Project 1- Reliable Pub/Sub Service

Introduction & Table of Content

Inserted in the context of Large Scale Distributed Systems enrolment, this project intends to build an API for reliable Pub-Sub architecture for message exchange using Zeromq Library as the base ground for socket communication.

The API specification is a handout requirement. With the following structure:

`put()` to publish a message on a topic
`get()` to consume a message from a topic
`subscribe()` to subscribe to a topic
`unsubscribe()` to unsubscribe a topic

Zeromq is a library available in all mainstream programming languages. Its definition, according to Wikipedia, is the following:

> ZeroMQ (also spelt ØMQ, 0MQ or ZMQ) is an asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. The library's API is designed to resemble Berkeley sockets.

- Zeromq already offers an At-Most-Once Pub/Sub out of the box, so our goal is to build over this functionality an API that ensures **exactly-once** functionality.

It wasn't imposed any kind of technological restrictions, so we chose Python as the programming language to implement this project.

Table of Features:

- Reliable Proxy XPub/XSub Pattern.
- Data Retransmission using REQ/REP Pattern.
- Use of Poller in Proxy and Client to handle multiple input sockets.
- Keep state with run time updating of subscribers that received a certain message.
- Runtime delete of message tracking after all subscribers receive it.
- Periodically backups in the disk of the Proxy state.
- Bootstrap load of the proxy state from disk.
- CLI interface to publish new messages.
- Clients have an optional mechanism that simulates runtime errors and subscriptions/unsubscriptions.

- Logger class for enhanced quality CLI tracing with different colours.
- Publishers can publish on different topics in the same execution.
- Clients subscribe to a random sublist of a list of predefined topics.
- Message object with an id that tends to be unique (MD5 hash).
- Scripts for Windows and Linux execution.

Zeromq and Assumptions

XPUB/XSUB Sockets

Since Zeromq use was a project requirement. We intended to use it for our benefit, the main focus was to avoid entering into a reinvent the wheel scenario.

> “If I have seen further it is by standing on the shoulders of Giants.” Isaac Newton-1675

We followed a Newtonian approach to this problem, Zeromq is our giant, it already implements a quality solution for the Pub-Sub problem, but without exactly-once guarantees. Our aim was to over it build a protocol that guarantees that reliability.

REP/REQ Sockets

REP-REQ sockets over TCP offers exactly-once out of the box. We used as much as possible in our protocol definition journey to take advantage of that strong guarantee.

Poller/Event Based Architecture

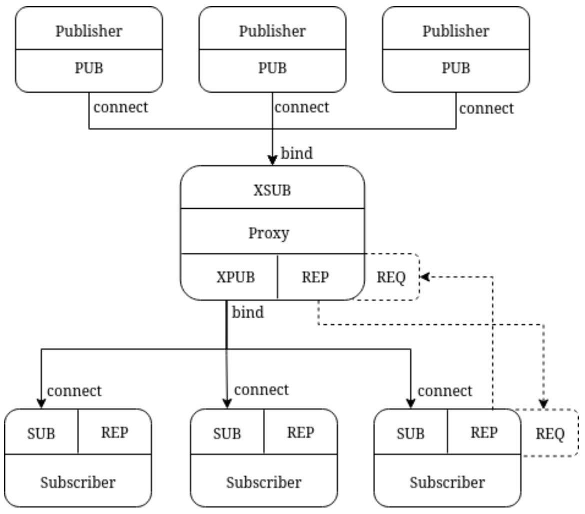
Zeromq ships with a smart poller. That functionality allows efficient polling without a busy wait.

The Poller class allows the code to poll from a list of sockets specified programmatically and unlock the process only when any of them has information to receive. **An event-based architecture.**

The alternative would be a multithread architecture that would introduce the typical race conditions problems. **If we can do it simply, let us make it simple.**

Implementation Details

Our architecture is described in the image below:



The proxy centralizes almost all the logic behind this system. In the proxy can be found 3 permanent sockets:

1. **XSUB** - Receives Messages From the Server (Publisher).
2. **XPUB** - Forwards those Messages to the Subscribers of each Message Topic.
3. **REP** - Used to receive acknowledgement messages from the clients

The publisher and subscribers are simpler to understand. Publisher contains only one zeromq native PUB socket. The client contains one zeromq SUB socket and an additional permanent REP socket whose initial purpose is receiving retransmissions.

Our implementation also uses temporary REQ sockets in the proxy and in the client, which are necessary to operate correctly within the Zeromq ecosystem to initiate communication with REP sockets.

After creating and binding the sockets to their addresses the proxy and the clients create each one a **Poller**. This class provided by ZeroMQ allow the program to efficiently read from the sockets, as mentioned before.

In addition to the traditional Zeromq Pub-Sub pattern we added the following features:

1. New Messages using REP-REQ Pattern Sockets.
2. Track of Messages Flowing Internally.
3. Proxy-State Persistent Storage.
4. Periodically Handler for Message Synchronization.

New Messages

The additional messages introduced were the following:

Type	Sender	Receiver	Frequency	Description
hello	Client	Proxy	On Client Topic Subscription	On subscription to a topic, the client sends an extra message introducing its REP address
ack_welcome	Proxy	Client	After Hello Message	Hello message reply
goodbye	Client	Proxy	On Client Unsubscription	Overcome Zeromq limitations that don't allow efficient identification of unsubscriptions.
ack_goodbye	Proxy	Client	After Goodbye Message	Goodbye message reply
data	Proxy	Client	On Retransmission	Data retransmission; data is inserted in the body of the message
ack_received	Client	Proxy	After data Message	Ack for retransmission message

Internal Message Tracking

The Proxy is also responsible for keeping track of messages and if they've been delivered to every subscriber of the message's topic.
To support the additional Proxy logic behind the functionality described in the paragraph above, we were forced to design a simple data model schema to support the operations.
The final result was two Python dictionaries:

- Clients:
  - Access: `clients.['topic_name']['client_id']`
  - Content:
    - Client REP Socket Address
  - Reason: Clients can subscribe to different topics. Since all messages are identified with `topic` and `origin_id` we can retransmit in O(1) time complexity.
- Messages:
  - Access: `messages.['topic_name']['message_id']`
  - Content:
    - Copy of Message Object
    - List of Clients ids that remain to acknowledge that message.
  - Reason: Messages are organized in topics. Since all data messages are identified with `topic` and `message_id` we can remove a record from the list of clients remaining to acknowledge after an acknowledgement in O(1) time complexity.

Persistent Storage

We implemented a storage system where the information about the clients, respective subscribed topics, and the messages that our proxy has received was stored in disk. In case of proxy's fault, when it restarts, it will load two files: *clients.pickle* and *messages.pickle*, which contain the information formerly saved and make it possible to preserve the state just before its suspension.

Periodically Handler for Message Synchronization

To support operations that have an implicit periodical behaviour, like retransmission and the writing to the disk of the proxy state, we implemented a periodical time-based handler.

This way every three seconds we perform the following operations:

- Retransmit Messages.
- Proxy-State Storage in Disk
- Data Maintenance-Delete records of messages already acknowledge entirely.

Reliability Scenarios

To implement an "exactly once" delivery, the implementation must take into account various scenarios in which the transmission of a message is not able to be completed.

1. Client Failure

We started by addressing the possibility of a client becoming inaccessible to the proxy for an indefinite period, whether because they disappear momentarily, or because they are permanently absent.

With this in mind, when a client first subscribes, a Hello message containing the address of the REP socket in which they must be contacted in cases of retransmission is sent to the proxy.

Thus, after forwarding a new message to the subscribers of said message topic, the proxy must receive an acknowledgement message from each client and if that doesn't happen within a defined period, the message must be retransmitted to the previously stored address. This process repeats itself until the missing acknowledgement message is received.

2. Proxy Failure

Additionally, the possibility of the proxy failing was also taken up. In this regard, for the sake of knowing where and what was running up until an error occurred we implemented the persistent storage functionality mentioned above. Upon its boot, the proxy must recover that persistent data, so that additional fault tolerance to the system can be guaranteed.

Conclusion

We were able to design an API that ensures the requirements specified in the handout. Exactly-once concerns were introduced successfully. Moreover, during development, we always keep in mind the need to design a simple solution with a software quality adequated to fourth-year students skills. We are confident in the result we obtain.

Furthermore, we extended the goals specified for this project, by creating features like the random fault spawning on the client-side and a Logger for enhanced logging and debugging.

Finally, we describe in the table below some dilemmas we faced during development.

Some Tradeoff Identified

Topic	Pros	Cons	Final Decision
ZeroMQ XPUB/XSUB use	In the absence of failures ZeroMQ handles it properly	Build on the top of something we do not control.	Use it
Publisher-Proxy Additional Reliability	A failure on the server-proxy communication always introduces the absence of communication. No violation to exactly-once occur	Pub-Sub is not reliable.	Do not implement any additional protocol on this side
Data Structures based on Python Dictionaries	It allows direct access. We avoid for loops	Lists are simpler to work with	Use Dictionaries
Mechanism That simulates faults in the client-side	It makes it easy to ensure reliability schemas are working and to demo the API.	We easily spam the terminal and lose control of what is going on	We implemented it.
Interface that automatically publishes message	It runs without the input of the user	We easily spam the terminal and lose control of what is going on	Do not implement an automatic publishing mechanism

FEUP 2021 SDLE T4-G12
Academic Project Made By:

- Francisco Goncalves [up201704790@edu.fe.up.pt]
- Inês Quarteu [up201806279@edu.fe.up.pt]
- Jéssica Nascimento [up201806723@edu.fe.up.pt]
- Rúben Almeida [up201704618@edu.fe.up.pt]