First Assignment Message Broker

Professors: Diogo Gomes, Nuno Lau
Subject: Computação Distribuída

Work done by:
- Daniel Gomes nºmec 93015
- Mário SIlva nºmec 93430

Pub/Sub Protocol

As our base to complete this assignment, we used the Publish-Subscribe protocol, where the publishers (Producers) publish messages to certain topics, publishers are loosely coupled to subscribers (Consumers), and need not even know of their existence, and every subscriber that subscribed to those topics, will receive those messages. It was implemented an intermediary entity, a Broker, with the function of forwarding the messages to the subscribers and prioritize messages in a queue before sending them, whenever a message was published or when they connected with the Broker (sending the last Message) encoded in Serialization Mechanism since they can each have different mechanisms. It's present as well a middleware that abstract all producers/consumers from the communication process. This protocol provides the opportunity for better scalability than traditional client-server, through, for example, message caching.

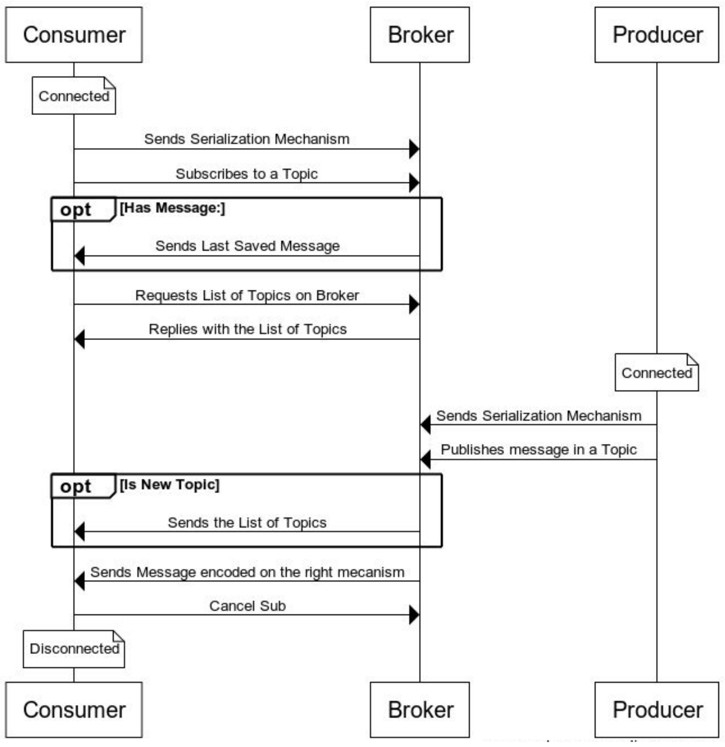
Data Structures

- ❏ **usersdict**: socket as a key, serialization mechanism as a value. By doing this, we manage to save each user's mecanism value in order to know how to encode/decode each message to/from this user.
- ❏ **topicmsg**: topic as a key, and as a value a dictionary with both users and messages for that topic. In this data structure we manage to get the subscribers and the last message saved of the topic. Example: {'/root': {'messages': [], 'users': []}, '/root/weather': {'messages': [], 'users': []}, '/root/weather/temperature': {'messages': [], 'users': []}}

Both of these structures are present in the Broker. By saving as cache, this information makes it easier not only, to get information in broker for most of the operations, but also avoids losing more time to obtain information.However, our space complexity increases. As we used regex in order to do the data processing in Broker, using data structures like ***topicmsg***, makes it more simple to solve the problem at inserting data, listing all topics and also canceling subs, but for hierarchy it requires more resources since we iterate all topics and use regex. Note that we use "/root" before every topic created, so that when someone subscribes the root ("/"), it subscribes all topics with regex (subscribing to the root , "/", is not the same as subscribing to a topic called "root", because "/"="/root" and "/root"="/root/root").

Flow/Software Architecture

During the execution of this assignment, the flow of messages between the presents entities obtained the following form, as we can see on the Message Sequence Chart. These flow of messages will be explained during the course of this report.



Consumer
Broker
Producer
Connected
Sends Serialization Mechanism
Subscribes to a Topic
opt [Has Message:]
Sends Last Saved Message
Requests List of Topics on Broker
Replies with the List of Topics
Connected
Sends Serialization Mechanism
Publishes message in a Topic
opt [Is New Topic]
Sends the List of Topics
Sends Message encoded on the right mecanism
Cancel Sub
Disconnected
Consumer
Broker
Producer

Middleware

In order to know which serialization mechanism is supported by each entity on the middleware side (Producer and Consumer), after each queue establishes a TCP connection with the Broker, it’s sent an message with its mechanism as a String to the Broker before any other message is sent. After this, the Middleware knows how to encode/decode the messages because these functions to encode and decode data of the different mechanisms have the same name on the class representative of the mechanism. Same it’s applied to Broker, because it will have registered the mechanism of the socket and will know which operations to use.

To distinguish each type of messages that are sent by a consumer/producer, each message is encoded with an method (Example: PUBLISH, SUBSCRIBE), topic (Example : If we want to subscribe a certain topic, cancel an specific subscription or publish a message into a topic) and last, but not the least, the message itself. This allows to make a generic type of “protocol” for encoding and decoding messages. We decided to do the subscription of the topic passed as command line argument everytime we run a new consumer, because we could not get the same socket if the connection with the Broker ended.

Broker

The **readPubSub** function is one of the most important functions on the Broker side. As it would make the code repetitive for some similar parts in read/write operations, we decide to only use one function that is able to know whether its a publish message or subscribe. If its a publish message then it will be passed as an argument a *msg*, else the msg will be “None”. To iterate we used regex so that we could split the topic passed as argument and concatenate them one by one on each iteration, which works out as an really quick way to do the publish operation because the length of the “for loop” will always be dependent of how many elements the split method gave us and we can just insert messages right away on the right place. While iterating we pass a list of users of the topic to its subtopics (avoiding always any duplicate information). Important to refer that we used a FIFO Queue for the messages of each Topic, so that we could not only save the last published message but also to avoid that two producers send a message at the “same” time (first message arrived will be sent first, and then popped out of the list). The worst part of using regex to implement this, it's in the Subscribe operation due to the fact that obliges to iterate the topicmsg dictionary to see if the topic that the consumer wants to subscribe contains any subtopics using regex, which can be a really slow operation if the dictionary has a considerable dimension. For this case, a better approach would be using a non binary tree implementation for the topics avoiding an O(n) time complexity in searching operation.

List Topics and Cancel Subscription Operation

Our approach to list the Topics consisted in sending all current topics (not including the root) present on Broker after the first subscribe operation and everytime a new topic was created. By doing this, allows the consumer to get to know the other topics that exist in the “channel” at the time of the first subscription and also if any subtopic of a topic that the user subscribed had already been created; the users would receive that information before any message of a possible new subtopic. To achieve this we created a method in the Middleware that sends a message to the broker asking for the initial list of topics, the broker responds with a string with all the topics (avoiding interruptions if it sended the topics individually), after that, it’s the broker’s responsibility to send the list of topics everytime a new topic is created.

As we used Python Sockets to allow the required TCP connections, when a socket closes it’s impossible to send any information to it afterwards, and as a consequence, our implementation required to remove from our Broker’s Data Structures, those sockets (avoiding to send to a closed connection). All of these points were the base of our Cancel Operation. We decided to cancel a subscription in two possible scenarios:

- If we want to cancel a certain subscription, we only remove the socket from that topic and, of course, it's subtopics, using once again regex;
- If we close a connection we remove simply the closed socket from all data structures.

To test these possible scenarios, we printed on the Broker side the data of the topicmsg dictionary. In the last scenario, using a KeyBoard Interrupt Exception in the consumer, it was triggered the cancelSub function on the Middleware side passing the consumer topic as argument. Important to refer that even if the second scenario occurs, we still remove all socket entries from the Broker data structures.

Solution Validation Strategy

After implementing our approach to this problem, we recurred to some tests, to check if everything was working the way it should be. To test the variable serialization mechanisms we used, for example, consumers with XML serialization Mechanism and Producers with JSON Mechanism. The scalability of our solution, was tested by running a consumer that subscribes to the root ("/") and other 2 producers that published for the topic "/weather" and "/msg", and as result, we manage to receive the messages. The FIFO Queues seem to be working correctly because if a new consumer subscribes to a topic where had been sended messages, then this entity will receive the last message.