Reliable Publish/Subscribe Service

Large Scale Distributed Systems, 2022/2023

ADRIANO SOARES, ANTÓNIO RIBEIRO, FILIPE CAMPOS, and FRANCISCO CERQUEIRA

The goal of this project is to develop a publish-subscribe messaging service that withstands failures and crashes. We first planned how the communication sequence would occur, in order to study malfunction scenarios, and how to prevent and circumvent them. We chose to materialize this idea using Python and the proposed message oriented middleware, ZeroMQ[1], resorting to its Request-Reply socket utility for any type of communication. File persistence and storage also played a big role in safeguarding the fulfilment of the necessary requirements for the application.

At the end of the project, we achieved a reliable system that guarantees failure tolerance and resilience in different types of crash situations, in both the server and client counterparts (with a few exceptional occasions). This rigorous safety may have come at the expense of overall scalability and concurrency.

ACM Reference Format:
Adriano Soares, António Ribeiro, Filipe Campos, and Francisco Cerqueira. 2022. Reliable Publish/Subscribe Service: Large Scale Distributed Systems, 2022/2023. 1, 1 (October 2022), 5 pages.

1 INTRODUCTION

ZeroMQ is an open-source message-oriented middleware that provides useful amenities for communication and abstracts many of the utilities often used in distributed systems, such as sockets, message queues and brokers. This report describes our development of a reliable publisher-subscriber service to explore the possibilities given by this library, as well as understanding and implementing failure tolerance mechanisms that ensure data consistency and delivery.

2 DESIGN

The system is constituted by a central server, that processes and replies to all incoming requests. Clients can contact the server by performing one of the following operations:

- SUB: subscribe to a topic;
- UNSUB: unsubscribe to a topic;
- PUT: publish to a topic;
- GET: request the next unread publication that is stored in the server.

The server keeps track of all subscribers and publications from a topic by keeping them in persistent storage.

[1]https://zeromq.org/

Authors' address: Adriano Soares, up201904873@edu.fe.up.pt; António Ribeiro, up201906761@edu.fe.up.pt; Filipe Campos, up201905609@edu.fe.up.pt; Francisco Cerqueira, up201905337@edu.fe.up.pt.

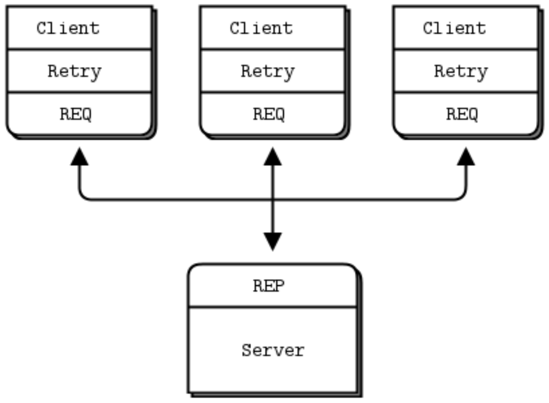


Fig. 1. Lazy pirate communication pattern [1]

2.1 Message protocol

Both requests and replies are critical points of failure for this system since a message can be lost due to a crash or a network failure. In this subsection, we analyse carefully how our protocol handles these common scenarios.

All messages sent from the client (GET, PUT, SUB and UNSUB) include a unique client identifier and a topic name.

2.1.1 SUB operation. To obtain publications on a topic, a client must first subscribe to it. The client must send a SUB message, containing its unique client id and the topic name.

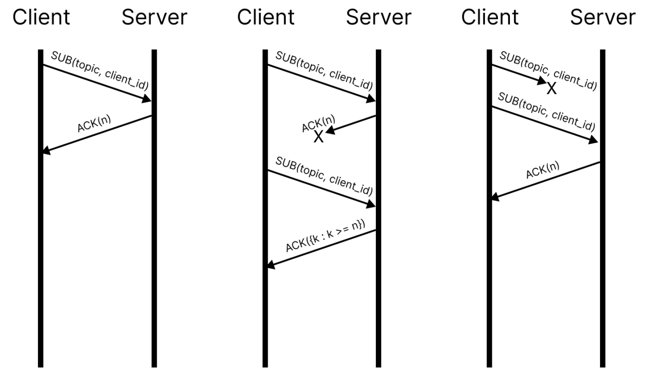
The server should reply with an acknowledgement containing the id of the last publication present on the topic, which will be used to ensure that the client only obtains publications that were created after its subscription. If the server reply is lost, the client should repeat the original request until it gets a valid acknowledgement.

These behaviours are represented in Figure 2.

In the case of "Error on reply" (Figure 2), the k integer (belonging to the ACK of the second SUB attempt) can be equal to n, if it's the same publication id sent on the previously lost ACK, or greater than n, if new publications were added to the topic in the meantime.

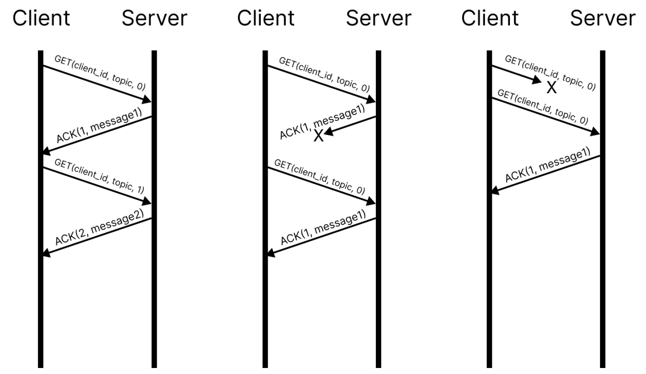
2.1.2 GET operation. The GET request should contain the client_id, the topic name and the id of the last publication received. If this is the first request after subscribing the value of the last publication received should be the value returned in the ACK from the SUB operation. By sending the id of the last publication received we ensure that the client will never receive a repeated publication, as seen in Figure 3.

2.1.3 PUT operation. To publish a publication on a topic, a client should send a PUT message, containing its client_id, the topic name, operation_counter and the publication to be sent.



(a) No errors. (b) Error on reply. (c) Error on request.

Fig. 2. SUB operation sequences.



(a) No errors. (b) Error on reply. (c) Error on request.

Fig. 3. GET operation sequences.

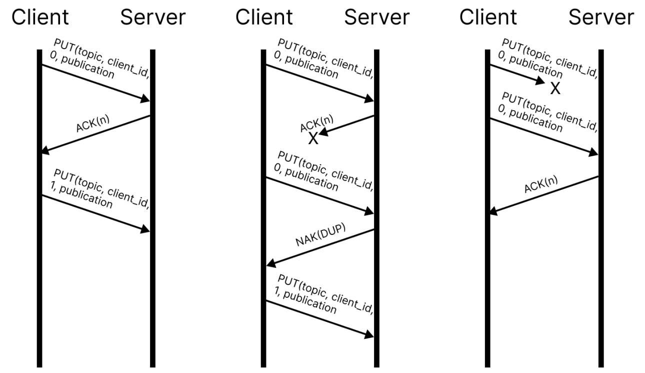
Whenever a client issues a PUT action the lower layer of communication tries to send the request at most three consecutive times. During retries, to ensure that the server does not store/publish the same publication more than one time, due to the client not receiving an acknowledgement somehow, a counter is sent alongside the message. If the server receives the same counter twice it shows that the client did not receive the acknowledgement previously sent and therefore this PUT operation should be discarded. In these rare cases, the server issues a NACK (non-acknowledgment) message with the string "Duplicated message" in its body.

In essence, the main objective of the counter is to distinguish between communication trials in the lower communication layers, it needs to change whenever a PUT operation is sent.

The server will reply with an ACK that contains the id of the current publication, as described in Figure 4.

2.1.4 *UNSUB operation.*

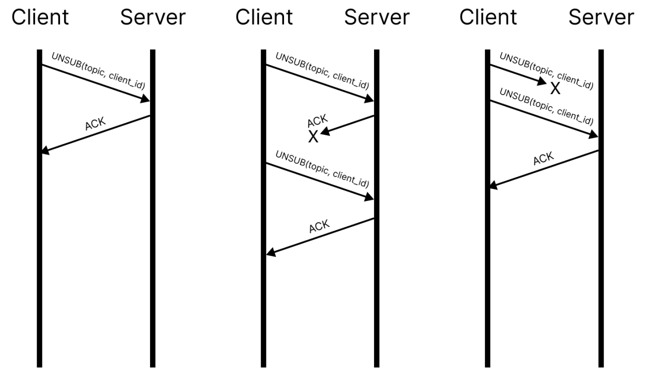
To unsubscribe from a topic, a client should send an UNSUB message containing its client_id and the topic name. The server will reply with an empty ACK.


Client
Server
PUT(topic, client_id, 0, publication
ACK(n)
PUT(topic, client_id, 1, publication
X
NAK(DUP)

(a) No errors. (b) Error on reply. (c) Error on request.

Fig. 4. PUT operation sequences.

This operation is idempotent, therefore, in the presence of failures, the client can try to send the UNSUB request multiple times until it is acknowledged, as seen in Figure 5.



(a) No errors. (b) Error on reply. (c) Error on request.

Fig. 5. UNSUB operation sequences.

3 HOW TO RUN

Make sure the PyZMQ[2] library package is installed. This can be achieved by executing the command *pip install pyzmq* in the command line (assuming Python is installed).

From the project's root:

- Start the server process: *python3 src/server.py*;
- Run the client command line interface: *python3 src/cli.py client_directory*.

The client_dir may already exist as a result of a previous execution, in that case, all the information it contains will be used in the current session. Else, a new directory is created.

In the client command line interface, type the following operations after the "Enter command" prompt accordingly:

[2]https://github.com/zeromq/pyzmq

- GET <topic>;
- PUT <topic> <publication>;
- SUB <topic>;
- UNSUB <topic>;
- EXIT.

4 IMPLEMENTATION

The service was implemented using the Python language and the PyZMQ library wrapper.

Our communication service is based on the lazy pirate pattern [1], which utilizes a simple Request-Reply socket, repeating the message envoy (for a specified amount of tries), and polling until a response is received (with a maximum timeout). If no messages are received during that time period, the attempt fails.

This utility supplied an easy to use abstraction for a reliable messaging transport system that did not constrain the implementation of the fault-tolerant portion of the project, left freely to our choice.

4.1 Message structure

All communications are made through *utf-8* encoded strings carrying JSON objects. They can be split into two groups: the client requests and the server-side acknowledgements (as seen in the figures present in section 2.1).

4.1.1 Client requests. The client requests can be one of the following types: GET, PUT, SUB, UNSUB and REQUEST_ID.

```
{
    'type': ,
    'topic': ,
    'client_id': ,
    'body': {...}
}
```

All requests fill the header attributes "type, topic, client_id", with the exception of the REQUEST_ID messages that have all of the attributes empty. The body field could include an object with the previous read publication id, in case of GET requests, or the operation counter and the publication text, in case of PUT requests.

4.1.2 Server acknowledgements. Both transferring information to the client and confirming the reception of a request are done via server acknowledgements. They are divided into two types, ACK and NAK, depicting the operation's success or failure (e.g. Figure 2).

```
{
    'type': ,
    'body': {...}
}
```

The body section varies according to the context of the operation: a GET operation ACK returns the next unread publication and its id; a PUT operation ACK mentions the current_publication_id (the one that was just added); the SUB operation ACK responds with the latest publication id of the topic.

A NAK simply returns the string representing the error that occurred on the server side and it is the CLI's responsibility of turning it into a warning/error displayed to the user.

4.2 Client Identification

Upon connection, if a client does not have an id stored locally, it requests an id assignment from the server. This process is executed using a REQUEST_ID request, to which the server replies with an ACK containing a universally unique identifier (*uuid*), a 128-bit label unique to each client.

4.3 Storage

Several files are created both in server and client instances to guarantee data persistence and operation consistency between failures.

4.3.1 Client. The client records the counter sent in the latest PUT operation in *counter.txt*. It collects "topic_id, last_publication_read" pairs in *topics.csv*, representing the last publication retrieved from the most recent GET request. Additionally, it keeps the ID retrieved from the REQUEST_ID operation in *id.txt* to ensure a unique client ID throughout client failures.

4.3.2 Server. The server records the latest counters sent in PUT operations in the *client_counters.csv* file with pairs "client_id, counter". It stores the topic subscriptions status in the *topic_id/subscriptions.csv* file with tuples "operation, client_id, last_message_read", in which the field "operation" may be one of the following values: SUB, UNSUB or UPDATE. The UPDATE operation is purely used to distinguish between a SUB operation, improving the overall readability of the file. Finally, it gathers the topic publications in *topic_id/publications.csv* file with pairs "publication_id, publication".

4.4 Garbage collection

To ensure that IO write execution times are minimal we prioritize not rewriting files in their entirety, as many times we only want to change a line. Alternatively, we only append to the end of the file. For instance, whenever a client subscribes, unsubscribes or demands a publication from a topic the server appends to the *topic_id/subscriptions.csv* the corresponding tuple with the respective operation. As a trade-off, we need to implement a method that will not allow the file size to grow indefinitely, hence garbage collection.

The server has a scheduler, implemented using the sched[3] Python package, that executes two types of tasks: the handling of incoming requests and garbage collection. Since the handling of requests is scheduled consecutively (after each managed request) we give higher priority to the garbage collection chore so that it can be executed with certainty every 5 minutes.

We should note that, if the server receives no requests for a long period of time there is no need for this task to be executed as the files were not modified. Therefore, we authorize the server to wait for a request indefinitely, remaining blocked in the receive function if no request has been sent by clients.

The garbage collection task reads through every file and rewrites them accordingly:

- in *client_counters.csv* it removes previous counters for the same client_id, keeping only the latest received;
- in *topic_id/subscriptions.csv* it reads all SUB, UNSUB and UPDATE operations in order, computes a list of the current

[3]https://docs.python.org/3/library/sched.html

subscribers of a topic and proceeds by rewriting the same information with SUB operations only;
- in *topic_id/publications.csv* it removes all publications from the topic that have been received by all clients (note that the server only knows that the publication has been received when a client asks for the next publication).

Since the garbage collection action is very infrequent we should not have as much downtime as to doing a rewrite on every request received due to IO rewriting operations.

Moreover, we make sure to execute the garbage collection task upon a server startup to ensure a "clutter-free" initial state, reducing subsequent downtime due to IO rewriting operations.

5 FAULT TOLERANCE

5.1 Assumptions

The described system is built upon the following assumptions:

- Both the clients and the servers have **stable storage**;
- Each message structure/composition **must follow the templates mentioned in Section 4.1**;
- The client **cannot tamper with its local file structure**, as this could affect the exactly-once semantics, for example, by altering the last publication read id of a topic, causing the client to miss publications that should have been delivered.

Albeit, to further ensure that no side effects can be observed if a client storage tampering occurs, client counters are randomized and do not default into a value (e.g. 0). To demonstrate why that would be an issue envision the following failure scenario:

(1) The user requests "PUT topicA message1" with counter 0 (default value) and stores it;
(2) The user crashes but the server sends an ACK;
(3) The *counter.txt* is deleted;
(4) When the user recovers it defaults its counter to 0;
(5) The user requests "PUT topicB message2" with counter 0;
(6) The server will send a NAK("Message duplicated").

5.2 Exceptional cases

After receiving a PUT request, the server stores the publication and the counter in two distinct files. These file writes occur sequentially, therefore, there is no guarantee that both tasks are executed in unison (atomicity). A server IO exception can occur in both writes invalidating the PUT operation as a whole. Moreover, if it happens after the first write there will be inconsistencies between the file structures and server data structures, jeopardizing the server. To really achieve operation atomicity we could have implemented an atomic commit protocol.

Moreover, there is an exceedingly remote chance that, if the client tampers with its stable storage by deleting the *counter.txt* file, the newly randomized counter sent by the first PUT request being equal to the counter sent in the previous PUT request (before client crash). Consequently, the server rejects this first PUT request (replying NAK("Message duplicated")).

6 POSSIBLE TRADEOFFS

6.1 Design

The server is a critical point of failure and is not scalable. Since this component attends to all client requests and serves as the only record of the topics, their contents and subscribers, any corruption of its state or long-term crash that disabled any type of communication and would end up hindering the network.

Other alternatives we considered were: separating the message broker from the topics storage (using message queues, XSUB and XPUB sockets [2] or Router/Dealer sockets and patterns [3]), placing a central load balancer that could improve the request quantity tolerance (it would remain as a single point of failure and depend on the availability of the other participants). Using the XPUB and XSUB socket pattern would require a change of strategy since the flow of information is not unidirectional (the client socket is instantiated as a gateway for publishing and subscribing). The message framing and composition would also have to follow strict library guidelines.

To take advantage of the Router/Dealer pattern, we had to include the division or replication of the topics (for redundancy) on different sites, rendering the need of some kind of leader election and consistency insurance algorithms. These were not considered the main objective of this project.

6.2 Implementation

All communication is done synchronously, which means a request has to be fully attended to before we proceed to the next one, generating more latency and response delay (especially considering how many IO calls are made). This translates to an operational bottleneck, since one requested instruction stales the whole system, from the perspective of other users.

We considered multi-threading as an option (request thread and worker threads), however, since ZeroMQ guidelines do not advise transferring sockets between threads we choose not to implement this approach.

> *– Don't share ZeroMQ sockets between threads. ZeroMQ sockets are not threadsafe. Technically it's possible to migrate a socket from one thread to another but it demands skill. The only place where it's remotely sane to share sockets between threads are in language bindings that need to do magic like garbage collection on sockets.* [4]

A multi-threading approach regarding file IO operations was also taken into consideration. Still, given that the storage resources are the same, and every request makes at least one read/write action, a *mutex*/semaphore would have to be placed to guard each file access. The time spent waiting for the availability of the file would not take advantage of having a thread(s) dedicated to this purpose.

7 CONCLUSION

Upon completing the assignment, we achieved a sturdy and reliable publish-subscribe service, proven to resist several failure situations, and that reacts in the best way to stop the problem when the participant (server or client) is re-established.

ZeroMQ's ability to abstract communication tools and patterns revealed very useful to provide the foundations of the network system, as well as file persistence and storage.

The significance and wide usage of the publish-subscribe pattern are now clear to us since it is a simple concept, that (when correctly executed) supplies an infrastructure that can attend to a large number of users. It can go from simple information publishing (database type) to more complex event-driven distributed systems, that utilize asynchronous communication to distribute workloads.

Future work on this project could include, making it more scalable (in terms of topic distribution) and amending the exceptional cases, in which the system could subside.

REFERENCES

[1] 2013. *Zeromq. Messaging for many applications.* (1st ed.). O'Reilly Media. Chap. Chapter4:Reliable Request-Reply Patterns, Client-Side Reliability, 144–147. ISBN: 978-1-449-33406-2.

[2] 2013. *Zeromq. Messaging for many applications.* (1st ed.). O'Reilly Media. Chap. Chapter2: Socket and Patterns, The Dynamic Discovery Problem, 45–47. ISBN: 978-1-449-33406-2.

[3] 2013. *Zeromq. Messaging for many applications.* (1st ed.). O'Reilly Media. Chap. Chapter2: Socket and Patterns, Shared Queue (DEALER and ROUTER Sockets), 48–50. ISBN: 978-1-449-33406-2.

[4] 2013. *Zeromq. Messaging for many applications.* (1st ed.). O'Reilly Media. Chap. Chapter2: Socket and Patterns, Multithreading with 0MQ, 63–67. ISBN: 978-1-449-33406-2.