Reliable Pub/Sub Service

Large Scale Distributed Systems

Faculty of Engineering, University of Porto
Porto, Portugal

João Matos, `up201703884@edu.fe.up.pt`
Miguel Neves, `up201806658@edu.fe.up.pt`
Rui Pinto, `up201806441@edu.fe.up.pt`
Tiago Gomes, `up201806658@edu.fe.up.pt`

Abstract

This report explains how we used ZeroMQ's REQ and REP sockets, along with algorithms like Lazy Pirate to implement the functionality that was required for this reliable publish-subscribe like "exactly-once" delivery, as well as fault tolerance. It also covers how we made use of configuration files to interact with the service and all the details of our implementation.

Keywords: communication, publishing, subscribing, fault tolerance, reliability, ZeroMQ.

1 Introduction

The goal of this project was to create a publish-subscribe service that offered two main operations: **put** and **get**, along with two complementary operations: **subscribe** and **unsubscribe**. The implementation also needed to provide "exactly-once" delivery and be capable of handling failures in the different processes that make up the service. In this report we're going to explain how all these things were done, the implementation details, the decisions we had to make in order to deliver what was required and how ZeroMQ helped us accomplish all this.

2 Design

The design of our Service is based on 3 main types of processes: publishers, subscribers, and the proxy. By publishers and subscribers, it should be noted that they do not follow the structure mentioned in the "Pub-Sub network" topic of the ZeroMQ guide [1]. This is mostly because, even though the publisher can send a message by calling `put`, the subscribers do not get those messages immediately. They still need to call `get` in order to receive messages even if they already subscribed to the topic. So, publishers here are the ones that call `put` to send messages associated with pre-defined topics. Subscribers are the ones that call `get`, `sub` or `unsub` to get a message associated with a topic, subscribe to a topic, or unsubscribe from a topic, respectively. Furthermore, in Fig. 1, we also find a process named proxy. The proxy is responsible for handling all the communication between publishers and subscribers. Every type of communication between publishers and proxy and between subscribers and proxy is made via Request-Reply Sockets [2]. Inside the proxy, we find data structures responsible for storing the received messages and also "pointers" associated with the position of each subscriber in the message queue, to ensure the delivery of different messages upon invocation of the `get` method. Taking another look at Fig.1, the Retry section present in every publisher and subscriber has to do with the Lazy Pirate Pattern [3] which we will further detail in the Algorithms section.

3 Implementation Aspects

To implement this distributed system, we used the Python programming language and the `zmq` library. We started by creating a program for each unit of the service: publisher, subscriber and proxy. We also made use of utility programs and configuration files, as we will explain later on.

3.1 Configuration Files

To facilitate the interaction with the publisher and subscriber programs, we used configuration files, which provide all the commands we want to pass to the programs. For the publisher, an example configuration file can be found in Fig. 2.

Each step in the configuration file represents a sequence of `<number_of_times>` `put` operations about the topic `<topic>`, with the message `<message>`, having a sleep between messages of `<sleep_between_messages>` seconds and a sleep after the sequence of instructions of `<sleep_after>` seconds. Internally, we inserted a sequence number to the contents of the message so that we don't have too many similar messages. Essentially, they follow the `<message>_<sequence_number>` structure.

An example of a publisher configuration file with two sequences of `put` operations can be found in Fig. 3.

In the case of the subscriber, the configuration file can be seen in Fig. 4. In this case, there are three types of instructions: `get`, `subscribe` and `unsubscribe`. The `get` operation has a syntax similar to the `put` syntax in the publisher configuration file. To use the `subscribe` and `unsubscribe` operations, we need to specify a topic and can provide a sleep after. In case no `sleep_after` and no `sleep_between_messages` is passed, the assumed delay is 0.

An example of a subscriber configuration file where it subscribes, gets messages, and unsubscribes to two different topics can be found in Fig. 5.

3.2 Publisher Implementation

The publisher program uses two command line arguments: `--config-file` or `-f`, to specify the configuration file that will provide the commands to be executed, and `--id` or `-i`, to specify the publisher id. After the parsing of the arguments, we setup the connection with the proxy, by creating a `zmq` context, setup a `REQ` socket and establish the connection, providing the proxy's IP and port. To finish the initial setup, we read and parse the configuration file, getting two arrays of instructions: `put_steps`, which contains all the `put` operations to be executed, and `sleep_steps`, with all the sleep times in seconds, which will be used between the `put` operations.

Having all the steps we need, we iterate the `put_steps` and `sleep_steps` arrays and execute the corresponding `put` and `sleep` instruction, using the `inject` function shown in Fig. 6.

Note that we are using `sequence_num` to track the current instruction, and writing it to the local storage when it is updated after the `put`. The `sequence_num` variable protects the program flow against crashes, as it can be used as an index of `put_steps` and `sleep_steps` to recover the program by making it start in the next instruction after a crash, allowing it to be fault tolerant. Furthermore, `sequence_num` is sent in the `put` message header along with the publisher id forming a message id, so that the proxy can find out if the received `put` message is duplicated or not, this same id will be checked by the subscriber to confirm if a message is duplicated.

The `put` function definition is available in the Appendix, in the figure 7. We made use of the Lazy Pirate Pattern, which will be explained in the Algorithms section. We start by building the message, which has the structure `[msg_id, topic, message]`, where `msg_id` contains the publisher id and the sequence number, as mentioned above. After sending this message to the proxy, the publisher waits for an acknowledgement message `ACK`. Because we are using request-reply sockets, the publisher blocks after sending the message, until it receives a reply from the proxy or reaches the timeout value.

3.3 Subscriber Implementation

Similarly to the publisher program, the subscriber starts by reading and parsing the command line arguments, the `--config-file` and the `--id`. It also sets up the connection with the proxy, parses and reads the configuration file. After the parsing, we again have two arrays: `actions`, which contains the type of operation to execute and the necessary metadata; `sleeps`, with all the sleep times in seconds, which will be used between the all the operations.

With all the necessary steps, we iterate the `actions` and `sleeps` arrays to execute the corresponding instructions, using the `inject` function shown in Fig. 8. As with the publisher, we are writing the `sequence_number` to the local storage, to track the current instruction, so that the subscriber can recover after a crash, making it fault tolerant. We are also storing the last received message ids for successful `get` invocations in a dictionary named `last_get_ids`, where its keys refer to a topic and the values refer to, as mentioned, the last received message id for that specific topic.

The `sub` and `unsub` operations are very similar. We start by building the message, with the type of operation (`sub` or `unsub`), the topic name and the message id, which contains the subscriber id and the `sequence_num` of the message. Then, we send the message to the proxy and wait for a reply, like in the publisher's `put` operation. It is important to notice that the `sub` and `unsub` operations are idempotent, that is, if the proxy receives repeated `sub` or `unsub` messages, it will send a normal acknowledgement, not having any effect on the final result.

The `get` operation is a little bit more tricky, because it needs to guarantee the "exactly-once" delivery discussed above. Like all the other operations, we start by building the message, sending it to the proxy and waiting for a reply. In this case, the reply in not an acknowledgement, but the next message of the specified topic. We guarantee that the received message is not repeated by storing the last received message id for that topic in the `last_get_ids` structure and comparing it with the actual received `get` response id. If it is a duplicated id, we simply ignore it and try to poll the server `MAX_RETRIES` times expecting to receive a non-duplicated message. If after that we still can not get a successful response, we advance to the next instruction.

3.4 Proxy Implementation

The proxy program starts by checking if a backup file already exists (in case it's restarting after a crash) and, in that case, it retrieves the backed up data and creates an instance of the `Proxy` class, otherwise, an instance is created where all the data structures are empty. The data structures used by the class are 3 Python dictionaries: a message queue which contains a list of messages for every topic, a second dictionary which registers for each topic the position of each subscriber in the queue for that topic, and a third one which, for each topic, keeps track of the id of the last message sent by each publisher to that topic. Additionally, the constructor also creates the ZeroMQ context, configures the `REP` sockets used to communicate with the publisher and subscriber programs and creates a poller used to check for the reception of messages in those sockets. After creating the instance, the `run` method is called. This function creates an endless loop where we poll the sockets we configured previously.

When we receive a message from a publisher, we decode it using a helper class that will be mentioned in the following subsection and parse it to obtain several elements: the message's id, the topic and the content of the message itself. The id is then split into the id of the publisher who sent the message and a sequential number which tells the order of the messages. After this, we check if the message is a duplicate (using the sequence number) and ignore if it is. If the message needs to be stored we check if we've reached the limit of the amount of messages we can store and delete the oldest ones if we need to. We then reply to the publisher with an acknowledgment message.

When receiving messages from the subscriber, the helper class is used again and the message is parsed to obtain the following elements: message id, message type (`get`, `sub` or `unsub`), topic and subscriber id. Depending on the message type, different actions are performed. For GET messages the program will check the pointer that has been stored for the subscriber who sent the message. If no new message can be found, it is going to reply saying there are no pending messages, otherwise the reply will contain the oldest message that has been received after the subscriber subscribed to the topic. When replying with a message, the pointers get updated and messages that all subscribers have already received get deleted from the queue. If the subscriber has not subscribed to the topic yet, the reply will contain that information and no message from the queue will be sent. For `sub` messages, the proxy program will reply with an acknowledgment message whether the subscriber was already subscribed or not. Finally, for `unsub` messages, the subscriber gets deleted from the structure with the pointers for the queue positions and an acknowledgment message is sent as a reply.

3.5 Utility Programs

The programs make use of common functions which can be found in the `utils.py` file which contains the following functions: `parseIDs`, `atomic_write`, `read_sequence_num_pub`, and `read_sequence_num_sub`. The first function simply checks if the ids provided as command line arguments to the programs contain invalid characters like `"_"` which we use when assembling the messages that are exchanged by the processes and, therefore, can not be present in the provided ids. The second function, like the name suggests, atomically writes a backup file to the file system using the `pickle` library to serialize the data structures used by the processes. Additionally, functions from the builtin `os` module are used. The last two functions are used by the publisher and subscriber programs to retrieve the `sequence_number` and/or the `last_get_ids` (in case of subscriber) of the last message they sent/received from a backup to make sure they don't send/receive duplicate messages when crashes occur.

Furthermore, the proxy process makes use of a class defined in the file `backup_proxy.py` called `Backup` which receives in its constructor the data that a proxy needs to backup and then a background thread is created in order to continuously write this data to the file system using the `atomic_write` function mentioned previously.

Finally, another helper class is defined in the `message.py` file which receives the content of a message and optionally its id. This class simply provides two methods: `encode` and `decode`. Since messages have to be sent as a string of bytes, these functions perform the encoding and decoding tasks for us, which allows us to avoid having duplicate code that is hard to read in every section where a message is sent or received.

4 Possible Trade-offs

While developing the project several choices had to be made, each one with its own advantages and disadvantages. This forced us to evaluate each choice carefully and decide based on what we believed to be the optimal outcomes. First we chose to add a proxy, there were not many downsides to this option since the added scalability provided by the existence of a process that new publishers and subscribers could connect to more than made up for the slight inefficiency that would not exist if the other two programs could talk to each other directly. After that we chose to use `REQ` and `REP` sockets because, in order to provide "exactly-once" delivery, we needed bidirectional communication between the proxy and the subscribers and between the proxy and the publishers. We also chose not to use `ROUTER` and `DEALER` sockets since we did not need the asynchronous functionality provided by these sockets and we needed to be able to talk to the `REQ` sockets in the publishers and subscribers.

Another choice we had to make was where to check for duplicate messages when `get` operations are performed. When failures occur, the subscriber can receive the same message more than once, so we store the message id of the last message of that topic received by the subscriber so that it ignores duplicated messages. We chose not to do this kind of verification in the proxy since in our implementation it would be always necessary to do this verification in the subscriber for a few scenarios. In particular the scenario where the proxy receives a `get`, answers with the content but shuts down before managing to advance the respective subscriber pointer in that topic, as such when the proxy restarts and receives a `get` (that was resent with the same message id) it will send a duplicate message because the pointer had not moved and the subscriber needs to be able to deal with this situation by keeping track of the last message id received for that topic.

It is important to highlight that we choose to make the proxy answer the `get` first and only then advance the pointer, because this way we will have duplicates being sent to the subscriber (but we won't miss messages) if the proxy does the first instruction but not the second because it crashes. Having we done this instructions in the reverse order, the proxy might advance the pointer without answering the `get`, thus a message would be lost, violating the exactly-once principle.

Another trade-off we managed was how often to backup relevant data. For the proxy we decided to do this operation every 0.01 seconds so that it does this often enough as to not lose much data but not so often that performance is highly impacted. In the subscriber we backup relevant data after each message received, and the publishers do the same operation after each `put` they perform.

5 Algorithms

Since it was recommended by our professor, we decided to use the Lazy Pirate Pattern where a REQ socket is polled and reception only occurs once a message arrives, requests are resent if no reply has arrived within a certain amount of time and the program gives up after several attempts to get a reply. This pattern helped us ensure fault tolerance since every request needs to have a reply and a process will eventually stop sending requests instead of getting stuck waiting for a program that may never recover. This also helped with "exactly-once" delivery since we stop re-sending the request after confirming that a reply was received. As mentioned, this pattern was both used in the publisher and subscriber processes.

6 Conclusion

With this project we learned how to build a reliable publish-subscribe service where new publishers and subscribers can be added in a scalable way, while also providing "exactly-once" delivery and fault tolerance. The ZeroMQ library was also a very useful tool that made the implementation much easier and allowed us to focus on other aspects of the project.

References

[1] The Dynamic Discovery Problem, `https://zguide.zeromq.org/docs/chapter2/#The-Dynamic-Discovery-Problem`

[2] Ask-and-Ye-Shall-Receive, `https://zguide.zeromq.org/docs/chapter1/#Ask-and-Ye-Shall-Receive`

[3] Lazy Pirate Pattern `https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern`

[4] Multithreading-with-ZeroMQ `https://zguide.zeromq.org/docs/chapter2/#Multithreading-with-ZeroMQ`

7 Appendix

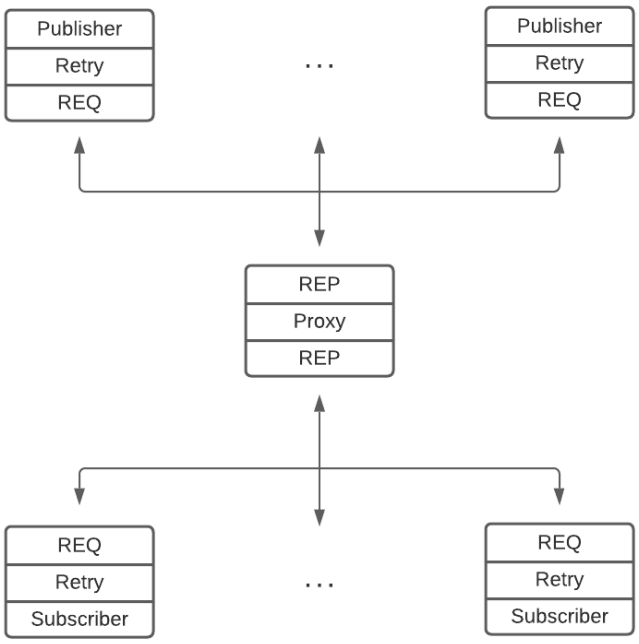

Figure 1: Service Architecture

```
steps:
  - topic: <topic>
    message: <message>
    number_of_times: <number>
    sleep_between_messages: <sleep>
    sleep_after: <sleep>
```

Figure 2: Code

```yaml
steps:
  - topic: topic1
    message: Topic1 Msg
    number_of_times: 3
    sleep_between_messages: 1
    sleep_after: 5
  - topic: topic2
    message: Topic2 Msg
    number_of_times: 5
    sleep_between_messages: 1
    sleep_after: 1
```

Figure 3: Code

```yaml
steps:
  - action: subscribe
    topic: <topic>
    sleep_after: <sleep>
  - action: get
    topic: <topic>
    number_of_times: <number>
    sleep_between: <sleep>
    sleep_after: <sleep>
  - action: unsubscribe
    topic: <topic>
    sleep_after: <sleep>
```

Figure 4: Code

```yaml
steps:
  - action: subscribe
    topic: topic1
    sleep_after: 1
  - action: get
    topic: topic1
    number_of_times: 3
    sleep_between: 3
    sleep_after: 2
  - action: unsubscribe
    topic: topic1
    sleep_after: 2
  - action: subscribe
    topic: topic2
    sleep_after: 1
  - action: get
    topic: topic2
    number_of_times: 5
  - action: unsubscribe
    topic: topic2
    sleep_after: 2
```

Figure 5: Code

```python
def inject(self, puts, sleeps):
    for i in range(len(puts)):
        self.put(puts[i]["topic"], puts[i]["message"])

        # Update sequence_num in publisher file.
        file_path = f"{BACKUP_FILE_PATH}/{self.id}"
        atomic_write(file_path, self.sequence_num)

        time.sleep(sleeps[i])
```

Figure 6: Code

```python
def put(self, topic, message):
  retries_left = MAX_RETRIES

  try:
    msg_id = f"{self.id}_{self.sequence_num}"
    sendMessage = Message([topic, message], msg_id).encode()
    self.req_socket.send_multipart(sendMessage)
    while True:
      if self.req_socket.poll(REQUEST_TIMEOUT) & zmq.POLLIN != 0:
        recvMessage = Message(self.req_socket.recv_multipart())

        [_, response_type] = recvMessage.decode()

        self.sequence_num += 1

        if response_type != "ACK":
          raise Exception("Put request was not received!")
        else:
          print(f"Sent [{topic}] {message}")
          return

      retries_left -= 1
      logging.warning("No response from Proxy.")

      self.req_socket.setsockopt(zmq.LINGER, 0)
      self.req_socket.close()

      if retries_left == 0:
        print("Proxy seems to be offline, abandoning")
        sys.exit()

      print("Reconnecting to Proxy...")

      self.req_socket = self.context.socket(zmq.REQ)
      self.req_socket.connect(f"tcp://{PROXY_IP}:{PROXY_PORT}")
      print("Resending (%s)", sendMessage)
      self.req_socket.send_multipart(sendMessage)
  except (zmq.ZMQError, Exception) as err:
     print(err)
```

Figure 7: Code

```python
def inject(self, actions, sleeps):
    for i in range(len(actions)):
        if (actions[i]["Action"] == "SUB"):
            self.subscribe(actions[i]["topic"])
        elif (actions[i]["Action"] == "UNSUB"):
            self.unsubscribe(actions[i]["topic"])
        elif (actions[i]["Action"] == "GET"):
            self.get(actions[i]["topic"])

        # Update sequence_num in publisher file.
        file_path = f"{BACKUP_FILE_PATH}/{self.id}"
        atomic_write(file_path, [self.sequence_num, self.last_get_id])

        time.sleep(sleeps[i])
```

Figure 8: Code

```python
def sub_unsub(self, prefix, topic):
    retries_left = MAX_RETRIES

    try:
        msg_id = f"{self.id}_{self.sequence_num}"
        message_parts = [prefix, topic, self.id]
        message = Message(message_parts, msg_id).encode()
        self.req_socket.send_multipart(message)
        while True:
            if (self.req_socket.poll(REQUEST_TIMEOUT) & zmq.POLLIN) != 0:
                [_, response] = Message(self.req_socket.recv_multipart()).decode()

                self.sequence_num += 1


                if response != f"{prefix}_ACK":
                    raise Exception(f"{prefix} message was not received!")
                else:
                    action = "Subscribed to"  \
                        if prefix == "SUB" else "Unsubscribed from"
                    print(f"{action} topic: [{topic}]")

                    return

            retries_left -= 1
            logging.warning("No response from Proxy.")

            self.req_socket.setsockopt(zmq.LINGER, 0)
            self.req_socket.close()

            if retries_left == 0:
                sys.exit()

            print("Reconnecting to Proxy...")

            self.req_socket = self.context.socket(zmq.REQ)
            self.req_socket.connect(f"tcp://{PROXY_IP}:{PROXY_PORT}")
            print(f"Resending {message}")
            self.req_socket.send_multipart(message)
    except (zmq.ZMQError, Exception) as _:
        print("An error occurred!")
```

Figure 9: Code

```python
def get(self, topic):
    retries_left = MAX_RETRIES
    dup_msg = False
    try:
        msg_id = f"{self.id}_{self.sequence_num}"
        message_parts = ["GET", topic, self.id]
        message = Message(message_parts, msg_id).encode()
        self.req_socket.send_multipart(message)
        while True:
            if (self.req_socket.poll(REQUEST_TIMEOUT) & zmq.POLLIN) != 0:
                msg = self.req_socket.recv_multipart()
                [resp_msg_id, response_type, response] = Message(msg).decode()
                possible_response_types = ["NOT_SUB", "MESSAGE", "NO_MESSAGES_YET"]
                if response_type not in possible_response_types:
                    raise Exception("Message with invalid type received!")

                if response_type == "MESSAGE":
                    if topic in self.last_get_ids and \
                        self.last_get_ids[topic] == resp_msg_id:
                        print("Ignored response: ", \
                            [resp_msg_id, response_type, response])
                        print("Retrying GET...")
                        time.sleep(3) # Delay send()
                        dup_msg = True
                    else:
                        dup_msg = False
                        self.last_get_ids[topic] = resp_msg_id
                else:
                    dup_msg = False
                if not dup_msg:
                    self.sequence_num += 1
                    print(f"Response: {response}")
                    return

            retries_left -= 1
            if retries_left == 0:
                if dup_msg:
                    self.sequence_num += 1
                    print("Proxy can't seem to given a message other than duplicated.")
                    return
                else:
                    print("Proxy seems to be offline, abandoning...")
                    self.req_socket.setsockopt(zmq.LINGER, 0)
                    self.req_socket.close()
                    sys.exit()

            self.req_socket.setsockopt(zmq.LINGER, 0)
            self.req_socket.close()
            self.req_socket = self.context.socket(zmq.REQ)
            self.req_socket.connect(f"tcp://{PROXY_IP}:{PROXY_PORT}")

            if not dup_msg:
                print("Reconnecting to Proxy...")
                print(f"Resending {message}")

            self.req_socket.send_multipart(message)
    except (zmq.ZMQError, Exception) as _:
        print("An error occurred!")
```

Figure 10: Code