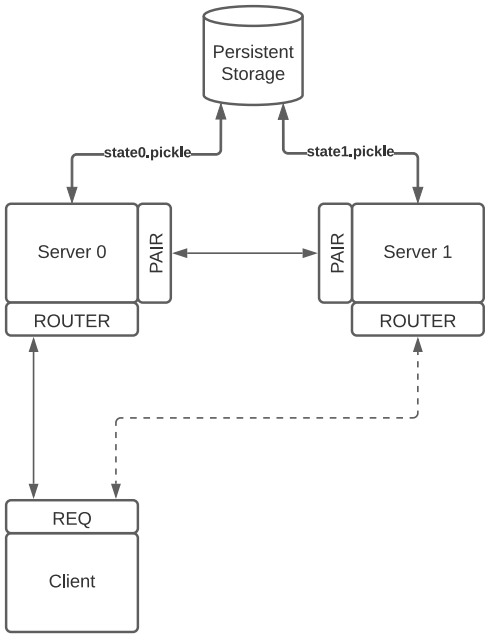

Figure 1. Service Architecture

Server-side, the ZMQ_ROUTER socket was chosen, as they are the asynchronous version of ZMQ_REP, allowing for independent reception of requests, and eventual replies. Furthermore, they allow the server to track and process the *identity* frame of the requests, which is instrumental for covering exactly-once delivery and built-in support for subscriber IDs: any given request is signed with the subscriber id via setting the IDENTITY *sockopt*, and the server socket raises a ZMQError whenever their response is routed to an unavailable client by setting the ROUTER_MANDATORY

option. The active server polls for requests, and creates tasks to handle each request. Periodically, it sends a state update through the ZMQ_PAIR in order to inform the passive server that it is alive and to allow replication. Both servers also store their state on persistent storage with some given frequency in order to account for total failure.

2.1 Subscriptions

As defined on the project specification, the service should support the *subscribe* and *unsubscribe* operations. These enable the management of subscriptions on the server, determining which messages are *put* on a given subscriber's message queue, thereby dictating what a client may *get*. Upon a *subscribe(id, topic)*, all subsequent messages on that topic will be stored and available for *get(id, topic)* (almost) exactly once, until *unsubscribe(id, topic)* is called.

2.1.1 Client Handling. The subscribe/unsubscribe request will be made to the server in the format: *subscribe(id, topic)* or *unsubscribe(id, topic)* and a subscribe/unsubscribe object will be created for communication with the server. If the client gets a response from the server via the *get_ack* function, an output will be granted indicating either that the subscription/termination was registered - *ACK* - or that the client was already (un)subscribed to that topic - *ASUB* or *NSUB*.

2.1.2 Server Handling and Management. On a subscription, the server adds the subscriber's id to the topic's subscriptions. On an unsubscribe, the server proceeds to remove the id from a topic's subscriptions. Afterwards, the topic's messages are removed from the message queue associated to the subscriber, and the ids associated to the topic get cleaned from the message pool in case their content is no longer required to be stored by any subscriber.

2.2 Getting Messages on a Topic

In order to consume messages from a topic, the service provides the *get* operation. This is the most critical operation, as reliability hinges on *exactly-once* delivery of messages, *except in "rare circumstances", i.e. only rarely and only under some failure conditions*. The guarantees and rare failure conditions are detailed in Section 3. On a *get(id, topic)*, the system will return one message of *topic* from the message queue of subscriber *id*, if there is one, and pop it from the queue. If there is none, the client will receive *N/A*.

2.2.1 Client Request. The get request will be made to the server in the format *get(id,topic)* and a get object will be created. If the client gets a response from the server via the *get_ack* function, an output will be granted indicating that.

2.2.2 Server Handling and Management. If a get request is successful, the server will proceed to pop the message from the message queue associated to said subscription and topic. If every other client that was subscribed to said topic has already received the message, the server proceeds to remove it from the message pool, since it doesn't belong to any other subscriber queue.

2.3 Putting Messages on a Topic

Lastly, the *put* facility enables putting messages on a topic. As with regular pub/sub patterns, publishers are anonymous and only send the topic and content of messages to the server. However, since they are not the stablest point of the architecture, they neither *bind* nor use the classic ZMQ_PUB socket. Instead, a *put* is handled as any request, via request/router. On a successful *put(topic, message)* operation, any current subscriber of *topic* will have *message* on their message queue.

2.3.1 Client Request. The put request will be made to the server in the format *put(topic,message)* and a put object will be created. If the client gets a response from the server via the *get_ack* function, an output will be granted indicating that.

2.3.2 Server Handling and Management. If the topic we're putting a message on exists, then an id is generated and the message is added to the message pool. The generated id is then added to the queues that are subscribed to said topic.

3 Exactly-once delivery

Exactly-once delivery is deemed to be impossible [2]. Assuming that we're using the service in a real-world scenario, the transport of messages is unreliable and the operations are time-bound. As such, since both the server and the client don't have access to each other's internal states, the exactly-once delivery can't be met.

There's a simple explanation to this. In the server's perspective, it must know that the message reached the client. As such, the client must send an acknowledgement message to the server - otherwise, the server does not know if the message transport has or has not been successful. However, if this acknowledgement message fails to reach the server, the server does not know if the communication protocol failed before or after the processing of the message. As such, if the server decides to re-send the information but the protocol failed after the processing, the client will receive a duplicated message. In the other hand, if the server decides not to send the information and the protocol failed before the processing, the client would not have received any message. In both cases, the exactly-once delivery is not respected.

To at least assure that the *get* operation only succeeds when the client receives it - considering that the ZMQ sockets detect routing failures -, we've altered its processing flow. As such, when processing a *get* request, the server peeks the message and sends it to the client. It only removes the message from the server's state queue of that subscriber when the ZMQ socket *send* function returns successfully, that is, the message was routed successfully and delivered

to the client. If the client does not receive the message, the server won't remove it from its queue, and thus it will still be possible to retry the operation and get the desired result.

However, there's still two faulty scenarios. The system will fail the *exactly-once* delivery when:

1. the server crashes right after sending the message to the client and before processing the message removal from the queue
2. the client crashes right after receiving the message from the server and before outputting the message

In the first case, the message will still exist in the server queue state, and thus will result in duplicated *get* operations (reflecting a *at-least-once* design). In the second case, the message will not be shown to the client and, as such, will be lost (reflecting a *at-most-once* design).

Apart from the referred scenarios, the server guarantees an *exactly-once* message delivery, as the project specification demands.

4 Asynchronous Design

Since the service's design does not account for multiple servers - not taking into account the backup server, which we'll explain later -, all requests will be handled by the primary server. As such, to ensure that all requests are taken care of, even in situations of high traffic, we've chosen to adopt an asynchronous design.

Given that the service is I/O bound, we chose to use Python's *AsyncIO* [4] package, paired with *aiofiles* [3] for asynchronous persistent storage reading and writing, and *zmq.asyncio* [7] for asynchronous zmq operations, such as sending, receiving and polling. A *ThreadPoolExecutor* [5] was also considered, but the differences are not big, and both allow concurrent execution - even though in a different way.

AsyncIO allows for concurrent execution of coroutines. A coroutine is a programming pattern that generalizes routines, so that they can be suspended or resumed. Unlike a ThreadPoolExecutor, which uses preemptive multitasking, AsyncIO uses cooperative multitasking, which requires threads of execution to explicitly yield control.

In our implementation, all coroutines are defined using the *async def* keywords. Given the current event loop, tasks are created and added to that event loop. Some of this tasks are:

1. handling incoming requests
2. writing to persistent storage
3. sending messages between the primary and backup servers
4. executing Binary Star checks
5. and others.

Whenever a coroutine is added to the event loop, it will eventually execute when another coroutine yields its control - such control yielding is associated with the *await* keyword. This allows for all I/O operations, namely writing to and reading from sockets, to not block, freeing the CPU for the execution of other coroutines. Since these read and write coroutines do not block, they return *Futures* instead - which can be awaited until completion.

This design allows the server to keep up high request loads by not blocking on any I/O operation, significantly increasing the service's availability.

5 System's Error Resilience

It is important to build a service that is error resilient, as its usage in the real world might come with several problems such as network unreliability, unexpected downtime and many others. We took into account two important factors in this service that are affected by those failure scenarios: availability and consistency.

5.1 Availability

To assure high server-side availability, apart from designing the server with an asynchronous pattern, we've implemented the Binary Star Pattern [1]. With this pattern, there's two running servers: a primary and a backup.

All requests are handled by the primary server - active - while the backup server monitors the primary server's activity. Whenever the backup server detects that the primary server is unavailable, failover occurs and it takes over as the primary server.

However, we've designed this pattern in a slightly different way. In a traditional Binary Star Pattern, whenever the primary server fails or crashes and the control is taken over by the backup server, to pass the control over to the first server when it recover one would need to shut down both servers and restart them by the correct order - when the service's traffic is the lowest, for example. To avoid this, we've changed its semantics - the backup server, when it takes over, becomes the primary server. As such, when recovering the other server, it should restart as a backup server. Passing the control over to the first server would only require the administrator to shut down the now primary server temporarily, so that the control can pass to the intended server.

The client has access to both the primary and backup's ports, but doesn't know which one is which - it does not need to. As such, when trying to send a request, it attempts both servers, and verifies which responds.

This approach, while simple and easy to understand, is effective and allows great, and in most cases sufficient, availability.

5.2 Consistency

In order to guarantee that the server keeps up and stores incoming requests properly, we've implemented a state to file saving mechanism.

Every 5 seconds, the current state of the server application, which includes the list of subscribers on each topic, as well as

the message pool and the message queues for each subscriber on each topic, is stored using *pickle* [6]. This package allows object serialization, so that it can be easily read from memory.

This serialization also happens when sending the primary server state to the backup server. By doing so, one can easily reconstruct the state on the yet stateless server. Upon failure of the primary server, the backup server can easily start acting as the primary with the most recent state it has received.

This approach allows for the general application state to stay consistent in both the primary and backup servers, and to also recover with the most recent saved state after crashing.

6 Improvements to be made

Although the presented solution is resilient and works as intended, some improvements could be made to some key parts of the project.

Firstly, the asynchronous design could be easily paired with threads pools. Although AsyncIO is concurrent by nature, using threads, each with its own event loop and set of coroutines, could help improve the server's availability even further. However, the specification of ZMQ indicates that its sockets are thread-unsafe - thus, this improvement is subject to further investigation, since ThreadPool tasks are required to be thread-safe.

Another improvement would be to add more servers for load balancing. By having more servers, each responsible by its own shard of topics, the service could handle even bigger traffic spikes.

Some other solutions regarding the service's consistency could also be explored, since the time difference in-between heartbeats between the primary and backup servers could lead to eventual inconsistencies. For example, if requests are handled before the server's state is sent to the backup, and the primary server crashes, the backup's state may be outdated and the service may be inconsistent.

7 Conclusions

All of the project's goals were achieved - all features were properly implemented and are complete, well designed and working as intended. Despite possible improvements, the presented solution is robust, tackling even the most glaring "rare circumstances" under which such a system might fail to deliver the promise of exactly-once delivery. By combining various distributed systems patterns, we were able to achieve a fairly available, consistent, persistent and reliable publish/subscribe service.

References

[1] Pieter Hintjens. 2022. *ZeroMQ.* https://www.oreilly.com/library/view/zeromq/9781449334437/ch04s13.html

[2] Savvas Kleanthous. 2022. *The impossibility of exactly-once delivery.* https://blog.bulloak.io/post/20200917-the-impossibility-of-exactly-once/

[3] PyPI. 2022. *AIOFiles.* https://pypi.org/project/aiofiles/

[4] Python. 2022. *Asyncio - asynchronous I/O.* https://docs.python.org/3/library/asyncio.html

[5] Python. 2022. *Concurrent.futures - launching parallel tasks.* https://docs.python.org/3/library/concurrent.futures.html

[6] Python. 2022. *Pickle - Python object serialization.* https://docs.python.org/3/library/pickle.html

[7] ZMQ. 2022. *AsyncioZMQ.* https://pyzmq.readthedocs.io/en/latest/api/zmq.asyncio.html

[8] ZMQ. 2022. *ZeroMQ.* https://zeromq.org/