Licenciatura em Engenharia Informática - Computação Distribuída

Docentes:
Prof. Diogo Gomes & Prof. Nuno Lau

Projeto 1
Message Broker

Rui Fernandes, 92952

DETI
Universidade de Aveiro

04 – 2020

1 Introduction

The goal of this project was to develop a Message Broker that could connect Producers and Consumers that use different serialization mechanisms (JSON, Pickle, XML).

For this, two files were created, *broker.py* which is the Message Broker itself, it receives serialized data, unpacks and works with it, then sends if necessary data to Consumers accordingly serialized. The second file, *middleware.py*, works as an abstraction layer to facilitate the Consumer's and Producer's use of the broker.

In addition to that, a PubSub protocol (based on TCP) was defined and utilized to comunicate between the broker and middleware.

2 Data Structures

In terms of the middleware it's important to mention that four classes exist: a base class *Queue*, and three subclasses of *Queue* pertaining to each serialization. The base class contains most methods and some "skeleton" methods that are redefined in the subclasses.

They are all initialized with a *topic* that they subscribe or publish to, and a *type* (Consumer or Producer). Consumers and producers should create objects of the subclass that is relevant to them (JSON, Pickle, XML) and use the methods provided to interact with the broker, never creat an object of *Queue* as it's meant to be a sort of abstract class.

In the broker there are three dictionaries used to store information:

- *conn_types* → stores serialization mechanism type of each consumer and producer (json, pickle, xml).
- *consumer_topics* → stores the topics associated with each consumer.
- *last_entry* → stores the last published entry of each topic, also used to usefull to know the existing topics in the broker.

3 The Protocol

Every message of the created protocol has a mandatory *"op"* field that corresponds to the operation to be performed, and two potential fields, *"topic"* and *"value"* that are needed for certain operations.

A brief explanation of the structure and goal of each operation will now be provided:

- *"subscribe"* → sent from the middleware's method *sub* to the broker by queues of the consumer type, it contains the field *"topic"* which is the topic the consumer pretends to subscribe to. When received by the broker, if it's the first time the user subscribes, an entry will be added to *consumer_topics* composed of the consumer's socket and a list with the single topic provided. On further *"subscribe"* messages, if a new topic is provided it will be appended to the consumer's topic list in *consumer_topics.* After, the broker checks if the subscribed topic has an entry in last_entry and if so sends the associated data to the consumer.

- *"publish"* → sent from the middleware's method *push* to the broker by queues of the producer type, it contains the fields *"topic"* and *"value"* which are the topic to publish on and the content of the publication respectively. When received by the broker it will iterate the consumer's in *consumer_topics* and if any of the consumers are subscribed to the given topic or any of it's parent topics (topics are organized and treated as a tree structure), the broker will send the data to that consumer using the

send_to method. After, the broker stores/updates the given topic's entry on *last_entry*.

- *"unsubscribe"* → sent from the middleware's method *unsub* to the broker by queues of the consumer type, it contains the field *"topic"* which is the topic the consumer pretends to unsubscribe from. Similar to the *"subscribe"* message, it removes the given topic from the consumer's topic list in *consumer_topics*, if reduced to zero topics the conection with the consumer is closed to avoid potential problems associated with having no subscriptions.

- *"list"* → sent from the middleware's method *list_topics* to the broker and it's used to list the existing topics in the broker (by convention the ones that have been published on). Upon receiving the message, the broker iterates the topics in *last_entry* and sends each one to the consumer that requested the listing with the *send_to* method (an empty *value* parameter is used) when it's over it sends a message with the *topic* "done" to alert the consumer that there is no more topics.

- *"pull_reply"* → used by the broker's *send_to* method to send any data needed to the middleware, it contains the fields *"topic"* and *"value"* and is received by the middleware's *pull* method wich unpacks and returns the data to the consumer.

There is also a message that doesn't follow the protocol's format, it is the first message sent by any queue on initialization and it only contains the code name for the queues respective serialization mechanism ("json" , "xml" , "pickle"). It is used to inform the broker of the type of serialization that the consumer/ uses. When the broker receives it, it adds an entry composed of the socket and it's mechanism to the *conn_types* dictionary.

4 Tests

A first basic test was to have the broker, three consumers (one of each serialization), and then make two producers, the producers and consumers were hable to communicate, indicating that the broker is accomplishing it's purpose. The scripts used were *consumerV2.py* and *consumerV2.py*, which are identical to the provided ones, only adding an argument to choose the serialization wanted. The results can be seen in Figures 1 and 2.

Secondly, tests to confirm the functionality of multiple subscriptions, the "unsubscribe" message, and the "list" message were included in the file *tests.py.* The final results can be seen in Figures 3 and 4.

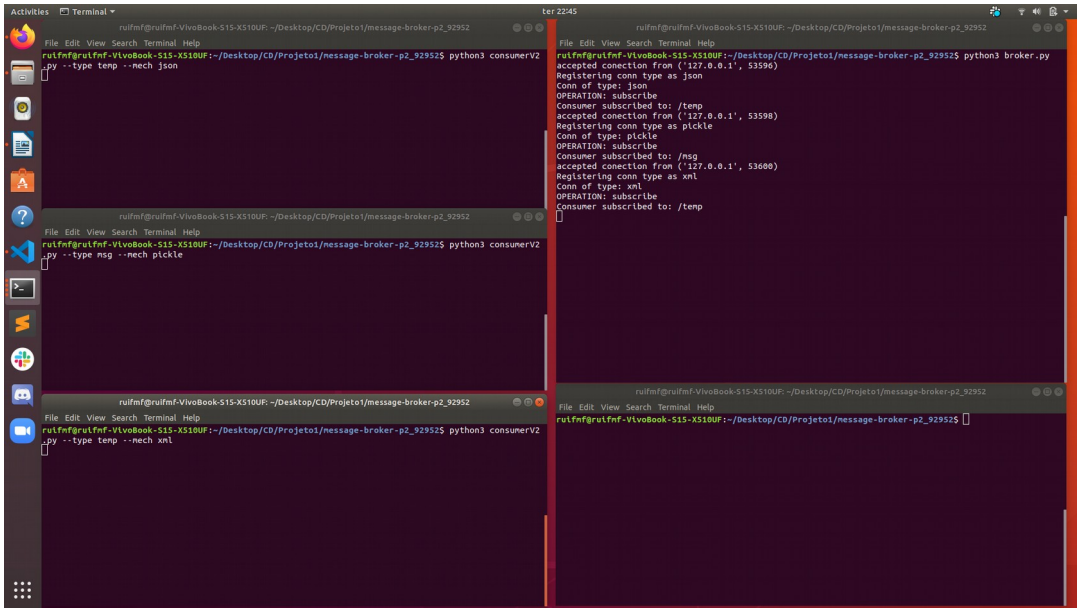

Figure 1

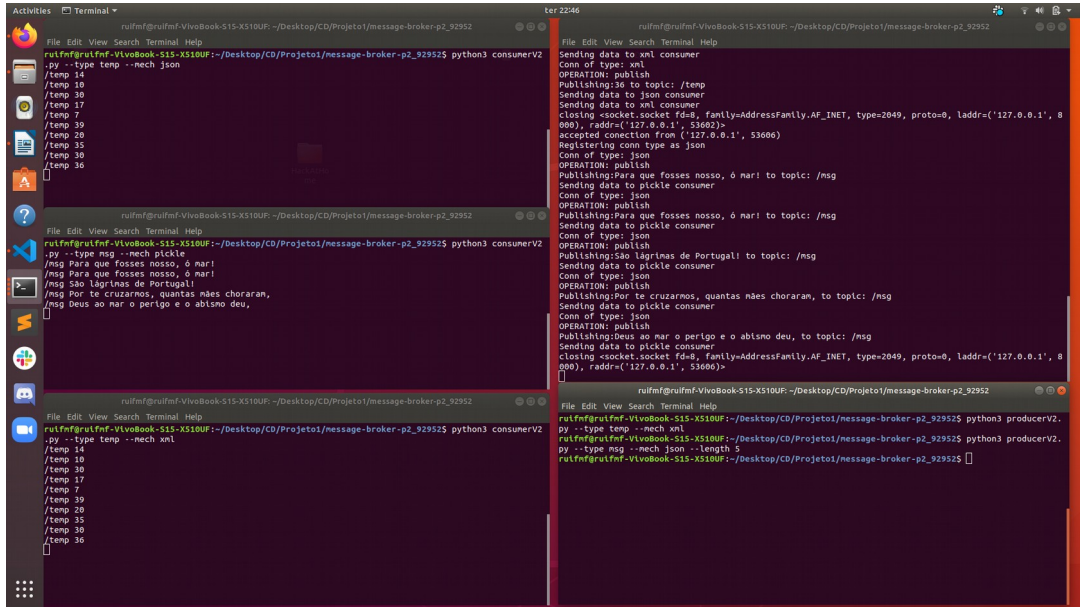

Figure 2

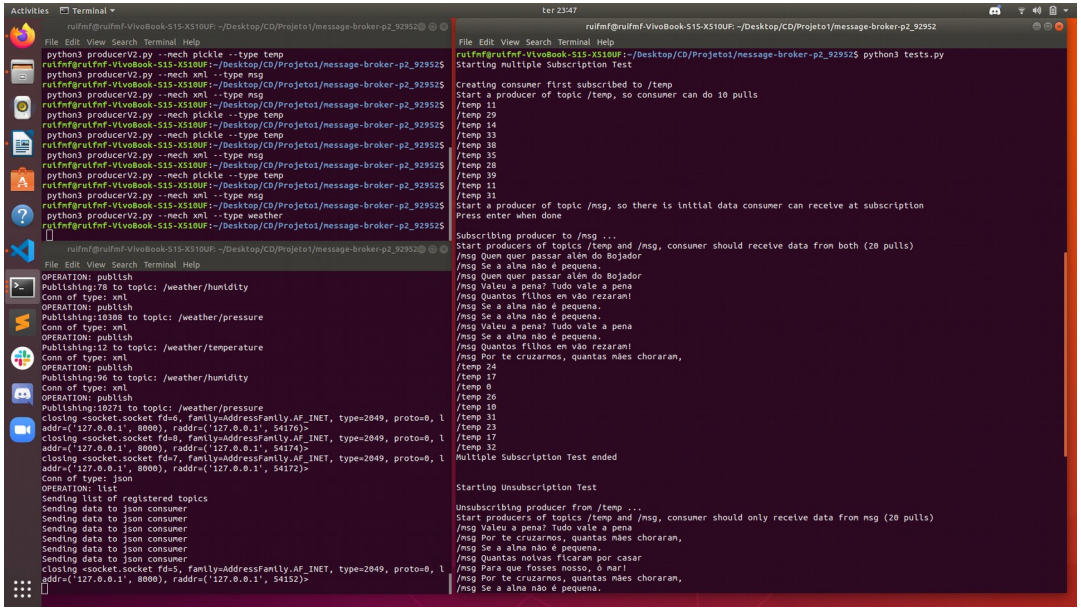

Figure 3

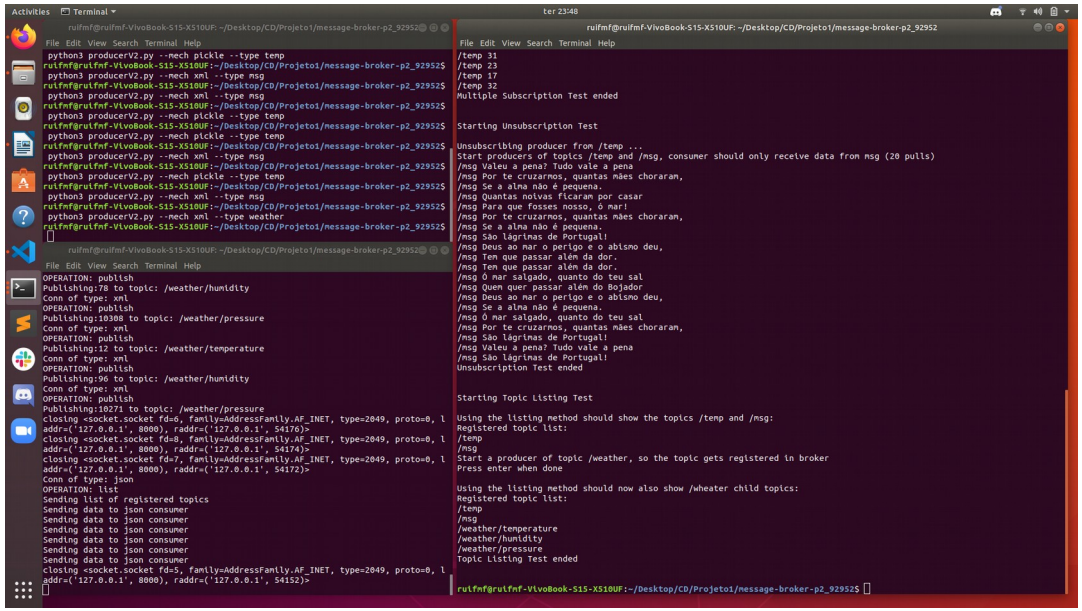

Figure 4