U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Large Scale Distributed Systems

Class 3 Group 13

Diogo Samuel Gonçalves Fernandes	up201806250@fe.up.pt
Hugo Miguel Monteiro Guimarães	up201806490@fe.up.pt
Paulo Jorge Salgado Marinho Ribeiro	up201806505@fe.up.pt
Telmo Alexandre Espirito Santo Baptista	up201806554@fe.up.pt

25th November 2021

SDLE Project - 2021/22 - MEIC

Teachers

Carlos Miguel Ferraz Baquero-Moreno	cbm@fe.up.pt
Pedro Alexandre Guimarães Lobo Ferreira Souto	pfs@fe.up.pt

Index

1 Introduction

This report describes the development of a reliable publish-subscribe service. All the features requested for this project were implemented. The instructions to run the program, also described in the README file, are presented below.

Start by installing all the required python packages running the following command on the terminal:

```
$ pip install -r requirements.txt
```

Run the proxy server, the publisher and the subscriber with:

```
$ make proxy
$ make publisher [id] [ip] [port]
$ make subscriber [id] [ip] [port]
```

It is possible to *subscribe* and *unsubscribe* to a topic using the following commands:

```
$ make sub [subscriber_ip] [subscriber_port] [topic]
$ make unsub [subscriber_ip] [subscriber_port] [topic]
```

To publish a new message to a given topic, run:

```
$ make put [publisher_ip] [publisher_port] [topic] [message]
```

To receive the following *nTimes* messages run:

```
$ make get [subscriber_ip] [subscriber_port] [nTimes]
```

Instead of using make, you can execute the program by running the python scripts *proxy_run.py* *pub_run.py*, *sub_run.py*, which start, respectively, the proxy, the publisher and the subscriber, and *exec_command.py*, which is used to invoke the commands.

This publish-subscribe service guarantees *exactly-once* delivery, even in the presence of communication failures or process crashes. In this report, we will explain our approach to building the publish-subscribe service and how we solved the problems that were described in the assignment.

2 Design

2.1 Proxy

Our program contains an intermediary server that is responsible for managing the communication between the publishers and subscribers, providing the resources as requested. This serves as a method to control the complexity of the requests, besides providing additional benefits such as privacy or security, since it avoids a direct connection between publishers and subscribers.

One of its roles, detailed in section **Storage**, is to store all the data of the system, such as the many existing topics and messages, and the corresponding sequence numbers. Once it establishes a single common entry point, it's possible to scale the number of publishers and subscribers freely.

When the Proxy is initialized, it starts an IOLoop, which is an I/O event loop for non-blocking sockets. This IOLoop was necessary to implement the Reactor Pattern [1] [2] to achieve more concurrency.

The class starts by initializing three sockets of type *ROUTER*, one for the *backend* communication (with the subscribers, for processing the *get* requests), another for the *frontend* communication (with the publishers) and the last one for the message exchange related to the subscribe and unsubscribe operations. Upon receiving a new message, a handler function is called, having each socket its own handler.

The *backend* handler starts by creating an object of the class *IdentityMessage*, which separates the message content, obtaining the following components:

- *identity*, useful for identifying each subscriber
- *key*, which in this case always has the value "GET", since this function only handles the *get* requests
- *sender_id*, so it can detect which topics the client is subscribed to
- *seq*, which is the sequence number of the last message received by the client

Then, it retrieves all the messages not yet received by the client, of all its subscribed topics, and sorts the list so that the messages with lower sequence numbers appear first. If this list is not empty, the Proxy sends the first appearing message, otherwise, it sends an *ACK* message, indicating that there are no messages to receive.

The *frontend* handler also creates an object of the class *IdentityMessage*, and adds a new publisher to the storage if the sender_id in the message didn't exist yet. After this, it checks that the sequence number of the message received is right after the last published message by that sender. If not, a *NACK* message is sent, informing the publisher about the sequence number of the last message received from it. It increments both the total sequence number of the Proxy and the last sequence number received from that publisher. Then, it stores the new message, sending an *ACK* message if the given topic doesn't yet exist or the topic has no subscribers. Otherwise, the Proxy just sends an *ACK* message to inform the publisher of the last received sequence number of the message.

Finally, the handler responsible for subscribes and unsubscribes, after creating the *IdentityMessage*, checks its key, to distinguish between a subscribe and an unsubscribe operation. In the first case, the Proxy creates a new topic if it doesn't already exist, and adds a subscription to it storing the client id. In the second case, it updates the storage so that the client is no longer subscribed to that topic. It finishes by sending an *ACK* message to the client indicating success.

2.2 Publisher

The publisher is the entity responsible for publishing a message on a specific topic, which is done by communicating with the Proxy server using the *put* command. The Publisher has no other responsibilities, as all the complex data manipulation is delegated to the Proxy, as described above.

Many publishers can be running at the same time, and each can publish messages to the desired topic using the *put* operation.

The Publisher communicates with the Proxy using the *Router Dealer* pattern. The Proxy serves as a *Router* and the multiple Publishers work as *Dealers*. This pattern allows bidirectional communication between the Proxy and an instance of a publisher.

In Figure 3 we can see a simple sequence diagram showing the communication between a Publisher and the Proxy. After a successful *put* message, the publisher receives an *ACK* message from the Proxy. We can see it in the first section (1) of the traded messages in Figure 3.

Also, in the case of the publisher sending a wrong sequence number as we can see in section (2) the server sends a *NACK*. The *NACK* message contains the last message sequence number and is used to indicate the need for a resent after message sequence order conflicts were detected. In this case, the Publisher attempts to resend the *put* message with the correct sequence number.

Given that each Publisher waits for an *ACK* before sending the next *put* message and the proxy only accepts messages that are sequential, it is ensured that messages arrive sequentially and the server receives **exactly-once**. If no *ACK* is received, a timeout occurs and the message is resent, repeating the process. In the case of the server receiving two duplicated messages, it will discard one since they will have the same sequential number.

2.3 Subscriber

The subscriber is a client capable of consuming messages from a topic. In order to receive those messages, one must first *subscribe* to the topic. Thenceforth, it will be able to receive any further messages published to the subscribed topic, until the *unsubscribe* operation is called.

There can be several subscribers in our network, and each one can *subscribe* to multiple topics. If a client subscribes to a topic and proceeds to leave the network, all messages published in the subscribed topic can be received once the client rejoins the network and calls the *get* operation.

The subscriber communicates with the Server using the *Router Dealer* Pattern. The Proxy serves as a Router and the multiple subscribers work as Dealers. This pattern allows bidirectional communication between the Proxy and an instance of a subscriber.

In Figure 4 we can see a simple sequence diagram showing the communication between a Subscriber and the Proxy, assuming that the client subscribed to some topics beforehand.

Analysing the section (1) of that Figure, the client starts by requesting a message with the sequence number of the last message received. If this is the first *get* request, the message will have 0 as the sequence number. Upon receiving the *get*, the proxy will check if there are any messages to be dispatched to that client. The message must have been published only after the client subscribed to the corresponding topic and its sequence number must be higher than the requested number. The next message to send is the one with the lowest sequence number, to ensure order.The client will then receive a message from the Proxy server. This message will have a sequence number that will be used for the next requests.

The next time the client makes a request by sending a *get*, he will use the newly updated sequence number, avoiding receiving repeated messages as seen in (2). This helps us to ensure that each message is received **exactly-once**.

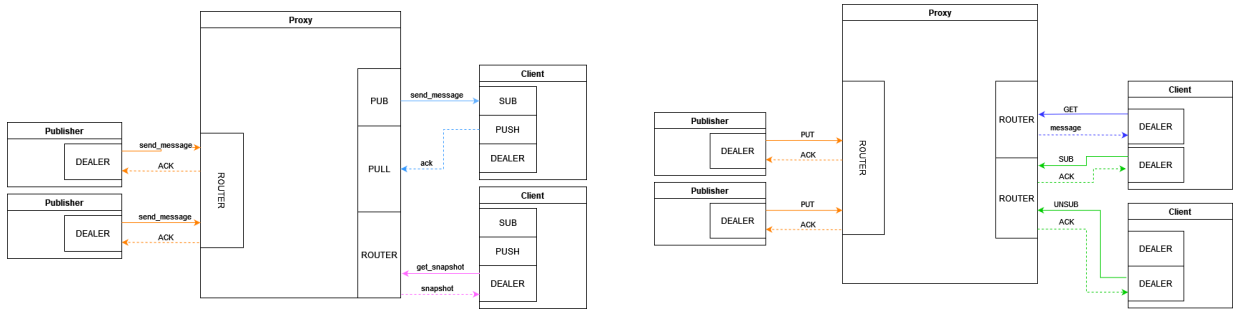
If the client attempts to *get* more messages but none are remaining in the queue, the Proxy sends an *ACK* message to inform the client it should not expect further responses for the *get* request (3).

3 Implementation Aspects

3.1 Architecture

To build our system, multiple patterns were studied to weigh the merits and demerits of each one.

We came across two patterns that fit our objectives, each one catering to different purposes, which can be found in Figure 1.



(a) Clone pattern with Proxy (b) Pattern implemented

Figure 1: Messaging patterns

At first, it was considered developing a Clone pattern [3], like the one shown in (a), with a proxy serving as the middleware between the publishers and the clients. Having a proxy server serving as the common endpoint for both clients and publishers allows the system to scale on both sides (scale both publishers and subscribers) without having to reconfigure subscribers to know the existence of additional publishers.

In this adaptation of the Clone pattern, the proxy serves as the server that binds a socket to serve as the sinkhole for the publishers to post their messages. This socket follows a ROUTER-DEALER pattern achieving a fully asynchronous communication between the publisher and the proxy with full control of the message formats. Upon a connection from a publisher, an identity is shared to the router so that it can communicate with that specific peer, this later allows the proxy to reply to the publisher with status messages of their published messages. With that in mind, the communication between the publishers and the proxy after the connection is established is as follows:

- A publisher sends the message of a given topic
- Proxy upon receipt makes verifications to ensure the message is on correct order and to verify if there are any clients subscribed to that topic
- The Proxy replies to the publisher with either ACK or NACK depending on whether the message passed the checks or not.

On the proxy-client connection, it's used the default Clone design, having a PUB-SUB socket to distribute the messages to all the clients. The client then uses the PUSH-PULL socket to inform the proxy of the status of the message received, replying with an ACK or NACK depending on whether there was an error on the message received or not. Additionally, there's a ROUTER-DEALER socket so that the client upon connection can request a state update to recover messages missed by the client while disconnected or crashed.

This design allows the system to serve a larger scale of clients more easily but brings more complexity to the system to ensure the order of messages and reliability on the exactly-once design, as the queue in the PUB socket would start filling up and removing messages that could be missed by clients that were crashed or disconnected for a long time. As such, we opted to implement the design represented on (b), explained in detail in the previous section, that allowed us to achieve a higher degree of reliability on ensuring exactly-once design and order of messages delivered to the clients,

which was the main objective of our service, as the messages were stored in the proxy and delivered at request to the clients, only getting removed from storage once all clients had been served. This, however, came at the cost of limiting the power of distribution to a larger scale of clients.

3.2 Storage

In order to guarantee reliability in our system, the proxy server needs to store information regarding the topics that are subscribed by a set of clients as well as information about the publishers publishing to that topic.

To solve that issue, we created a ServerStorage that will store information the proxy server needs to maintain. The conceptual model for the proxy server storage can be found in Figure 2.

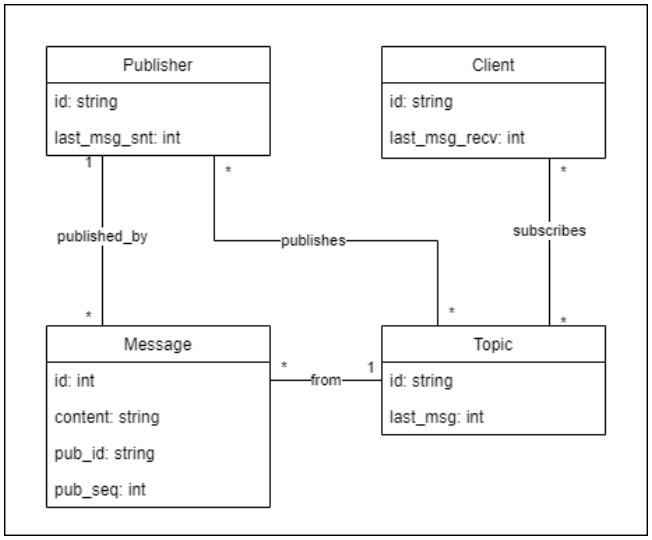


Figure 2: Conceptual Model

The *ServerStorage* implementation changes some parts of the conceptual structure, making it similar to a JSON file opposed to a relational schema. The storage is composed by four components: *topics*, *clients*, *publishers* and *sequence_number*. These components will be explained further in the following sections as well as other mechanisms such as the garbage collector. Additionally, the storage is saved to a file periodically, from which the proxy can restore its state on startup.

Sequence Number

The proxy server defines an independent sequence number to order the messages received from the publishers. This sequence number is needed on top of the sequence number provided by the publisher since we need to handle multiple publishers sending messages on the same topic, and we must ensure order not only from messages of a specific publisher but also between publishers.

Topics

Consists of a list of topics stored in the proxy server. Contains information relative to the topic: identity (*id*) and list of messages. Each message contains its identifier defined by the *sequence_number* given by the proxy server, message content, and publisher identity (*pub_id*).

The sequence number defined by the proxy, as explained above, is used to maintain an order of messages in the proxy independent from its publishers.

Clients

Consists of a list of clients stored in the proxy server. Contains the topics to which the client is subscribed to and each of the subscribed topics is associated with the sequence number of the last message received from that topic.

Additionally to the storage in the proxy server (*ServerStorage*), the client also keeps a small storage to save the sequence number of the last message received from the proxy.

Publishers

Consists of a list of publishers stored in the proxy server. Contains the existing topics and the sequence number of the last message sent by them. This sequence number is relative to the sequence of messages defined by the publisher and not the sequence in the proxy.

Garbage collector

To ensure the storage only holds onto relevant information, we developed a garbage collector system on the proxy server. This garbage collector will remove messages from storage that were already delivered to all the expected clients as they no longer hold any value, as well as remove all information regarding a topic once there are no more clients subscribed to it.

4 Conclusion

Throughout the development of the project, we thought of many different architectures for our system and ended up with a simple and efficient solution that resulted in a reliable publish-subscribe service. This service ensures some important properties, such as subscription durability, that is, a topic's subscriber should get all messages of that topic until it explicitly unsubscribes it, as long as it calls *get* a sufficient number of times. This should also be possible even if it runs intermittently (it might leave the network or crash). Another important property ensured is the "exactly-once" delivery, meaning that after a successful *put* on a topic, every subscriber will eventually receive that message and that after a successful *get*, that message will not be delivered on a later call to get by that subscriber.

Bibliography

[1] Schmidt, Douglas C., 'An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events', 1995

[2] Hintjens, P., *The ZeroMQ Guide*, pp. 265-267, March 2013.

[3] Hintjens, P., *The ZeroMQ Guide*, pp. 241-264, March 2013.

A Sequence Diagrams

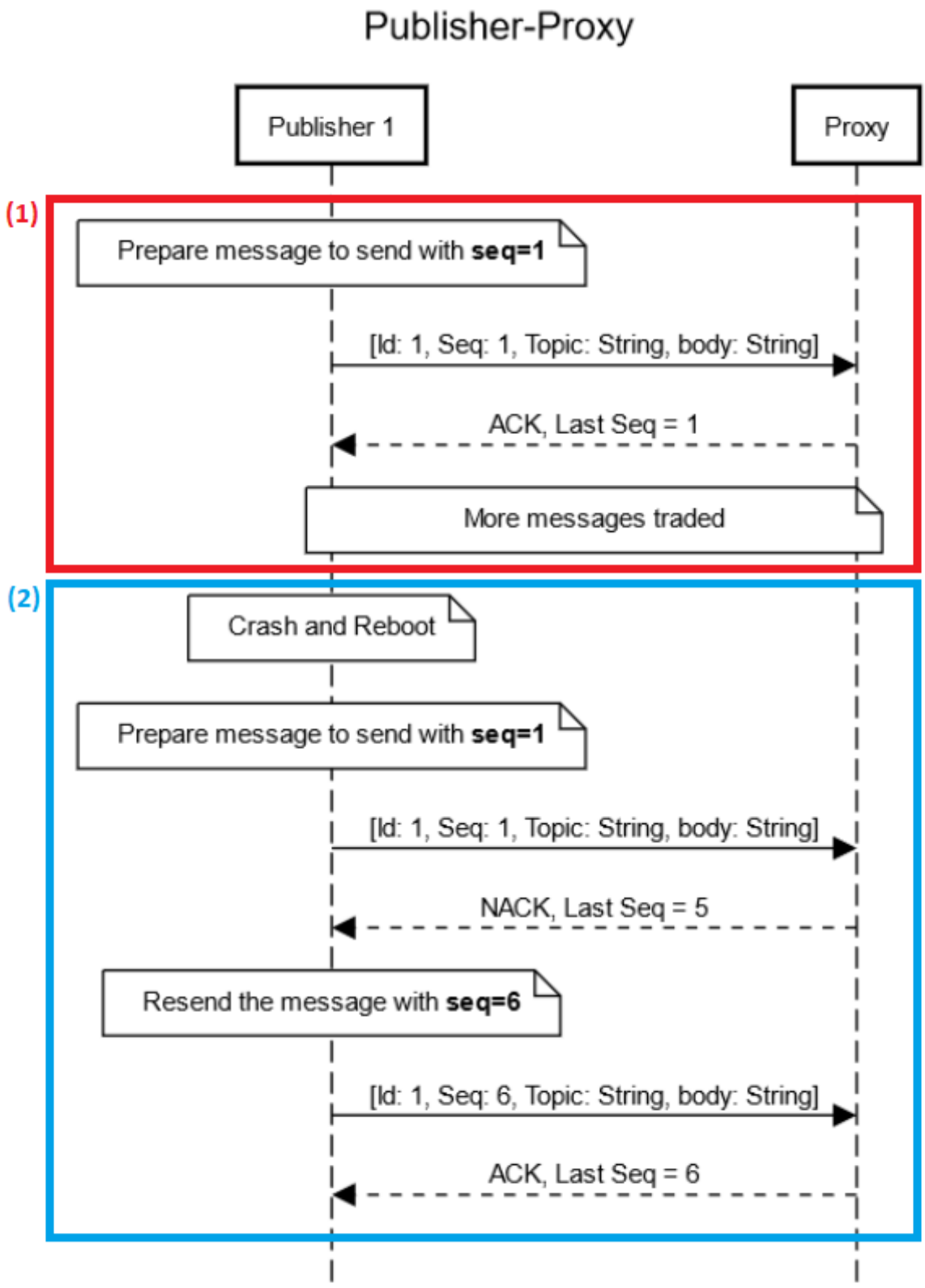


Figure 3: Simple Sequence Diagram between Publisher and Proxy

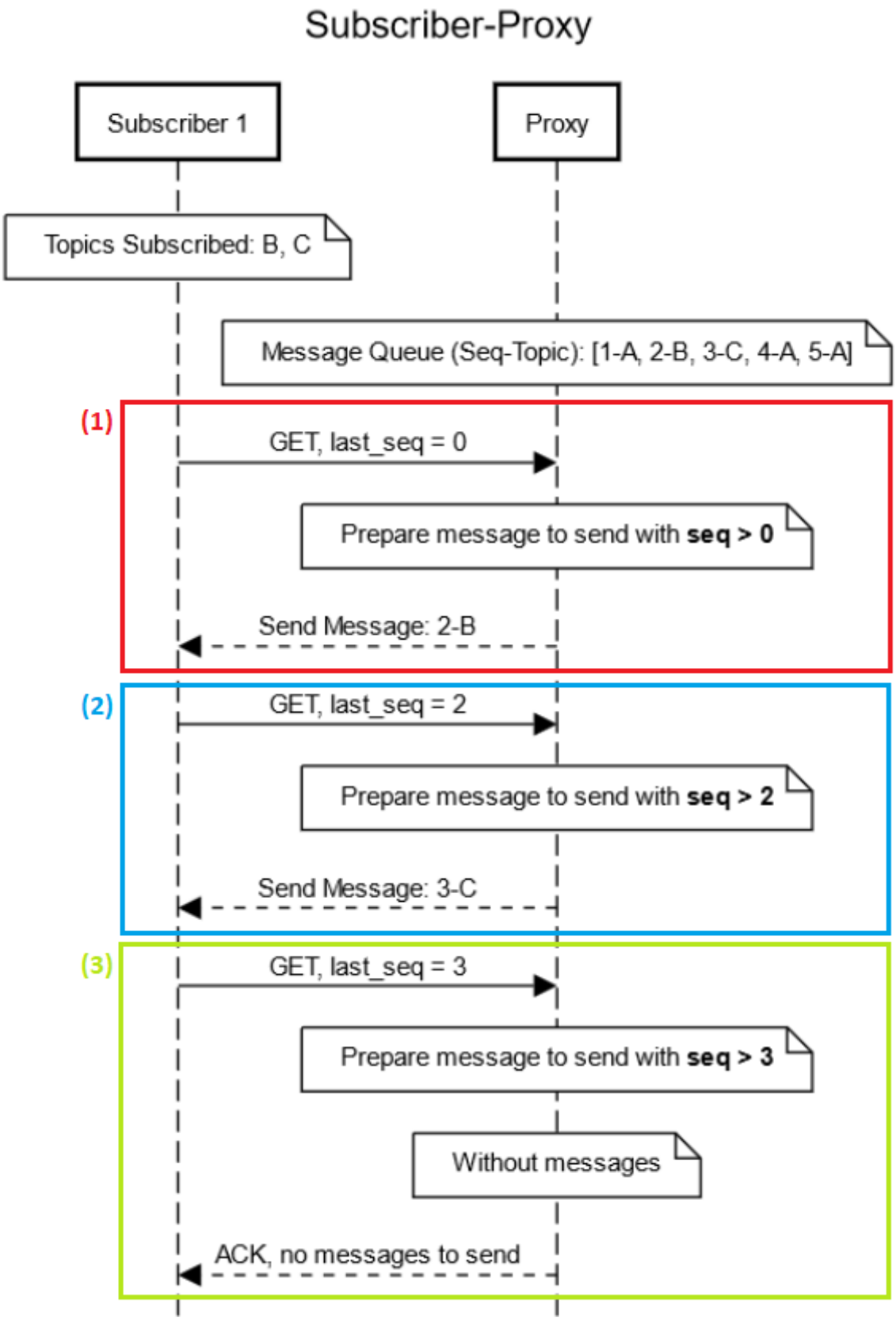


Figure 4: Simple Sequence Diagram between Subscriber and Proxy