BIG DATA MANAGEMENT

Spring 2022 - Programming Assignment

Project by:

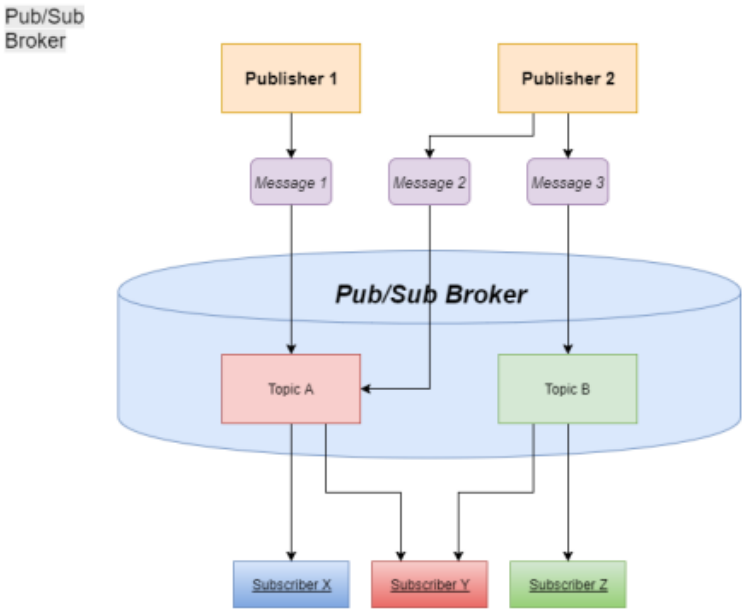
Konstantinos Panoutsakos

Student ID: 7115152100012

e-mail: cv13805@di.uoa.gr

1. General Scope of the Project

In this project we will be creating a simple version of a Publish-Subscribe (Pub/Sub) system, similar to Apache Kafka or RabbitMQ. We will implement a simple version of those systems. More specifically, we will deploy three components: a Publisher that can run with multiple instances, a Subscriber that can also run with multiple instances and a Pub/Sub Broker that will be handling the message relaying between Publishers and Subscribers. Our architecture will be similar to that of the following picture:



More specifically, let's dig into the basics of this architecture with some help provided by https://cloud.google.com/pubsub/architecture.

Pub/Sub is a *publish/subscribe* (*Pub/Sub*) *service*: a messaging service where the senders of messages are decoupled from the receivers of messages. There are several key concepts in a Pub/Sub service:

- *Message*: the data that moves through the service.
- *Topic*: a named entity that represents a feed of messages.
- *Subscription*: a named entity that represents an interest in receiving messages on a particular topic.
- *Publisher* (also called a producer): creates messages and sends (publishes) them to the messaging service on a specified topic.
- *Subscriber* (also called a consumer): receives messages on a specified subscription.

The service that we created, was in its simplest form, meaning that the Topic and the Subscription are the same thing. Thus, each topic corresponds to a single subscription, and the subscribers that are enrolled in it are stored based on the topic itself.

2. Resources - Deliverables

The basic concept of the proposed solution was to develop 3 sub-programs, one for each task. (Broker - Publisher - Subscriber). Regarding the source - code - editor, all three programs were developed in Visual Studio Code 1.66.2, and the programming language of choice was Python 3.7. All tests of the communication between these three programs were executed through the PowerShell.exe command-line tool, by starting a Windows PowerShell session in a Command Prompt window. To prevent automatic activation of any other selected environment, we added "python.terminal.activateEnvironment" : false to the settings.json file of VS Code. Finally, for a less restrictive execution environment and just for the scope of this project, the execution policy was set to remotesigned. So the deliverables of the Project include

- A broker.py file
- A sub.py file
- A pub.py file
- A README.txt file that provides all the necessary insight regarding the required arguments given as input by the user when executing the above programs
- The current Report

3. Architecture of the System - Broker

The Broker connects the Publishers and the Subscribers together. Its job is to remember all the topics that each subscriber has subscribed to and, when a publisher publishes something about a specific topic, it should forward the message to all subscribers that have subscribed to that topic. The broker does not store messages from publishers for future use, meaning that each message is only forwarded to the subscribers of the respective topic at the time of the receipt, and then dropped. Let's analyze some key features of the program:

- Broker uses a dictionary to store all the subscriptions of the connected subscribers in a key - value form. Every key represents the topic (it is mandatory to start with #) and each value is the fingerprint of all subscribers subscribed to the topic, stored as a list of lists. More specifically, each subscriber corresponds to a list that includes his sub_id and the socket that connects the broker with the specific subscriber. For example a dictionary of the subscriptions may look like:
{'#hello': [], '#world': [['s01', <socket.socket fd=548, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9090), raddr=('127.0.0.1', 40001)>],
['s01', <socket.socket fd=352, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9090), raddr=('127.0.0.1', 43000)>]],
'#great': [['s01', <socket.socket fd=548, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9090), raddr=('127.0.0.1', 40001)>],
['s01', <socket.socket fd=352, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9090), raddr=('127.0.0.1', 43000)>]]}.

In this dictionary, there are 2 subscribers connected to the socket '127.0.0.1': 9090 that serves the hearing of the subscribers from the broker. These two subscribers are enrolled to topics #world and #great. Mind that there is also the topic #hello that does not include any subscriber. This happened because this dictionary is constantly updated meaning that if a subscriber unsubscribe from a topic he will be erased from this dictionary, though the topic will remain. Moreover, if a connection is lost (e.g the subscriber suddenly terminates his program) all his subscriptions will be erased, but again the topic will remain in the dictionary.

- In order to tackle the problem of servicing multiple subscribers (or publishers - I will refer just to subscribers but same rules apply to publishers), first of all, each time a connection of a subscriber with the broker is established, its credentials (socket and address) are stored in a list called sub_con_adr (or pub_con_adr for publishers). Then, since we have just one

socket to hear multiple users, the select library is exploited as a direct interface to the underlying operating system implementation. More specifically, in a daemon thread, we define a perpetual iteration of all the sockets stored in sub_con_adr (or pub_con_adr in another thread for the publishers). Then, using select, we can monitor the activation of each socket by giving its subscriber's socket the chance, in a tiny time interval (0.00001 secs) to see if the corresponding subscriber had send a message to the broker. Therefore if no activation is initiated, the next socket is immediately checked and so on. And since sub_con_adr/pub_con_adr are constantly updated in other daemon threads, we can secure that when each iteration is finished (each iteration lasts only a couple of μs), the next iteration will include the renewed list of subscribers/publishers.

- So the final concept was to initiate 4 daemon threads, 2 for the publisher and 2 for the subscriber. One thread (for each case pub or sub) renews the list of the connected subscribers (or publishers), and the other thread iterates over this list to constantly hear from them (receives messages from the publishers to deliver them to the subscribers or receives messages from the subscribers to renew the subscriptions list).
- Finally, into the thread of the publisher that iterates over the list of the connected publishers, when one of them sends a formalized message to the broker, then another function is activated that iterates over the subscriptions list and, depending on the current subscriptions, delivers the corresponding message to all the subscribers concerned.

4. Architecture of the System - Publisher and Subscriber

These programs' architecture has a double goal:

- Receive (either from the automated commands procedure of the command file or manually from the user) formalized commands which are transformed into messages to the broker. So a perpetual loop of the main program waits an input from the user and when this is received, it is transmitted to the broker. Then the program is waiting again for some inputs from the user and so on.
- Receive from the broker all the messages that the broker is intended to send to them. With regard to the publisher, this is only associated to the answers of the broker related to the publications of the publisher. As regards the subscribers, it is associated to both the answers of the messages of the subscribers to the broker and to the forwarded publications regarding the topics that the subscriber is subscribed as well. In order to achieve that, a daemon thread is deployed and its only usefulness is to constantly hear the broker.

5. Conclusion

Another part of the whole system was to be as user friendly as possible. Therefore, in addition to the minimum requirements regarding the messages that should be printed in the users' terminals, more cases were discovered and the respective messages are displayed. Moreover, to support the usability of the system, some complementary minor delays have been added to certain parts of the programs, so that the user has the time to read messages that are printed to his terminal. These delays do not affect at all the functionality of the program.

This is also the thinking behind the way many exceptions were handled, using try statements. We tried to catch many exceptions and treat them accordin to the scope of the system. Some (but not all) of these cases include:

- The user can not initiate each program if he does not provide the right arguments as input, and he is repeatedly asked to do so if he does not.
- There is a gentle exit of the pub/sub program if a ConnectionAbortedError occurs due to reaching maximum capacity of the system (5 subs or 5 pubs connected).
- The formalization of the messages provided by the pub/sub is mandatory and, if not followed, leads to the printing of the appropriate messages in both broker's and user's (pub or sub) terminal
- All the 4 treads of the broker.py are daemon threads and ,combined with an os.system("pause") line of code at the end of broker.py, give the user the chance to terminate the whole system with the press of any button (The user of broker.py should not press any button in any case since there is no need of him to provide any input during the execution of the system).
- If a subscriber/publisher quits and kills the connection with the broker, an appropriate message appears to the broker's terminal and immediately a spot of the subscribers/publishers list is released. Thus, in case the limit of 5 users was reached, a new publisher/subscriber is now capable of connecting to the broker.