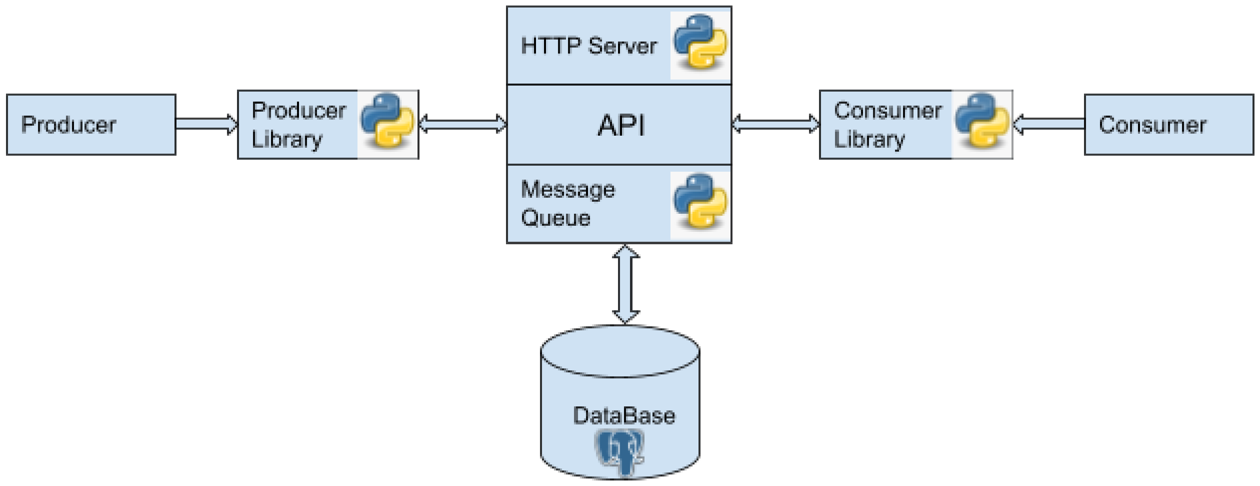
CS60002 Distributed Systems

Assignment 1: Implementing a Distributed Queue

Subham Jain	(22CS60R16)
Aditya Chillara	(22CS60R18)
Venkat Sai Konduru	(22CS60R50)
V.V.S. Sri Harsha	(22CS60R77)
Tudi Jayadeep Reddy	(22CS60R83)

Contents

Architecture:



Design Choices

How frequently put the data in the SQL DB?

Our implementation updates/modifies the DB whenever the in-memory data is updated/modified. We had to choose between frequent DB access and the consistency of DB in case of an unexpected crash. We decided to go with frequent DB access as keeping the system consistent was more crucial.

How to implement the fetch message since different consumers might have to consume from different positions?

The in-memory **topics** data structures(as described below) has a **bias** key which keeps track of how many messages were deleted/completely consumed. Along with that each consumer keeps track of the position from which they need to consume messages. **bias** and **position** are used together to keep the message sequence for each consumer in order.

Do we have to use a queue?

One choice was to use the actual queue data structure to implement the Message queue, but having the problem of different consumers consuming from different positions made us go with dictionaries/ lists like data structure.

How to deal with the publish_message request if the topic has no subscribers?

One choice that we had to make was either keep the message in-memory for an indefinite period so that every new subscriber can start consuming from the start or to delete the message when all the consumers (who subscribed to the queue before the message got published) consume the message. As we went with the later approach we had to make out data structure as such that the producer cannot publish to a topic to which no consumers are subscribed to.

The Message Queue

Introduction

The message queue implementation uses 3 dictionaries for storing relevant data.

- **topics**
- **consumers**
- **producers**

The message queue implementation exposes 10 functions for performing different operations along with that one constructor function **__init__** which acts as a constructor whenever the instance of **Message_Queue** is created. The functions are mentioned below and explained in details further in this document.

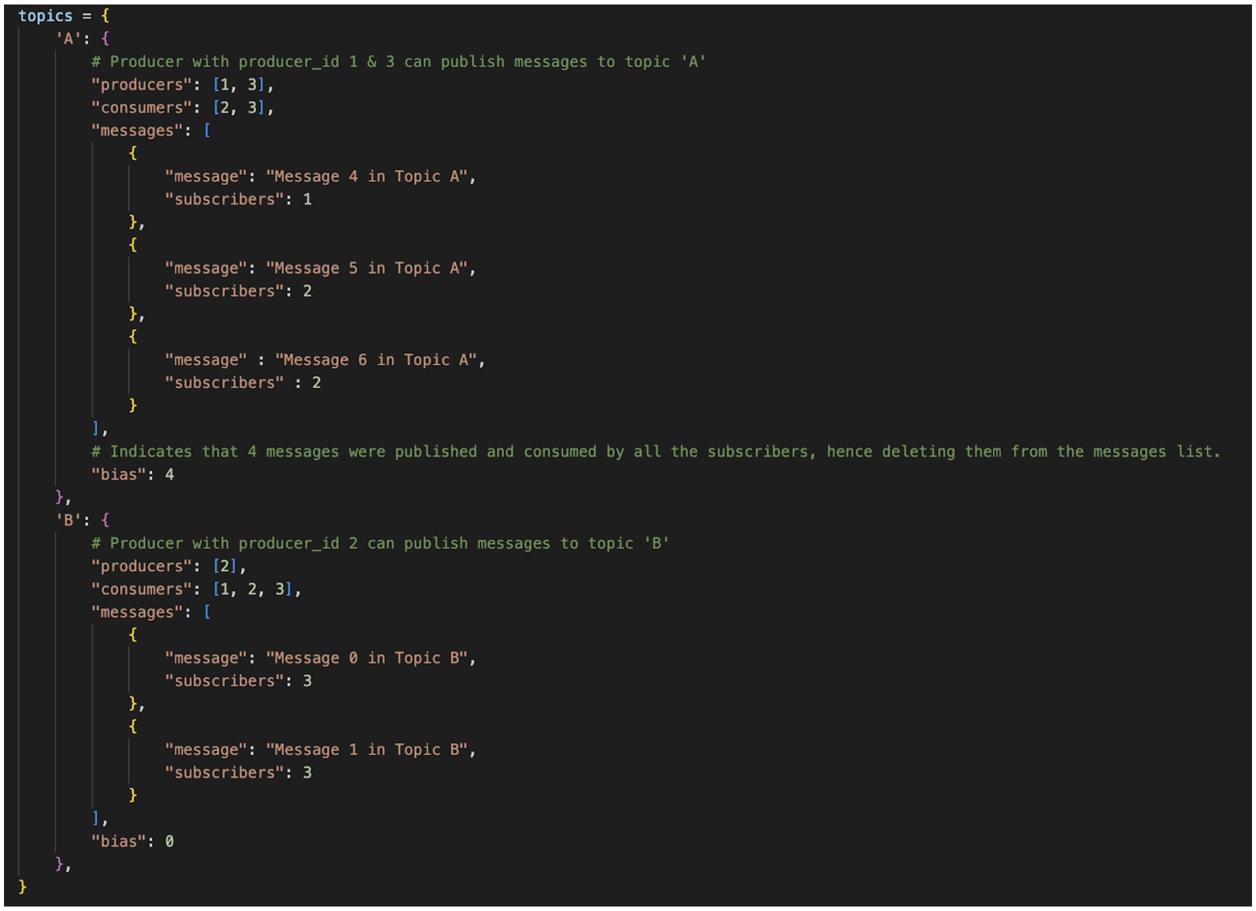
- **init_DB**
- **add_topic**
- **publish_message**
- **register_producer**
- **register_consumer**
- **subscribe_to_topic**
- **add_producer_to_topic**
- **consume_message**
- **list_topics**
- **log_size**

topics

This dictionary holds all the relevant information regarding each topic in the Message Queue. Each topic registered under the MQ can be found as a key in the **topics** dictionary, where the value against the key is again a dictionary holding the below mentioned **<key,value>** pair.

- **producers :-** A list of **producer_id's** that can publish messages to this specific topic.
- **consumers :-** A list of **consumer_id's** that can consume messages from this specific topic.
- **messages :-** A list of dictionary where each dictionary holds information about a message. The dictionary holds two **<key,value>** pairs, first one is **message,** which is the actual message posted and the other being **subscribers,** this key holds the number of consumers that are yet to consume this message, when the message is pushed to the **messages** list the subscribers value is initialized with the number of current subscribers to this topic and as the consumers keep on consuming the message the value decrements and when it reaches 0, the message is deleted from the list to free up the space.
- **bias :-** This is a integer value which is incremented everytime a message from the **messages** key is deleted, used for calculating the correct position of the message that needs to consumed by a consumer.

Snapshot



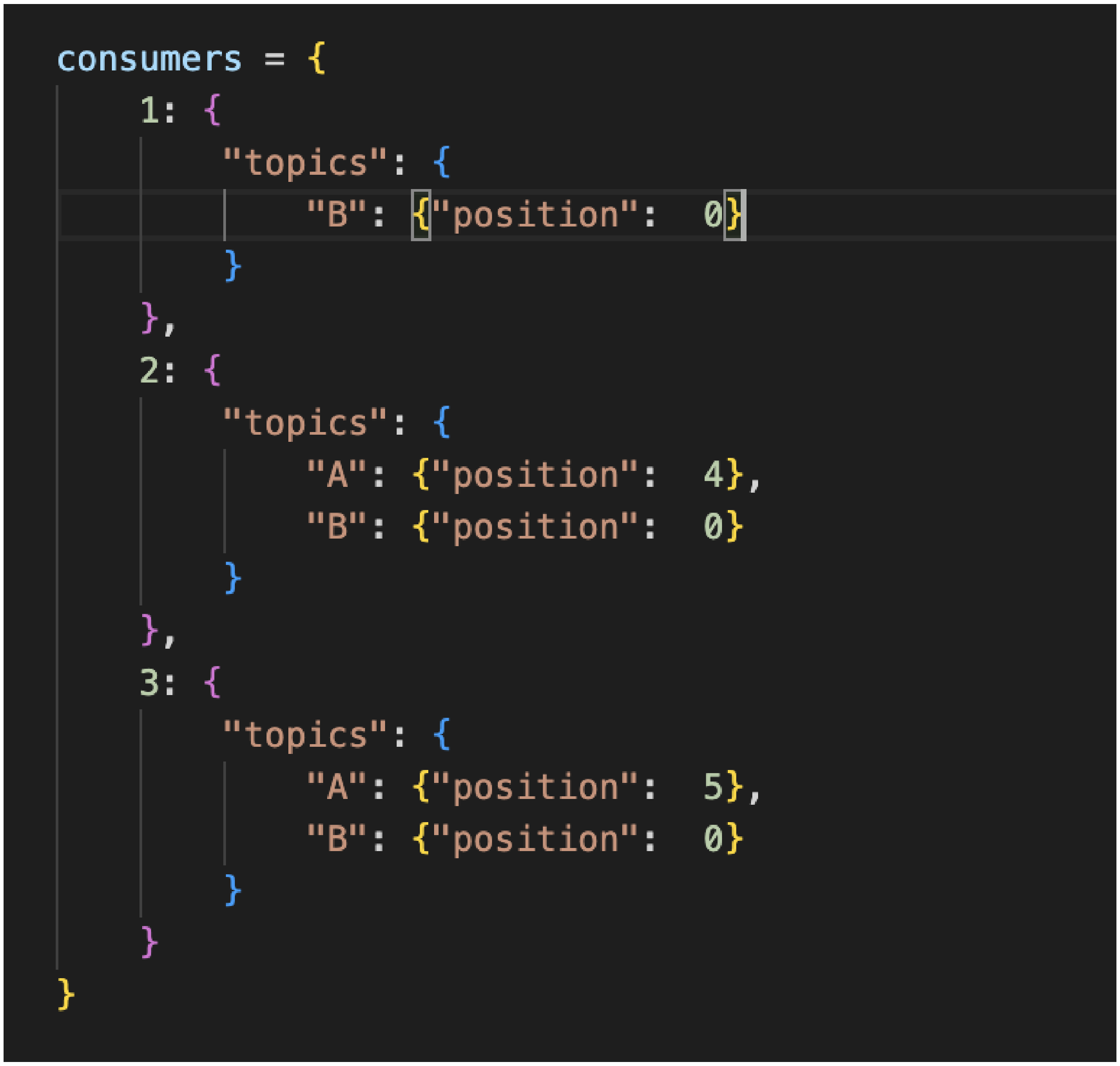
```
topics = {
    'A': {
        # Producer with producer_id 1 & 3 can publish messages to topic 'A'
        "producers": [1, 3],
        "consumers": [2, 3],
        "messages": [
            {
                "message": "Message 4 in Topic A",
                "subscribers": 1
            },
            {
                "message": "Message 5 in Topic A",
                "subscribers": 2
            },
            {
                "message" : "Message 6 in Topic A",
                "subscribers" : 2
            }
        ],
        # Indicates that 4 messages were published and consumed by all the subscribers, hence deleting them from the messages list.
        "bias": 4
    },
    'B': {
        # Producer with producer_id 2 can publish messages to topic 'B'
        "producers": [2],
        "consumers": [1, 2, 3],
        "messages": [
            {
                "message": "Message 0 in Topic B",
                "subscribers": 3
            },
            {
                "message": "Message 1 in Topic B",
                "subscribers": 3
            }
        ],
        "bias": 0
    },
}
```

consumers

This dictionary holds all the relevant information regarding each consumer registered in the MQ. Every **<key,value>** pair in this dictionary is unique where, **key** is the consumer_id and **value** is a dictionary that holds information regarding that consumer. The **value** dictionary holds the below mentioned **<key,value>** pairs.

- **topics :-** A dictionary where each **<key, value>** pair is a topic to which the consumer is subscribed to. The **key** is the topic name and the value is again a dictionary holding just one key which is **position,** which tells the index in the **messages** list that the consumer will be consuming next.

Snapshot



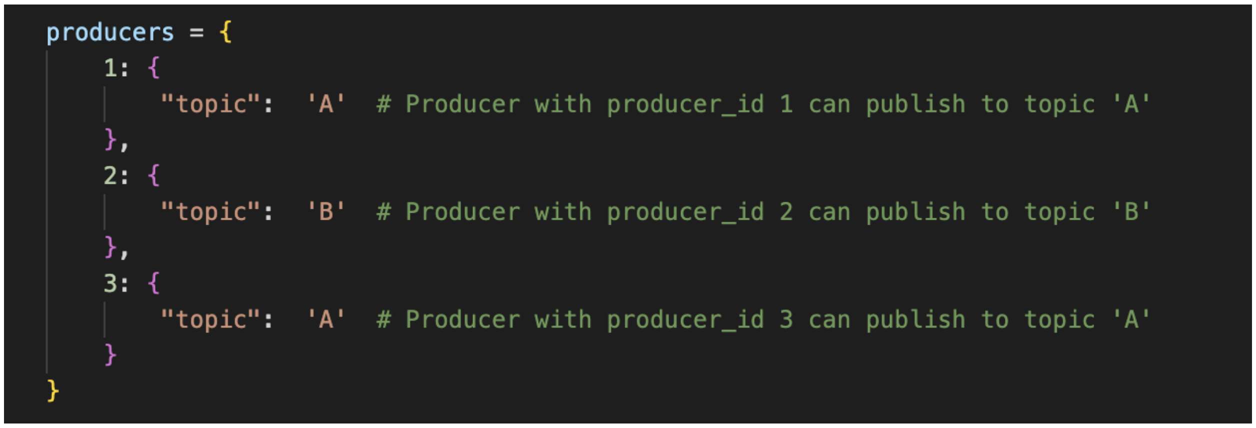
```
consumers = {
    1: {
        "topics": {
            "B": {"position":  0}
        }
    },
    2: {
        "topics": {
            "A": {"position":  4},
            "B": {"position":  0}
        }
    },
    3: {
        "topics": {
            "A": {"position":  5},
            "B": {"position":  0}
        }
    }
}
```

producers

This dictionary holds all the relevant information regarding each producer registered in the MQ. Every **<key,value>** pair in this dictionary is unique where, **key** is the producer_id and **value** is a dictionary that holds information regarding what topic the producer can publish to. The **value** dictionary holds the below mentioned **<key,value>** pairs.

- **topic :-** A string value which stores the topic name.

Snapshot



```
producers = {
    1: {
        "topic":  'A'  # Producer with producer_id 1 can publish to topic 'A'
    },
    2: {
        "topic":  'B'  # Producer with producer_id 2 can publish to topic 'B'
    },
    3: {
        "topic":  'A'  # Producer with producer_id 3 can publish to topic 'A'
    }
}
```

init_DB

This is used to initialize the class members from the Postgres DB.

add_topic

Pushes a topic to the **topics** dictionary. Anyone can add a topic to the system.

Error handling:-

1) If the topic already exists then throw an error.

publish_message

Publishes a message to the provided topic. Only a producer who's registered with the topic can publish messages in it.

Error handling:-

1) If the topic doesn't exist, throw an error.
2) If producer with producer_id doesn't exist then throw an error .
3) If the producer hasn't registered to any topic or the producer has not registered to this topic then throw an error.

register_producer

Creates a producer in the system.

register_consumer

Creates a consumer in the system.

subscribe_to_topic

This adds consumers under a topic.

Error handling:-

1) If the topic doesn't exist then throw an error.
2) If the consumer with consumer_id doesn't exist then throw an error.
3) If the consumer is already subscribed to the topic then throw an error.

add_producer_to_topic

This will add a producer as a publisher for the given topic.

Error handling:-

1) If the topic doesn't exist then throw an error.
2) If the producer with producer_id doesn't exist then throw an error.
3) If the producer is already a publisher in another topic then throw an error.
4) If the producer is already a publisher in same topic then throw an error.

consume_message

This will send the message at the start of the queue to be consumed by the consumer.

Error handling:-

1) If the topic doesn't exist then throw an error.
2) If the consumer with consumer_id doesn't exist then throw an error.
3) If the consumer is not a subscriber of that topic then throw an error.

list_topics

This will list all the topics which are there in the MQ.

log_size

This will return the number of the messages that a consumer has remaining that needs to be consumed at that point of time.

Error handling:-

1) If the topic doesn't exist then throw an error.

2) If the consumer with consumer_id doesn't exist then throw an error.
3) If the consumer is not a subscriber of that topic then throw an error.

HTTP Server API

The HTTP server API is written in python. A simple implementation of request handler is made using python built in module BaseHTTPRequestHandler.

Server address: http://127.0.0.1:8002
Following are the API Endpoints made available:

- /topics
    - GET request
    - Returns a list of topics available
- /consumer/consume
    - GET request
    - Params = consumer_id, topic
    - Returns a message after dequeuing from the Message Queue
- /size
    - GET request
    - Params = consumer_id, topic
    - Returns size of the queue of given topic in Message Queue
- /topics
    - POST request
    - Params = topic_name
    - Creates a new topic of given name
- /consumer/register
    - POST request
    - Params = topic
    - Registers consumer and returns corresponding consumer_id
- /producer/register
    - POST request
    - Params = topic
    - Registers producer and returns corresponding consumer_id
    - Creates the topic if it doesn’t already exist
- /producer/produce

  - POST request
  - Params = topic, producer_id, message
  - Adds the given message to the Message Queue

Persistent Storage Layer

The message queue is lost if the server crashes or restarts as the messages are stored in the memory. Hence, we have added a persistent storage layer to recover from such issues. There are three tasks:

1. Determine schema
2. Store and retrieve log messages
3. Tests to ensure that the functionality is as expected and the message queue is able to recover from failure.

Why Postgres?

PostgreSQL is an enterprise-class open source database management system. It supports both SQL and JSON for relational and non-relational queries for extensibility and SQL compliance. PostgreSQL supports advanced data types and performance optimization features, which are only available in expensive commercial databases, like Oracle and SQL Server. The features supported by postgres are user-defined datatypes, table inheritance, foreign key referential integrity and so on.

Determining Schema

The schema used are:

1. topic : The table topic contains topic_name as primary key and bias.
   Bias is a value that is incremented if a message in the message queue for a particular topic is consumed by all the subscribers.
2. producer : The table producer contains pid, topic_name
   producer id - pid
   topic_name is the topic the producer registered to and is a foreign key referred to in the topic table.

3. consumer : The table consumer contains cid, topic_name and position.
   consumer id - cid
   topic_name of the topic consumer registered to and is a foreign key referred to in the topic table.
   position is the message position of the consumer in the message queue.
4. message: The table message contains message, topic_name and subscribers
   message is the message of the topic.
   topic_name is the topic name of the message and is a foreign key referred to in the topic table.

subscribers are the consumers that are subscribed to the topic name which has the message.

Producer Library

Producer Library written in python language consists of Producer class with 4 different functions:

RegisterProducer

The objective of this function is to register a producer for a particular topic name .This function takes the Topic name as argument and registers the Producer to that topic. On success it returns a Producer ID indicating that Producer is successfully registered to that topic.
On failure it raises myProducerError with an error message representing the error**.**

ListTopics

The Objective of this function to display the list of topics present in the distributed queue.The function takes no arguments. On success it returns the list of all the topics available that the producers have created.
On failure it raises myProducerError with an error message representing the error.

Enqueue

The Objective of this function is to enqueue a log message to a Particular Topic with Producer id.The function takes 3 arguments namely Producer ID,Topic name and Log message. On success it returns 0 indicating that the log message is enqueued into the queue.
On failure it raises myProducerError with an error message representing the error.

CreateTopic

This function takes Topic name as argument to create a topic name for the producer.On success it returns 0 indicating that the Topic has been created by the producer.
On failure it raises myProducerError with an error message representing the error.

Consumer Library

The Consumer is written in Python. It has the following functions:

RegisterConsumer

The function takes the topic name as argument and registers the consumer to the topic.
Inorder to subscribe/Register to multiple topics call this function multiple times with different topics.
On failure it raises a myConsumerError with the error message.

ListTopics
The function takes no arguments. On success it returns the list of all the topics available that the producers have created.
On failure it raises a myConsumerError with the error message.
Dequeue
The function takes one argument namely, topic name. On success the function returns the log message that is dequeued from the requested topic log queue.
On failure it raises a myConsumerError with the error message.
Size
This function takes one argument namely, topic name. On success the function Returns the size of the log queue for the requested topic.
On failure it raises a myConsumerError with the error message.

System Tests

systemtest1
Implemented 5 Producers and 3 consumers with 3 topics using the producer and consumer libraries where producers P1, P2, P3, consumers C1, C2, C3 are registered to topic T1, producers P1, P4, P5, consumer C1 are registered to topic T1 and producers P1, P2, consumers C1, C2, C3 are registered to topic T3.
The files producer_<ID>.txt contains the logs generated by the producer with corresponding ID.
The producers parallelly push the logs from log_file to the server with delay of 1 sec between two enqueue requests from the same producer to the server. Simultaneously, the consumers which will consume the logs in order.
Crashed the server while the producers are logging and verified that the data is restored correctly from the database.

systemtest2
Registered a producer and a consumer to the topic DS. The producer logs a message using the Enqueue function and the consumer retrieves the message by the Dequeue function. Verified that the log message retrieved by the consumer is the same as the logged message by the producer.