Reliable Pub/Sub Service

SDLE 2021/2022 — 1st Semester

November 2021

By 1MEIC02 — Group 12

Ana Barros	up201806593@edu.fe.up.pt
Diogo Rosário	up201806582@edu.fe.up.pt
Ivo Saavedra	up201707093@edu.fe.up.pt
João Martins	up201806436@edu.fe.up.pt

Table of contents

1. Introduction

This report aims to describe the process of designing and developing a reliable publish-subscribe service. Its focal point is to describe the chosen architecture, implementation aspects and possible tradeoffs, with a focus on algorithms to ensure the “exactly-once” guarantees.

The following section will begin by defining the base components of our implementation (Messages, Node, and Broker), followed by our initial architecture. It will then specify three auxiliary classes used to manage topics, manage thread work and ensure broker reliability. Finally, the final architecture, along with its benefits and drawbacks, will be discussed.

2. Project Overview

2.1. Project Specification

The project requirements state that the service must offer four operations, *put()*, *get()*, *subscribe()* and *unsubscribe()* methods, used to ease communication between a set of subscribers and publishers. The handout states also that all subscriptions are durable, i.e., are maintained after a peer reconnects, and the service should guarantee an “exactly-once” delivery, if “extreme circumstances” do not occur. The group believes to have implemented an API that guarantees these requisites.

3. Architecture

The C++ programming language was chosen due to its efficiency, reliability, and the group’s familiarity with it. The application was built on top of *zmqpp* [1], a high-level ZeroMQ library. Two main architectures were designed, one with a focus on efficiency and the other with a focus on reliability. Both of these architectures rely on a set of nodes that send requests to a central broker/server, which, after handling them, responds with a reply.

3.1 Messages

In order to ease communication between nodes and the broker, a *Message* class (message.hpp) was implemented. It encompasses all operations relevant for this project. Seven message types were created, four of which are relative to the **PUT**, **GET**, **SUB** and **UNSUB** requests. The remaining are: **ANSWER,** the response to a **GET** request; **OK**, an acknowledgment message and **KO**, a NACK message.

Every message has the following structure <*type*> <*topic_name*> <*id*> <*seq_num*>, where “type” is the message type and “id” is the identifier of the node. The only exceptions to this rule are the PUT and ANSWER messages, which require a body that is appended to the original message. A message also contains a sequence number, which is used in the Titanic Pattern to prevent confounding similar messages that belong to different requests. Furthermore, all messages are serializable, so that they can be written into files, as specified by the Titanic Pattern.

3.2 Node

According to the group's interpretation of the handout, there is no distinction between subscribers and publishers in the network. As such, a *Node* class (node.cpp) was implemented that represents any peer that makes requests to the broker. The concepts of node and peer will be used interchangeably from now onwards to represent a client of the API. The *Node* class contains the required functions of the API and can be used as both publisher and subscriber.

Each node has a single socket that is used to communicate with the broker and an unique identifier. To satisfy all requirements, it is vital to ensure that:

- The server can send an acknowledgment message to a client's request. This makes bidirectional communication mandatory.
- A client blocks when receiving a response if the server is unavailable.

For all these reasons, the **REQ** socket type was chosen for the nodes, as it is bidirectional and blocks in mute states.

The Node class provides the API through the use of the following methods:

```
int get(std::string topic_name, std::string &msg);
int put(std::string topic_name, std::string msg);
int subscribe(std::string topic_name);
int unsubscribe(std::string topic_name);
```

The *subscribe()* method is called whenever a node wishes to subscribe to a given topic. It starts by sending a *SubMessage* with the provided *topic_name* and the peer's ID. Afterwards, the node awaits for an **acknowledgment** message, *OKMessage*, from the broker. The *unsubscribe()* method has a similar flow, the only difference being sending an *UnsubscribeMessage* instead of a *SubMessage*.

The *get()* method is used by a subscriber node to request for an entry from the specified *topic_name*. First, a *GetMessage* is sent to the broker, which then replies with an *AnswerMessage* with the content from the requested topic. If no content is available for that topic, then this method keeps resending the same request until it receives an answer from the broker. If the node in question is not subscribed to the requested topic, the broker replies with NACK, KOMessage.

The *put()* method is used by publisher nodes, in order to publish a message on a topic. A *PutMessage* is created with the provided *topic_name* and *content* and is sent to the *Broker*. Empty messages are not allowed.

3.3. Broker

The information flow is handled by the *Broker* class (broker.hpp), which has two different sockets, a topics queue and a defined number of workers. It is responsible for the following tasks:

- Accepting connections from nodes and workers;
- Accepting requests from nodes and storing them in a queue;
- Sending the requests to the workers, managing traffic according to the load balancing pattern;
- Receiving the replies from workers and transmitting them to the requesting node.

During the initial implementation of the broker, it was clear that the bottleneck of the service was caused by handling message requests. To keep this delay to a minimum, all received requests were redirected to a set of workers running in a multithreaded fashion.

3.4. The Load Balancing Pattern

When designing the initial architecture, the focus was to keep the service as efficient as possible, without any concerns for fault tolerance. The **Load Balancing Pattern** was chosen, after reading and analyzing the ZeroMQ guide [3], because of its simplicity of implementation and apparent efficiency. A tag with the name of *load_balancing* is present in the group repository with the group's implementation. The UML diagram can be found in **Appendix 1**.

The pattern consists of a broker which receives requests from a set of nodes (publisher and subscribers) through a frontend socket (**ROUTER**). This allows receiving asynchronous requests, i.e., responding to an accepted request is not required before accepting a new one. These requests come with a client ID, which will be used to route the response back to the respective peer. The broker dispatches each request to a worker, if any is available. This communication is done through the backend socket (**ROUTER**). When a worker is finished with a request, it sends the response back to the broker.

The group's initial approach to distributing requests to workers was to use **round-robin routing,** i.e., the assignment of work is sequential and always follows the same order. But this wasn't efficient, leading to a considerable delay in task execution due to an unfair workload distribution between processes. To overcome this, the load balancing pattern was adopted. It states that the workers are responsible for notifying the broker when they are ready for a new request.

This pattern achieved satisfactory results, sending around 40000 messages per minute, without causing any message duplication. The tests were performed on an average processor (i5-6300U) and they can be found in the config/ folder (t2n1.txt, t2n2.txt).

3.5. Topic Queues

In order to manage peer subscriptions and pending subscription messages, the *TopicQueue* class was created. There is a map to handle each queue, where the keys are the topics' names and the values are *BetterQ* objects. The *BetterQ* class is responsible for information relevant to a specific topic, such as the content published on that topic, represented as a list. Additionally, which peers have subscribed to the topic and their respective position in the topic's list is also maintained, as well as a map of peer ID's to list iterators.

The use of a list instead of a queue is justified by the fact that the deque and queue STL classes reallocate their contents in memory when they reach their maximum size. In contrast, the list class requires no reallocation and provides O(1) when popping or inserting elements.

The possibility of simultaneous calls to the same topic queue, when using multithreading, requires the use of a mutex in the *TopicQueue* class. The *actor* model, as mentioned in the ZeroMQ guide [3], was considered as an alternative, but due to the group's unfamiliarity with it and due to time restrictions, it wasn't implemented.

The *TopicQueue* class provides a set of methods that are similar to the API. Due to this resemblance, they won't be analyzed in much detail, except for some non-trivial design decisions made to increase performance. These are:

- The *put()* method doesn't append content to a topic if it has no subscribers.
- The *get()* method yields an empty string if a subscriber has no available content of a topic.

- The *get()* method automatically trims all contents that are unreachable.
- The *unsubscribe()* method deletes all contents of a topic if it has no subscribers.

3.6. Workers

As explained previously, the backend layer consists of a set of workers, implemented in the *Worker* class. Its main functionalities are to process requests in a dedicated thread. Each worker has its own ID and has access to the topics queue (so it can manipulate the queue according to the requests it receives).

The process of handling each message won't be explained in detail, as the responses for each message request were previously specified in the Node section.

3.7. State

In order to save the state of the broker and its queues, we opted to use the *cereal library* [2]. This library allows us to serialize the *TopicQueue* class, so it is much easier to both save and load the state of the *TopicQueues*. The process of saving the broker is either made when handling a request or when the broker terminates successfully.

3.8. TestApp

A *TestApp* class was implemented to ease testing of the application. It reads from configuration files, which format is specified in detail in the README file. Each line represents an operation, which can be one of the API's calls or a sleep operation. A *main* program was also developed to test a single function call of the API. Its usage specification is also in the README file.

3.9 The Titanic Pattern

When confronted with the necessity of ensuring tolerance to failures, the group made tests to determine the robustness of our initial design, by simulating network disconnects and node/broker crashes. Duplicated messages were observed when the network reached high levels of traffic or when the broker was recovering from a crash. As such, the "at least once" restraint wasn't being enforced.

The section Reliable Request-Reply Patterns from the ZeroMQ guide [3] was consulted to help with finding an alternative that ensures fault tolerance. After some consideration, the *Titanic* pattern was chosen because it was easy to integrate with the initial architecture and because it was focused on ensuring exactly-once delivery.

The *Titanic* pattern establishes a new API to be used on top of the previous design. It consists of three requests, represented in the *TitanicMessage* class (titanic_messages.hpp). These are:

- *Request*: issue a new request to the broker.
- *Get:* get the result of a previous request, if available
- *Delete:* delete a request.

The UML diagram can be found in **Appendix 2**.

Each message has its own ID, independent of the client ID, which is created by hashing the request message (PUT, SUB, GET or UNSUB messages). A sequence number is appended to each message to prevent confounding different requests with the same

content. For example, two subsequent GET messages to the same topic would hash to the same value if not for this number.

A client that wants to issue a new request must: firstly, send a *Request* message with its request wrapped in it. The broker will respond with the ID of that request. Secondly, the client must send a series of GET messages with the request ID until it receives the desired response. Lastly, it submits a *Delete* message to delete the request from the broker.

The *Broker* class is no longer responsible for handling client requests. Instead, it redirects them to the *Titanic* class (titanic.hpp), which is responsible for handling the titanic pattern's messages. If a new *Request* is received, a new file, which is named after the message's ID, is created, and the request is sent to a *Worker*. When a *Worker* is finished with a request, it writes the response into the created file. When a client sends a GET request, the *Titanic* will respond with the content of the file if it contains the response, otherwise, it will reply with a KOMessage, signaling the client that the response isn't yet available. A *Delete* message is followed by deleting the file of the corresponding ID.

This approach makes the requests idempotent, as duplicate requests will be discarded by the broker. This fixes the issue of duplicate messages, and ensures exactly once delivery. It also fixes most of the problems related to fault tolerance, as all requests and their responses are stored in non-volatile memory, which allows for the broker to resume any pending client requests after a crash. It also aids in ensuring client fault tolerance, as a client state could be saved in order to resume pending requests after a crash. However, this wasn't implemented due to time restrictions.

The main issue with the *Titanic Design* is its need to store every message in separate files, making it inefficient. The same test mentioned previously (t2.txt) was performed, under the same circumstances. Around 13000 requests per minute were processed, which is almost a third of the throughput of our previous implementation. This lack of efficiency could be mitigated by storing all requests uniquely on a single file. One additional problem with the proposed design is that the handling of *TitanicMessages* is sequential, using a REP socket. The obvious solution is to use a ROUTER socket instead, similar to the first proposed implementation (*Load Balancing*). The group decided not to implement this enhancement because it would require storing the client ID on each message, and the project's deadline was approaching.

4. Trade-Offs & Possible Improvements

A recurring theme that the group's members found when reading about and designing distributed systems, is that achieving a perfect fault-tolerant system is nearly impossible, mainly because there are always some enhancements that can be made. Due to this reason, only the most valuable improvements will be mentioned, as the number of possible enhancements that the group drafted were too numerous for this report.

A possible improvement to the *Node* class is to issue all requests all at once, at the beginning of its execution. Afterwards, they could be retrieved with the use of multithreaded tools, as different sockets could be used to request different *Titanic* GET messages. The main problem with this improvement is that the current implementation of the broker doesn't ensure sequential consistency, and the order of handling each message could affect the end result. To mitigate this issue, a mechanism to wait for requests which have fallen behind could be implemented, possibly by using an auxiliary data structure to store which messages were requested by which client. This data structure could be used to ensure that no

message had fallen behind. One could also improve upon this solution in terms of efficiency, by deciding which messages were affected by the order beforehand, and only waiting for messages that would affect the end state. For example, the order of two SUB messages doesn't matter, but the order of a SUB and a GET message to the same topic affect whether a client receives the content. An intelligent approach would notice this and wait for the SUB message before the GET message, while not caring about both SUB messages. This would raise a new issue, which is whether the broker should guarantee linearizability. This would involve storing the timeframe of each request, and handling it in order. The group considered implementing these improvements proved too time-consuming when taking into consideration the scope of the handout.

Furthermore, when testing, it was observed that inducing a broker crash after it receives any request and before it could reply with a response would cause a client to block on the *receive()* call. This situation was discussed in the lectures, as illustrated in **Appendix 3**. This is caused by the nature of the REP-REQ socket, and is ZeroMQ intended behavior. To circumvent this, the **ZMQ_RCVTIMEO[4]** socket option and making the *receive()* call non-blocking were tried, followed by a socket reconnection to the broker. However, this attempt wasn't successful, as the send method in ZeroMQ is non-blocking and the flag **ZMQ_SNDTIMEO[4]** didn't work. This caused the client to never wait for a broker to start. Instead, the client was failing by trying to send requests to an offline broker. An alternative that the group found was to use **ZMQ_REQ_RELAXED[4]** and **ZMQ_REQ_CORRELATE[4]** in order to allow sending REQ messages before receiving their respective reply, REP. This wasn't fully implemented due to time constraints, because it requires processing and redirecting through the use of a new message ID.

When designing the initial implementation, it was also proposed the use of a fully distributed system, as that would eliminate the SPOF, the broker, in the architecture. The group drafted a chord design in which the content stored in each node would be a single *TopicQueue* (hashed with its name), and they would replicate it to neighboring nodes. The nodes would be responsible for detecting failures and replicating the queue if necessary. This implementation was discarded due to the handout not restricting the presence of SPOFs, and due to the fact that the proposed teacher's architecture included a central broker.

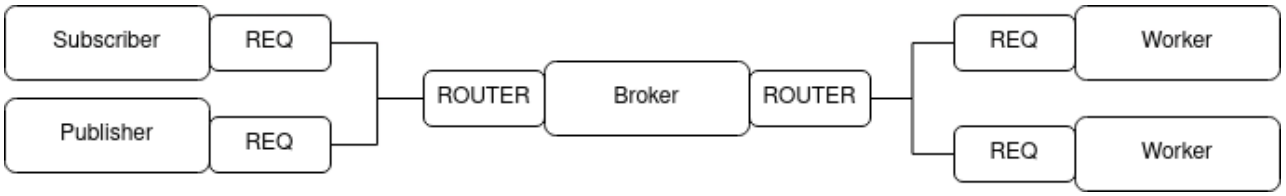
5. Conclusion

The group is satisfied with its final implementation, as we believe to have designed a robust and fault-tolerant approach to the problem at hand. In conclusion, it is possible to assess that the designed implementation guarantees exactly-once delivery in the presence of all failures, except when a broker crashes whilst responding to a client. This statement can be justified by the tests made, which are all within the *config/* folder.

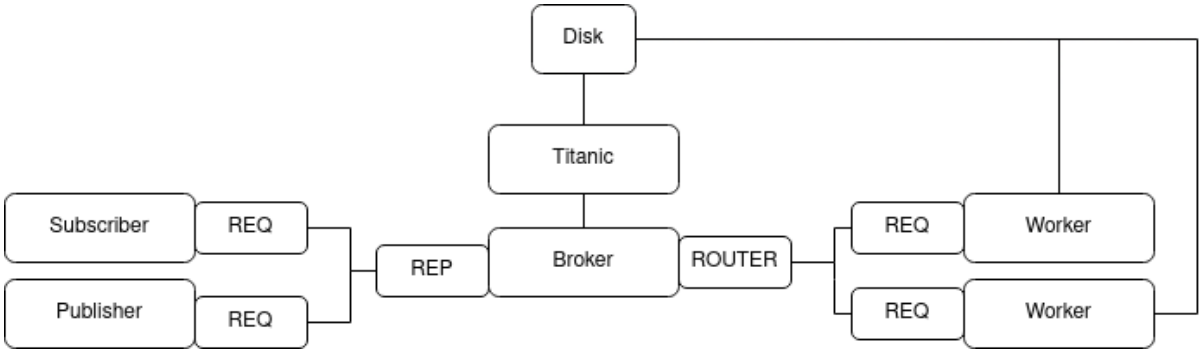
6. References

[1] zmqpp. ZeroMQ high level library in C++. Retrieved from https://github.com/zeromq/zmqpp.
[2] cereal. A C++11 library for serialization. Retrieved from https://uscilab.github.io/cereal/
[3] zguide. ZeroMQ guide. Consulted on https://zguide.zeromq.org/
[4] zeromq API. ZeroMQ Socket Options. Consulted on http://api.zeromq.org/4-2:zmq-setsockopt

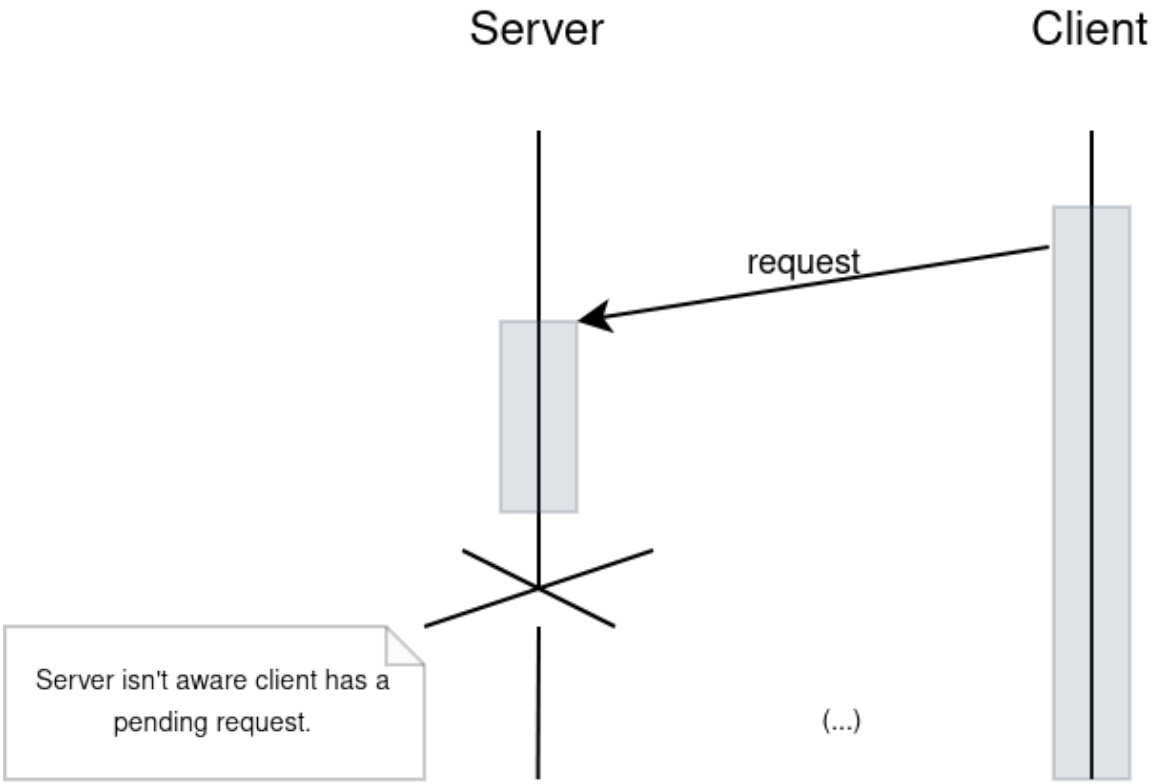
7. Appendix



Appendix 1: Load-balancing pattern



Appendix 2: Titanic pattern



Appendix 3: Client blocking when a server crashes and reconnects