**Faculdade de Engenharia da Universidade do Porto**
**Mestrado em Engenharia Informática e Computação**

# Reliable Publish-Subscribe Service

## Large Scale Distributed Systems - First Project Report

*Miguel Azevedo Lopes | Diogo Filipe Maia | Mariana Monteiro | Luís Carlos Barros Viegas*
*up201704590@fe.up.pt | up201904974@fe.up.pt | up202003480@fe.up.pt | up201904979@fe.up.pt*

October 2022

# 1 Introduction

In this first project of the Large Scale Distributed Systems course, we were tasked with developing a reliable publish and subscribe service.

### 1.1 Requisites and Constraints

The service supports 4 key operations: subscribe, unsubscribe, get and put. One of the main requisites was that it had to be developed on top of libzmq, a minimalist message-oriented library. The constraints/requisites provided in the project's specifications included:

- Topics are created implicitly when a subscriber subscribes to a topic that does not exist yet
- All subscriptions are durable, according to the Java Message Service's (Jakarta Messaging) terminology
- The publish-subscribe service should guarantee "exactly-once" delivery, in the presence of communication failures or process crashes, except in "rare circumstances"

# 2 Implementation Aspects and Design Choices

### 2.1 Implementation Language

The language we chose for this project was Java, for no particular reason other than, it is the language all four elements of our group feel the most comfortable writing code in.

### 2.2 Basic Architecture

When deciding on the architecture for our project, we opted for simplicity: a central server that holds all the topics and their respective messages. This server serves the clients' requests, whether their publishers or subscribers, through a single access point, or in this case, a single socket. We based our choice of architecture on the Lazy Pirate Pattern for client-side reliability, a pattern retrieved from the ZeroMQ guide (Chapter 4 - Reliable Request-Reply Patterns).

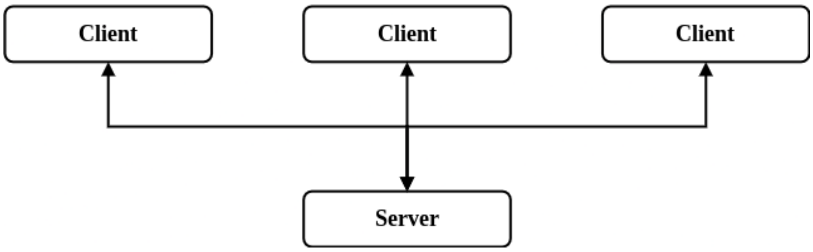


**Fig 1.** Architectural Schema

### 2.3 Communication Pattern

When creating a service that uses ZeroMQ, one of the first aspects that need to be determined is the communication pattern.

The ZeroMQ official guide provides a few possible patterns for a publish-subscribe service, however, after thorough analysis, we realized these patterns weren't compatible with the project's specifications. Quoting the guide itself, the PUB socket communicates to *"all of many"* and *"Subscribers can't tell publishers anything that would allow publishers to control the rate of messages they send. Publishers only have one setting, which is full-speed, and subscribers must either keep up or lose messages."*. This last quote alone invalidates the get mechanism, since, according to the project's specification page *"topic's subscriber should get all messages put on a topic, as long as* *it calls get() enough times"*. For this reason, we ruled out PUB and SUB socket types entirely and opted for a pattern described in chapter 3 (Advanced Request-Reply Patterns), the REQ to ROUTER socket combination. According to the guide, this combination is ideal for *"an asynchronous server that can talk to multiple REQ clients at the same time"* and can be used *"as an application that reads the message and acts on it"*, which is exactly what our design proposes: a main thread, that reads the messages sent by clients (publishers and subscribers) and submits work to a thread pool, as can be seen in Fig 2.

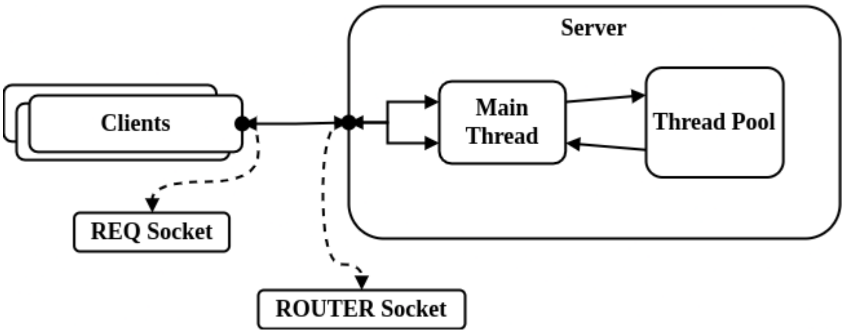


**Fig 2.** Communication Pattern Schema

### 2.4 Clients

The class Client is inherited by both Publisher and Subscriber. This class sets up ZeroMQ REQ sockets and handles socket-related operations such as socket setup, sending and receiving messages (with the help of the ZeroMQ ZMsg class), and closing sockets.

#### 2.4.1 Publisher

The Publisher's task is to publish messages on a topic. These messages are sent to the server and posteriorly delivered to the subscribers of such topics.

To achieve its goal, the Publisher has a method called Put.

#### 2.4.2 Subscriber

A topic is created when a Subscriber subscribes to a new topic. After being subscribed, the subscribers can request messages using the get method. A get can result in several outcomes such as informing the client that there is no topic with that name, informing the client that he is not subscribed to that topic, the desired topic content, etc.

If a client does not want to subscribe to a topic anymore, they can unsubscribe using the unsubscribe method.

### 2.4 Server

The server is composed of one main loop, where the main thread of the server polls the ROUTER Socket to check for new messages. If there are messages, they're retrieved and sent to be handled by a worker thread. This worker thread will treat the message according to its type and create a reply message, which is then placed in a queue, which is in turn emptied in the main loop (referred to before), as the main thread sends out those messages to their respective clients.

#### 2.4.1 Worker Threads and Synchronization

As mentioned previously, a worker thread treats a message according to its type. It then creates reply messages accordingly and adds them to a message queue. All the tasks performed by the threads leave space for some concurrency issues since they can be accessing the same data structures at the same time. In order to solve this problem we took advantage of Java's synchronization features. Methods in the Topic class for example, which is especially prone to concurrency, were turned into synchronized methods. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

### 2.4.2 Thread Pool

In order to reduce the number of application threads and provide management of the worker threads, we decided to implement a thread pool. We opted for a cached thread pool since it is supposed to increase performance in scenarios where a program is performing many short-lived tasks. A cached thread pool reuses previously used threads if available and creates new ones if needed. Idle threads in a cached thread pool are only kept alive for 60 seconds, after which they are discarded.

### 2.4.3 Maintaining State

To implement a system that guarantees "exactly-once" delivery, we keep the program state in a file, to ensure that if the server crashes, the information regarding each topic, its subscribers and messages, is recuperable. This file is updated every time the server shuts down or sends a new message. Whenever the server is started, it checks if there is a file with the state of the program.

## 2.5 Data Storage

Amongst all the classes used to implement this service, we decided to highlight the Topic class and the Message class in this section, seeing as they are the classes that carry the most information at any time during the execution.

### 2.5.1 Topic Class

The topic class holds information regarding each topic. All the topics are stored in a Hash Map in the Server class, which maps the topic's name (a string) to the actual class object.

In the class itself, there are 2 main data structures worth mentioning, a linked hash map that maps the messages' IDs to their corresponding Message object and a hash map that holds the topic's subscribers' IDs and maps them to the message ID of the first message they will receive when they call the get method. We opted for a linked hash map to store the messages because, unlike a simple hash map, a linked hash map maintains insertion order, which is key in determining which message a subscriber gets since that depends on when they first subscribed to the topic.

### 2.5.2 Message Class

The Message class is composed of 5 fields:

- Message Type which can be: SUB, UNSUB, PUT, GET, GET_REP, ACK, ERROR
- Client ID, the ID of the client who sent the message to the server or who is the recipient of the message from the server
- Topic, the topic to which the client on which the operations will be performed. May be null, depending on the message type
- Content, can be the content of a publish message, the content of a get reply message, or even an error message. Can also be null for some message types
- Message ID, the hash of the client's id and the time the message is created, used to prevent duplicates

The class also contains a constructor that creates a Message object from a ZMsg object and a method that creates a ZMsg object, which is then used to communicate through the sockets.

When creating messages for the server to reply to the clients, or in other words, from a ROUTER socket to REQ socket, the first ZFrame contained in the ZMsg is the client ID so that the ZMQ Socket can know where to send the message. The same isn't done for messages sent from the client to the server (REQ to ROUTER) because in that case, the ZMQ method to send messages adds the ZFrame containing the client ID automatically.

## 3 Service API

### 3.1 Subscribe

API Call: *subscribe <topic> <senderID>*

This message is sent by a client who wants to subscribe to a topic. There are three possible response scenarios. First, if the topic doesn't exist, it will be created, and the client will be added to the subscribers list. On the other hand, if the topic already exists and the client has not yet subscribed, an acknowledgment message will be returned by the server. Finally, if the client has already subscribed, the server will reply with an acknowledgment message, but nothing will happen on the server side since subscribing is an idempotent operation.

The idempotency of the operation means there is no need to guarantee anything other than the server receiving the message.

### 3.2 Unsubscribe

API Call: *unsubscribe <topic> <senderID>*

This message is sent by a client who wants to unsubscribe to a specific topic. After receiving this message, the server response is similar to the subscribe message. In this situation, the same three scenarios exist, but in this case, if the topic doesn't exist, it won't be created. In opposition, if it doesn't exist, an error message will be sent. At the same time, the client's ID will be removed if he is subscribed to this topic and an error message will be sent if the client is trying to unsubscribe from a topic he isn't subscribed to. Just like what happens with the subscribe message, there is an acknowledgment message to let the client know that their message was received.

The unsubscribe operation is also idempotent, so there is no need to guarantee anything other than the server receiving the message.

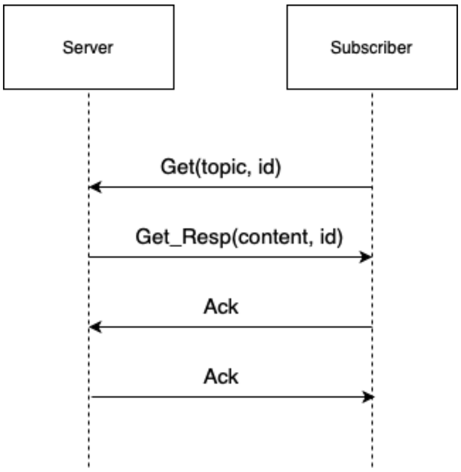
### 4.3 Get

API Call: *get <topic> <senderID>*

This message is sent by a client who wants to get new messages on a specific topic. If a client is not subscribed to this topic, the server will send an error message. On the other hand, if the client is subscribed, there are two possible situations: the topic has no new messages, so the server informs the client, or, if there are messages from this topic that the client did not receive, the next message will be transmitted.

After the server sends a message from the topic, the client informs the server that he has received it, with an acknowledgment message, and after, which is followed by another acknowledgment message sent by the server. This exchange of acknowledgment messages allows us to safely update the server state regarding the client status in that topic, hence ensuring exactly-once delivery.

**Fig 3.** Communication between Server and Subscriber



### 4.4 Put

API Call: *put <topic> <message> <senderID>*

This message is sent by a publisher that wants to publish a new message to a topic. If this topic exists, a new message will be placed in the topic's message list and the server will send an acknowledgment message. If this does not happen, or this message is already in this topic, the server will send an error message.

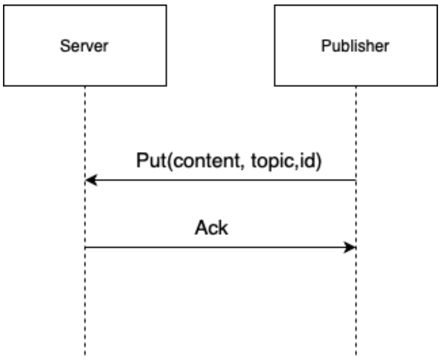


**Fig 4.** Communication between Publisher and Server

## 4 Failure Model

### 4.1 Exactly-Once Delivery

In the following sub-section we address the exactly-once delivery guarantees that were part of the project's specification.

### 4.1.1 Get Operation

As was explained in section 3.3, during the get operation, after the client sends an acknowledgment message, the server returns another acknowledgment message. This exchange of acknowledgment messages is done to ensure exactly-once delivery. It is only when the server receives the acknowledgment message from the subscriber that he proceeds to update that subscriber's next message. So, when the client does not receive the acknowledgment from the server, he tries to send the acknowledgment message three times until it gets a response. Subsequently, if the client doesn't receive a response from the server, he gives up. In case, the client crashes and does not receive the acknowledgment message, but the server received the acknowledgment, no harm is done since the last acknowledgment causes no change in the state of the service.

The only case that we can't ensure "exactly once" is when the server crashes for a long period of time and doesn't receive the acknowledgment message, so, when it finally restarts, the client has already given up. We consider that a rare circumstance because the server would have to crash exactly after sending the get reply and before the exchange of acknowledgments, which is a small window of time. In addition, in case of no response when sending messages, clients retry three times, and on each try, there is a timeout to wait for the server's response. In a real-life scenario, this timeout could be adjusted to reflect the time it takes the server to restart after a crash, hence reducing this already small risk.

### 4.1.2 Put Operation

To ensure that the put of a message in a topic occurs exactly once, we create a unique id for each message. This id is created by making a hash of the client's ID with a timestamp of the creation of the message. Since it is impossible that a client builds two messages at the exact same time, we can guarantee that the server can safely delete messages with the same id, since they were probably sent because of some sort of error, and not intentionally (two messages can have the same content but never the same timestamp).

### 4.2 Tradeoffs

When brainstorming about the project's structure, we thought of using an adaptation of the technique used on TCP connections regarding identifying out-of-order packets of data on a transfer. This way we could use a sequence number on the messages we used and that would allow us to identify not delivered messages and guarantee the 'exactly once' ideology. However, we designed this system so that the clients are stateless, and implementing this technique would violate this system principle. This is a tradeoff we were willing to make since we have confidence that with all the safeguards already in place, the chance of failure is minimal.

## 5 References

- Oracle. OpenJDK 17 Documentation. (https://devdocs.io/openjdk~17/)
- ØMQ. ZeroMQ Guide. (https://zguide.zeromq.org/)
- Two Generals Problem (https://haydenjames.io/the-two-generals-problem/)
- Exactly-Once Message delivery (https://exactly-once.github.io/posts/exactly-once-delivery/)