U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Reliable Pub/Sub Service

Project 1 Report

Class 7, Group 11

Clara Martins (up201806528)
Gonçalo Pascoal (up201806332)
Rafael Ribeiro (up201806330)

Course: MEIC 1st Year
Curricular Unit: Large Scale Distributed Systems (Sistemas Distribuídos de Larga Escala)
Year: 2021/22

1. Overview

The aim of this project is the development of a reliable publish/subscribe service. This service was implemented on top of `libzmq`, a minimalist message-oriented library. More specifically, we used the PyZMQ library, which offers Python bindings for `libzmq`.

The API supports four basic operations: publishing a message (`put`), receiving a message (`get`), subscribing to a topic (`subscribe`), and unsubscribing from a topic (`unsubscribe`).

Our service supports durable subscriptions, meaning that subscribers will remain subscribed to a topic until they explicitly call unsubscribe, regardless of possible failures. It also provides several fault tolerance mechanisms that help ensure exactly-once delivery in the majority of circumstances.

2. Design

The system we implemented is composed of four main applications: the service itself (`proxy.py`), the publisher client (`publisher.py`), the subscriber client (`subscriber.py`), and the test application (`rpc.py`). Communication is performed through ZeroMQ sockets, and `libzmq`'s polling functionality (specifically the `Poller` class in `POLLIN` mode) is used to retrieve messages from multiple sockets.

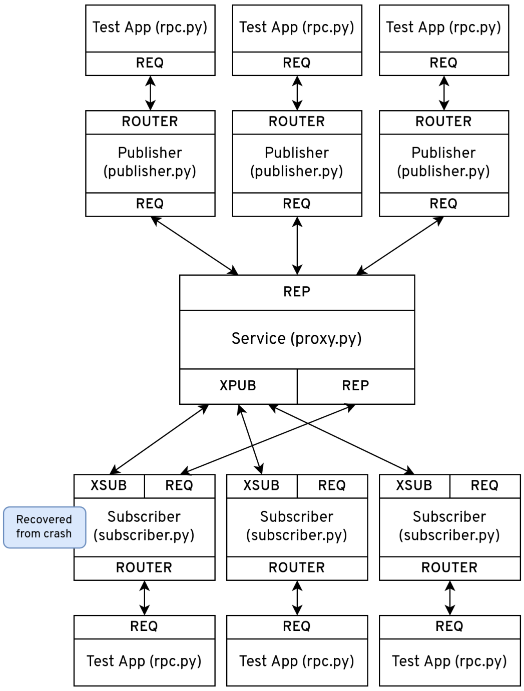

Figure 1: Network Topology Diagram

Figure 1 showcases the socket types that different entities in the network use to communicate, as well as the possible connections between nodes:

- `REQ/ROUTER` - used for communication between the testing application and the publisher/subscriber
- `REQ/REP` - used for communication between the publisher and the service; it is also used between subscribers and the service in case of subscriber failure and subsequent recovery

- XPUB/XSUB - used for communication between the service and the subscribers (reception of published messages and some subscription handling)
- PUSH/PULL - used for inter-thread communication in the subscriber (inproc), in order to redirect responses to get operations to the ROUTER socket where the test application is connected

3. API and Test Application

The testing application we developed interfaces with the client API in a manner similar to Java's RMI (Remote Method Invocation). The method call and its parameters are marshalled (in our case, converted to a JSON string with a specified format) and sent through a REQ socket to a publisher/subscriber, which receives them through its ROUTER socket. After unmarshalling the received data (converting the JSON string to a Python dictionary), the corresponding client method is called and its return value is sent back to the test app.

Using a REQ socket means that the testing application will hold while waiting for a response from the publisher or subscriber, and using a ROUTER socket allows the publisher/subscriber to asynchronously process multiple requests.

The operation to perform, as well as its arguments can be specified through command-line arguments. The valid command-line arguments for the test application are:

- **put** operation: `PUT <publisher_port> <message> [--ip <publisher_ip>]`
  `[-i <iterations>] [-d <delay>]`
- **get** operation: `GET <subscriber_port> <topic> [--ip <subscriber_ip>]`
  `[-i <iterations>] [-d <delay>]`
- **subscribe** operation: `SUB <subscriber_port> <topic> [--ip <subscriber_ip>]`
- **unsubscribe** operation: `UNSUB <subscriber_port> <topic> [--ip <subscriber_ip>]`

For PUT and GET operations, we can also specify that the operation should be repeated a certain number of times, with a delay between each execution (in milliseconds). The subscriber/publisher IP address is also an optional argument, defaulting to localhost to facilitate testing on the same machine.

4. Fault Tolerance

4.1. Durable Subscriptions

One of the specified requirements for our reliable publish/subscribe service is the concept of durable subscriptions: a subscriber to a topic should receive all messages belonging to that topic, as long as it keeps calling get. Subscriptions to topics should remain active regardless of crashes or network issues and can only be terminated by an explicit call to unsubscribe.

When implementing a publish/subscribe service using libzmq, PUB/SUB or XPUB/XSUB sockets seem like a natural fit, as these sockets handle both subscription/unsubscription and multicast delivery of messages to all subscribers of a given topic.

In order to implement durable subscriptions, we first needed to understand the inherent behavior of these sockets in the case of subscriber/service crashes. To achieve this, we took advantage of the fact that XPUB/XSUB sockets expose subscription/unsubscription messages, in contrast to regular PUB/SUB where subscriptions and unsubscriptions all happen under the hood.

We observed that whenever a subscriber becomes unavailable, the service receives an unsubscribe message for each of the topics it was subscribed to. This automatic unsubscribe is done automatically by `libzmq`. Therefore, to ensure durable subscriptions, we made each subscriber keep track of the topics it was subscribed to, periodically storing this information in non-volatile memory. After recovering from a crash, the subscriber will restore its state and resubscribe to all of its topics.

The convenience and functionality provided by the `XPUB`/`XSUB` sockets compelled us to choose this solution over devising our own subscription method, which would incur additional complexity in the delivery of messages to subscribers.

The transmission of messages that were not received during the time is covered in depth in the crash recovery section later in this report. In short, the service will store published messages for topics that have at least one subscription (and periodically writes this information to disk as well), meaning it can deliver messages that were missed by subscribers that recover from failure.

4.2. Exactly-Once Message Delivery

Another assurance that our service must maintain is *exactly-once* message delivery; in other words, each message that is published to a topic shall be delivered exactly once to all subscribers of that topic (no messages are lost and no messages are duplicated). In order to better explain how our service meets these requirements, we split the delivery of a message into two parts, the first being from the publisher to the service and the second from the service to the subscribers.

4.2.1. Publishers–Service Communication

The first guarantee that must be upheld for exactly-once semantics is that a published message will be delivered to all subscribers after a successful call to the `put` operation.

When implementing a basic publish/subscribe service with a proxy, an intuitive solution would be to use `PUB` sockets for the publishers and a `SUB` or `XSUB` socket for the proxy and making the proxy subscribe to all published messages (equivalent to subscribing with an empty string as a topic). However, since `PUB` sockets will drop all messages belonging to a topic when there are no subscribers for that topic, any messages published when the proxy is unreachable (such as during a crash or network failure) would be lost, despite the `put` operation returning successfully.

Therefore, we must use a different kind of socket. One option would be to use `PUSH` sockets in publishers and a `PULL` socket in the proxy. This avoids the issue of dropping published messages when the proxy is down since, in *send* operations, the `PUSH` socket will block whenever none of the nodes it is connected to is available.

Another option would be for the publishers to use `REQ` sockets (which also block in the event of connectivity problems) and the service to use a `REP` or `ROUTER` socket and send acknowledgment messages back to the publisher. When our service receives a message from a publisher, it must store the message in the queues of the topics that the message belongs to. If we used `PUSH`/`PULL` sockets and the service crashed before this processing was complete, the message would be lost. However, if we send an acknowledgment message after this step, the publisher can be certain that the message arrived and was stored by the service.

Inherent to this choice is a trade-off between efficiency and reliability. Using a communication model with no back-chatter (`PUSH`/`PULL`) is certainly more efficient, since sending acknowledgement messages takes time. However, due to this project's focus on reliability, we ultimately opted for `REQ/REP` sockets.

4.2.2. Service–Subscribers Communication

Having designed a reliable way to retrieve and store published messages, we must implement a communication protocol between service and subscribers that is resilient to failures in both parties. Several options can be considered regarding how messages will be distributed to subscribers as well as how lost messages will be delivered to subscribers that recover from a crash.

We considered two distinct methods when implementing the communication between the service and the subscribers, which mainly differ by which party is responsible for delivering messages from a certain topic in response to a `get` operation.

The first method is akin to a request/reply model: subscribers will forward `get` operations to the service, which is then responsible for delivering the next message from the topic specified in the `get` call. Therefore, the service must not only keep track of published messages from each topic but also the identifier of the last message that each subscriber has received, for each topic that the subscriber is subscribed to.

This model has the advantage of reducing the complexity of the subscriber, but it places significant additional strain on the service, both in terms of memory and processing power. It would also require subscribers to send acknowledgement messages so that the service could keep track of which message to send in response to a `get` operation. Given that we had already decided that subscribers would be responsible for keeping track of the topics they are subscribed to when tackling the problem of durable subscriptions, we opted for a different approach to service-subscriber communication, explained below.

In this second method, the service immediately forwards messages to subscribers as soon as it receives them from publishers. ZeroMQ's `XPUB/XSUB` sockets handle the filtering of messages, ensuring that subscribers will only receive messages from topics they are subscribed to. When subscribers receive a message, they immediately forward it to the corresponding topic queues. For more reliability, these queues are periodically saved to non-volatile memory.

We resorted to the `Queue` class, which belongs to the `queue` module of the Python standard library. In addition to being thread-safe, it provides the blocking functionality we need for the `get` operation when no messages for the requested topic are available.

This method shifts the responsibility of answering `get` calls to the subscriber, reduces the load on the main service, and means that, in certain cases, calls to `get` can succeed even when the service is temporarily unavailable. Since the `get` operation is entirely local in this communications model, if messages from the requested topic are present in the queue, no communication with the server will be required to deliver a message.

4.3. Crash Recovery

After determining how to provide a service with durable subscriptions and designing a communications model, we shall discuss the crash recovery mechanisms we implemented to ensure reliability.

Regarding the service, we only need to restore its state (the topic queues and the total number of subscribers per topic) from the disk when recovering from a crash, since ZeroMQ sockets can automatically restore subscriptions for active subscribers. However, we must ensure that these are restored before any queued up messages are sent through the `XPUB` socket, to avoid dropping messages. To implement this, after recovering from a service crash, we exclusively poll the XPUB socket with a timeout (by default 400 ms) before starting to process published messages.

The recovery process for the subscribers is more complex. Like the service, a subscriber will recover its state from a file (the file name contains the subscriber's id, to allow several subscribers to run in the same machine). This state contains information about the topics it is subscribed to, any messages it has received and stored locally, as well as the identifier of the last message received for each topic.

It then restores its subscriptions, as mentioned previously in the durable subscriptions section of the report, by sending subscription messages to the service through its XSUB socket. This means that newly published messages from the topics it is subscribed to will immediately start to arrive, but the subscriber won't begin reading them just yet, since it may have missed messages while it was unavailable. These new messages will simply remain in ZeroMQ's internal queues until the subscriber is ready to receive them.

To retrieve these messages, the subscriber will send a message through a REQ socket to a REP socket in the service, which by default is bound to the port of the XSUB socket plus one. This message contains a mapping of topics to message identifiers (each topic the subscriber is subscribed to maps to the identifier of the last message received from that topic).

Since the service keeps its topic queues ordered by the message identifier, it can employ a binary search to quickly find the first message of each topic it needs to send using the identifier values it got from the subscriber. We then take advantage of the union operation of Python sets to join messages from all topics into a single list, since Python sets don't allow duplicates and a message can belong to several topics. This list is then sent back to the subscriber. Since Python's sets are unordered, we also need to sort the resulting list by ascending order of its identifier. We chose to do this bit of processing in the subscriber in order to reduce the service's load.

We mentioned that we decided to restore subscriptions in the XSUB socket before requesting these messages from the service, therefore it is possible for messages, that were already present in the list of messages sent during the recovery process, to have been delivered to this endpoint. To avoid delivering duplicate messages in response to get operations, we made the subscriber discard any messages received through the XSUB socket whose identifier was less than or equal to the identifier of the last message received from that topic.

5. Scalability

One of the major trade-offs when developing distributed services is the trade-off between reliability and efficiency or scalability. Since the main focus of this project was reliability, we had to make a few sacrifices regarding the scalability of the services. Despite this, we also implemented a few mechanisms that can improve the performance of the system when under a heavier load.

The first of these mechanisms concerns cleaning up message queues for topics with no active subscribers. Storing messages for these topics is ultimately a waste of the processing power and resources of the service. In order to implement this, we need a way for the service to keep track of the number of subscribers of each topic. If this number drops to zero, we can safely delete the message queue for the corresponding topic.

The subscription and unsubscription messages exposed by the XPUB socket are not enough to achieve this: as we mentioned previously when a subscriber crashes, ZeroMQ will automatically deliver unsubscription messages for each of that subscriber's topics. If we used these messages to keep track of the number of subscribers of a topic, the service could delete the messages of that topic if all of its subscribers crashed, even though none of them explicitly unsubscribed from the topic.

As specified in the ZeroMQ documentation, subscription and unsubscription messages consist of a single byte followed by the topic in question. This byte is equal to 0 for unsubscription messages and 1 for subscription messages. Any other values for this first byte are ignored by the subscription system, which means we can define other types of messages.

Thus, we delineated two new messages that a subscriber sends through its `XSUB` socket: a byte value of 2 followed by a topic corresponds to an actual `unsubscribe` call, whereas a 3 byte followed by a topic corresponds to an actual `subscribe` call. We also made sure that these messages would not be sent for redundant calls to `subscribe` or `unsubscribe`.

We also chose to impose a physical limit on the number of messages stored per topic, by default 1000. If this limit is reached, the oldest message in the topic queue will be deleted before inserting a new one. The exactly-once delivery guarantees described previously can only be upheld if the subscribers keep calling `get` and do not crash indefinitely. If a subscriber crashed without recovering and this mechanism wasn't in place, the service would eventually run out of memory, since it would need to keep storing messages forever.

Another memory optimization our service supports is the fact that each message object is only stored once, so even though a message might belong to several topics and be placed in several topic queues, any instances of that message besides the first will simply be references to the original message. These references persist even if the service crashes and recovers, as we will explain later.

As described in the *API and Test Application* section, we used `REQ`/`ROUTER` sockets for communication between the test application and publishers/subscribers. The `ROUTER` socket allows the publisher/subscriber to handle several requests asynchronously. In `subscribe` and `unsubscribe` operations, simultaneous processing is not a requirement since they are simple operations and do not require waiting for an answer from the service. As for `get` operations, the expected behaviour is to block until a message from the requested topic is available, so we used a thread pool (`ThreadPoolExecutor`) to be able to process multiple `get` requests simultaneously. In `put` operations, if a request fails, any subsequent requests will also fail until the service is available, which explains why we did not implement concurrent `put` requests.

Regarding the disk backup of the subscriber and service state mentioned earlier, we used the Python serialization module `pickle` to convert the message queue objects and other state information into a byte stream to save in non-volatile memory. One of the major advantages of this module is that its serialization preserves references to objects (such as the references to messages we previously mentioned). This backup procedure is performed periodically by a background thread.