U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Large Scale Distributed Systems

Enforcing Exactly-Once Semantics in a Load-Balanced Pub/Sub Service With Multiple Proxies

Catarina Pires
up201907925@edu.fe.up.pt

Diogo Costa
up201906731@edu.fe.up.pt

Francisco Colino
up201905405@edu.fe.up.pt

Pedro Gonçalo Correia
up201905348@edu.fe.up.pt

October 23, 2022

Abstract

In this project, we present a reliable load balanced publisher-subscriber service on top of the ZeroMQ library. Our service enforces exactly-once semantics and is robust to faults, except in some rare circumstances. The service supports four operations: subscribe, unsubscribe, get, and put, which can be executed by a client. We consider that we designed and implemented this system correctly, and its features are working as intended.

1 Introduction

In a reliable publisher-subscriber service, it is important to ensure exactly-once semantics as much as possible, even in the presence of faults, in order to avoid undesired duplication or loss of messages.

We aimed to implement those semantics in a load balanced service in which each server is the sole responsible for a different set of topics. The communication between clients and servers is mediated by a fixed number of proxies, whose addresses are known to the servers and the clients. A proxy dispatches the messages received from the clients to the respective server, based on the topic, assigning new topics to servers when needed.

Our implementation used the JeroMQ[1] implementation of the ZeroMQ message oriented library as a base to implement the service communication with a REP/REQ mechanism.

In Section 2, we describe the operations supported in more detail. In Section 3, we specify our communication protocol. In Section 4, we discuss how the system behaves in the presence of faults and in which rare circumstances it fails. In Section 5, we note some details of implementation and in Section 6, we conclude.

2 Service

The service supports four operations: subscribe, unsubscribe, get, and put. All clients are able of execute any of the operations.

[1]`https://zeromq.org/get-started/?language=java&library=jeromq#`

2.1 Subscribe

May be invoked with `./gradlew client --args="<ID> subscribe <TOPIC>"`. The client with the given ID subscribes to the given topic, so that it can start receiving messages for that topic. Only messages processed by the service after processing the subscription can be received. On a successful return, it is guaranteed that the client will be subscribed. If an error is returned, it is still possible, but not guaranteed, that the client has become subscribed.

2.2 Unsubscribe

May be invoked with `./gradlew client --args ="<ID> unsubscribe <TOPIC>"`. The client with the given ID unsubscribes from the given topic. On a successful return, it is guaranteed that the client will be unsubscribed and it will become unable to receive any messages from this topic, even if they were sent before unsubscribing. If an error is returned, it is still possible, but not guaranteed, that the client has become unsubscribed.

2.3 Get

May be invoked with `./gradlew client --args="<ID> get <TOPIC>"`. On a successful return, a message from the given topic is received and will not be received again in the future by the client with the given ID. In case of an error, no message is received nor lost, and information about the error is returned.

2.4 Put

May be invoked with `./gradlew client --args ="<ID> put <TOPIC> MESSAGE_PATH"`. Reads the file in the given path and publishes a message with its content to the given topic. On a successful return, all subscribers of that topic whose subscription was processed by the service before processing the `put` will eventually receive the message if they keep calling `get` and do not `unsubscribe` from the topic. On an unsuccessful return, it is still possible, but not guaranteed, that the message was sent.

3 Communication Protocol

We defined a protocol to permit the communication between the clients, the proxies and the servers. This protocol assumes that the number of proxies is fixed and that their addresses are known beforehand. Each server is assumed to have the address of every proxy written in its configuration file, while the client is assumed to have at least the address of one proxy.

All messages sent require a response to be given within 100 milliseconds, otherwise the sender will retry up to a maximum of three times and then give up.

When a service operation is invoked in a client, the client tries to establish connection with one of the proxies whose address it knows. If it cannot establish a connection, it will try the next proxy in its configuration file and so on, until either establishing a connection or running out of proxies to try. In the latter case it aborts, while in the former it will send a message to the proxy with the respective operation.

When a proxy receives a message from a client, it uses the message topic to determine which server it should dispatch that message to. When no server is responsible for that topic, the proxy will assign the topic to the server that is responsible for the least number of topics. In case of a tie, the proxy gives preference to the smallest server ID in lexicographical order.

Periodically, the servers will send a message to the proxies with the topics they are subscribed to, so that proxies may update their mapping from topics to servers. If a proxy detects that two servers are responsible for the same topic, it will send a message to the server with higher ID in lexicographical order so that it transfers its data on that topic to the server with lower ID.

Each server receives messages from proxies or, in the case of transferring topic data, from other servers. When a server receives a message, it executes the appropriate operation.

3.1 Message Format

Each message must have a header and may optionally have a body. The header always starts with the message name and terminates with the sequence of control characters `CR LF CR LF`, denoted by `CRLF CRLF` below. Every header always includes the ID of the sender, which is unaltered by the proxy when it is redirecting a message from a client. It may also include other fields separated by spaces.

3.1.1 Subscribe Message

`SUBSCRIBE <SENDER_ID> <TOPIC> CRLF CRLF`

This is the message to execute the `subscribe` operation for the given topic. It is originally sent by a client.

3.1.2 Unsubscribe Message

`UNSUBSCRIBE <SENDER_ID> <TOPIC> CRLF CRLF`

This is the message to execute the `unsubscribe` operation for the given topic. It is originally sent by a client.

3.1.3 Get Message

`GET <SENDER_ID> <TOPIC> <LAST_MESSAGE_ID> CRLF CRLF`

This is the message to execute the `get` operation for the given topic. It is originally sent by a client. The message also includes a field with the message ID from last successful `get` response this client received on this topic, so that the server does not resend that message with that ID. If the client never executed a `get` before, this field will be the string "`-1`". The server guarantees to never create a message with this ID for a string.

3.1.4 Put Message

`PUT <SENDER_ID> <TOPIC> <COUNTER> CRLF CRLF <MESSAGE>`

This is the message to execute the `put` operation for the given topic. It is originally sent by a client. This message includes a counter that the client guarantees to be unique in every execution of the `put` operation, which means it is only repeated when trying to resend a message after a timeout. The body includes the content to `put`.

3.1.5 Periodic Message

`PERIODIC <SENDER_ID> CRLF CRLF [<topic1> ...]`

This message is sent periodically by a server to each proxy. Its body contains all topics this server is responsible for, separated by spaces.

3.1.6 Merge Message

`MERGE <SENDER_ID> <TOPIC> <OTHER_SERVER_ADRESS> CRLF CRLF`

This message is sent by a proxy when it detects that two servers are responsible for the same topic. The proxy sends the message to the server with the higher ID in lexicographical order and includes the server with the lower ID in lexicographical order in the field. This message will trigger a transference of the topic data from the server that receives it to the other server.

3.1.7 Transfer Message

`TRANSFER <SENDER_ID> <TOPIC> CRLF CRLF <TOPIC_DATA>`

This message transfers the data of the given topic from the sender to the receiver. It is sent by a server to another server. The body starts with the list of subscribers, with one subscriber in each list, and then includes the messages of this topic separated by the special character '`*`'.

Each subscriber line contains the subscriber ID and the IDs of the topic messages that this subscriber has yet to receive, separated by spaces.

Each topic message starts with its ID, followed by `CRLF` and its content. The content escapes the special character '`*`' with '`/s`', and '`/`' with '`//`'.

3.1.8 Status Message

`STATUS <ID> <STATUS> [MESSAGE_ID] CRLF CRLF [MESSAGE]`

This response message may report either success in executing the operation or a error that has occurred. It is sent by the receiver of one of the messages described in the previous sections. In the case of responding to the `get` message, this message includes an additional header field with the ID of the received topic message, as well as its content in the body.

4 Fault Tolerance

We designed our system to be tolerant to crashes and network problems. In order to handle possible loss of state of a client or a server during a crash, their state is saved in persistent storage, which is updated before reporting success on any operation. The file system organization is described in more detail in Section 5.1. The server will load this state on start, while the client only loads the relevant state for executing the operation being invoked. If the server fails to write to the file system, it tries to revert the changes to the state in memory and replies with an error. If the client fails, it will not make a request in the first place.

In order to ensure exactly-once semantics, the client saves the state, for each topic, of the last message ID received in a successful `get`, as well as a counter incremented on every `put` invocation. Every time the client sends a Get message, it includes the last ID received in a successful `get` for that topic, if any, so that the server may delete this message from the subscriber's queue before sending the next message. This ensures that the server will not delete a message that the client did not receive, but also that it will not send a duplicate of the last message the client received. Every time the client sends a Put message, it will send a unique ID based on its `put` counter for that topic, so that the server knows that this message is new. The client will only ever send the same ID twice if it is retrying to send the message after a timeout. This ensures that, on a successful return, the message has been `put` on the topic. However, if an error is returned or the server fails to respond, the service will not guarantee that the message has been `put` on the topic: it is possible that the server registered the `put`, but it is also possible that the operation failed. However, in any case, it is guaranteed that the server will recognize duplicates, thanks to the `put` counter, and only execute the `put` once.

Since our service allows for multiple proxies, there are other possible error scenarios. For example, due to a proxy crash, it may lose the state of which servers to dispatch the topics to. On the other hand, two clients may concurrently create a new topic using two different proxies, who assign this topic to different servers. Our service does not guarantee that two different servers cannot have data on the same topic at a given time, however it does guarantee that this situation will be detected, and those types of conflicts are eventually resolved. This happens because each proxy can detect those conflicts thanks to the periodic messages and may send a message to the conflicting servers so that they send their data on that topic to the server with the lower ID in lexicographical order.

Unfortunately, in the event where the two servers with conflicting topics start to diverge too much in the data they have stored, since it is not possible to know whether the subscribers of one server have subscribed before or after any given `put` in the other server, the merging of the data may lead to those subscribers not receiving messages from the `puts` in the other server. However, this issue does not occur in the single-proxy case. Even in the multiple proxy case, it is a rare circumstance and the servers will quickly remove the conflict.

If a server becomes unavailable, all topics for which the server is responsible may also become unavailable until it comes back up.

5 Implementation Details

5.1 File Systems

Each server creates a folder in order to organize its stored data, with a subfolder for each topic the server is responsible for. Each topic folder contains a subscribers and a publishers file and a *messages* folder with the messages that were put in the topic. Server's file system organization presented next.

```
server_{server_id}
└─ {topic_name}
    ├─ subscribers
    ├─ publishers
    └─ messages
        └─ {message_id}
```

Each client creates a folder in order to organize its stored data. For each topic, a new folder is created having the topic's name and containing a file with the counter of the last message `put` by the client and a file containing the id of the last `get` message. Client's file system organization presented next.

```
client_{client_id}
└─ {topic_name}
    ├─ last_get_id
    └─ last_put_counter
```

There is a configuration file outside the client or server's specific folder, which lists the address of each proxy in each line.

5.2 Periodic Messaging Thread

In order to allow the servers to send their periodic messages to the proxies without interfering with listening to other messages, the periodic messages are sent from a separate thread. This thread is a loop that sends a message to every proxy and then sleeps for five seconds.

6 Conclusion

The goals of this project were achieved. All features were properly designed and implemented, and are working as intended. A special care was put on fault tolerance, particularly in enforcing exactly-once semantics. We consider our service robust, since it only fails to uphold its guarantees in rare circumstances, which were described in Section 4.