Assignment 1 - Group 16

Introduction

In this assignment, we were tasked to design and implement a reliable publish-subscribe service. Using it, a user must be able to put a message in any topic, so that other users subscribed to it can get those messages. Aside from that, the service should guarantee "exactly-once" delivery and handle possible failures.

Implementation

The technologies used in the development of this project were the JAVA programming language and the JeroMQ library, a java implementation of ZeroMQ.

The team decided to use this object-oriented language because of its portability, security and ease of use. Java also has a very extensive user library and is very well documented. Beyond that, theres is a large community of developers and students around this language, the team considered this to be a major advantage since this makes more resources and suport available to the team members and flattens the technology's learning curve

Regarding the network messaging functionalities, the team used JeroMQ, a pure java implementation of the ZeroMQ open-source universal messaging library.The benefits of using this library rely on it being open-source high performance library that provides a messaging queque without the need of a dedicated message broker. It has asynchronous I/O engines which are used to carry atomic messages in applications and offers support for numerous patterns such as publisher-subscriber, request-reply and client-server, making it a perfect fit in concurrent or distributed systems

Server

The server is implemented as an instance of a runnable class **ConcreteServer** that implements the **Server** interface. The **Server** interface only has two methods specified, *receive()* which returns a **ReceivePair** and *send()* which returns void.

A **ReceivePair** is a record with two fields, an address (byte[]) and a message (Message), the address being the address of the client that sent the message.

The server uses a *ROUTER* socket to receive requests and send messages from and to clients. We used a *ROUTER* socket instead of a *REP* socket because this way we could receive multiple requests before responding since when a *REP* socket receives a message it goes into a state where it can only reply and not receive more messages. The *ROUTER* socket could also allow us to route incoming messages to worker threads through *inter-process communication* if we wanted, but we chose to use **Runnable** handlers and an **ExecutorService** thread pool instead.

Incoming messages are handled through instances of subclasses of **Handler**, an abstract class that implements the **Runnable** interface. Each type of message the server is capable of handling has its own **Handler** subclass, which is instantiated after the server receives the message. The **Handler** instance is then submitted to the **ExecutorService** which handles task scheduling.

Client

The client is implemented as an instance of a runnable class **ConcreteClient** that implements the **Client** interface. The **Client** interface has seven methods specified:

- *send()*: to send a message to the server;
- *receive()*: to receive a message from the server;
- *get(String topic)*: to get a message from a specific topic;
- *get()*: to get a message for each of the topics the client is subscribed to;
- *put()*: to publish a message to a topic;
- *subscribe()*: to subscribe to a topic;
- *unsubscribe()*: to unsubscribe to a topic.

The client uses a *REQ* socket to send requests and receive replies.

For that, it makes use of timeouts and, as such, in the event of communication becoming unavailable for a certain amount of time, the client will time-out after some time, 10 seconds, after which it will attempt to communicate again, by resending the message to the server.

Messages

The **Messages** are the communication between Server and Clients and are divided into 2 categories: **serverMessages** and **clientMessages**.

All the specific **Messages** extends the **Message** abstract class.

The **clientMessages** are messages that the client sends to server, in this case, represents all the requests sent to the server.

There are 5 types of **clientMessages**:

- *GetMessage*: Consume the last unread message of the specified topic from the server;
- *GetMessageAck*: Acknowledge having received a requested message from the server;
- *PutMessage*: Publish a message on the specified topic in the server;
- *ShutdownServerMessage*: Shutdown the server;
- *SubscribeMessage*: Subscribe specified topic;
- *UnsubscribeMessage*: Unsubscribe specified topic.

The **serverMessages** are messages that the server send to client, in this case, represents all the replies sent to client.

There are 5 types of **serverMessages**:

- *NotSubscribedMessage*: Serves as an acknowledgment message that tells to the client that are not subscribed to a topic;
- *ShutdownReplyMessage*: Serves as an acknowledgment message that tells to the client that the server is shutdown.;
- *PutReplyMessage*: Serves as an acknowledgment message that guarantees to the client that the message was put on the topic.;
- *SubscriptionReplyMessage*: Sends the result of subscribe/unsubscribe operation. The responses are: **SUBSCRIBED** if the client is now subscribed to a topic, **UNSUBSCRIBED** if the client is not subscribed to a topic and **ERROR** if occurs an error;

- *TopicArticleMessage*: Sends a message from a topic to the client. Usually is the reply of a get operation.;
- *AckMessage*: Acknowledge the confirmation that the client has received a requested message;
- *EmptyTopicMessage*: Serves as an acknowledgment that tells the client there are no messages available for them to get from a topic;
- *NonExistentTopicMessage*: Serves as an acknowledgement that tells the client the topic they requested a message from doesn't exist.

Design

The application followed a client-server architecture. The main class **ConcreteServer** runs the main server and includes the necessary functions and handlers to treat incoming requests. The main class **ConcreteClient** allows a user to get or put messages and subscribe or unsubscribe to topics. This is done by typing simple messages in the terminal (ex: `get [topic]`).

Failing Circumstances

Exactly-once

In a **put** request, the following happens, sequentially:

- A client sens a *PutMessage*, with the topic and message;
- The server will, if necessary, create the topic and then place the message in it, replying to the client with a *PutReplyMessage*

Since the topics and its messages are saved on non-volatile memory, thus, even with a server failure, no information will be lost, and, thus, ensure that as long as a previous subscriber calls the **get** method enough times, it will get that message.

In a **get** request, the events are as followed:

- A client sends a *GetMessage*, with the topic in which they want the message from;
- If the client is subscribed to the topic, the server replies with a *TopicArticleMessage*. If it is not subscribed, it will send a *NotSubscribedMessage*. If the topic doesn't exist, it will reply with a *NonExistentTopicMessage*;
- Should the client receive a *TopicArticleMessage*, it will then send an *GetMessageAckHandler* to the server. Otherwise, it won't send anything else;
- Finally, having sent a message before, the client will then receive an *AckMessage* from the server.

In the server, not only are the subscribers for each topic saved, but also the last message they have gotten from it, which will only be updated after a client acknowledges that it received a message from a topic.

Since the last message the client received is saved and since the client acknowledges it when it receives one, meaning no re-transmits of the same message, we can ensure that the client won't receive the same message after having received it successfully before.

However, there are a few rare circumstances where exactly-once would not be guaranteed:

- If the server crashes after receiving a *PutMessage* and handling it, but before it sends a reply to the client, the client will then resend the request, thus resulting in a duplicate message in the server, which,

for the client, will seem as if it will receive the same message twice in a row in eventual future **get** requests.
- If a client crashes after receiving a message requested from the serve, but before acknowledging it to the server, the server won't have updated the last message read from the client and, thus, when the client sends another **get** request, it will received the same message as before.

Race Conditions

In situations where two server threads were trying to access the same topic a race condition could occur, leading to data inconsistency. To avoid this we marked the functions that would access the topics as *syncrhronized*, making sure only one thread may execute them at a time.

Contingency plan

Whenever a new topic, message or subscriber is added to the system, the server saves that data on non-volatile storage. As such, were a server to crash, it wouldn't lose any previous data, aside from anything being currently handled.