Design & Implementation of Distributed Logging Queue

Models

1. Consumer Class:
   It represents a single consumer that has a unique consumer_id, a topic_name it is subscribed to, and a cur_topic_queue_offset.
   a. The offset was maintained for each view of a topic by a consumer to maintain different states of the same topic for different consumers interacting with it.
   b. The class contains methods to get the next message from the topic queue, get_next_message(topic_queue), and get the count of messages that need to be fetched from the topic queue, get_count_messages_to_fetch(topic_table).

2. Producer Class:
   a. __init__: Initializes a producer object with the given producer_id and producer_topic.
   b. This class will be used to call the Producers in the server side

3. Message Class:
   It is used for the log messages

In Memory Structures

4. ConsumerTable Class:
   It is used to maintain a dictionary of consumer_entry and an auto-incrementing consumer_id_auto_inc field to keep track of the number of consumers registered. The class contains methods to get a consumer by its consumer_id, create a new consumer given consumer_id and topic_name, and register a new consumer to a given topic_name.

5. ProducerTable Class:
   a. __init__: Initializes an empty dictionary producer_entry to store producer objects, and an auto-incrementing integer producer_id_auto_inc to keep track of unique producer IDs.
   b. get_producer: Returns the producer object associated with the given producer_id.
   c. create_producer: Creates a new producer object with the given producer_id and topic_name, and adds it to the producer_entry dictionary.
   d. register_new_producer_to_topic: Registers a new producer to a topic by incrementing the producer_id_auto_inc counter, creating a new producer object with the updated ID and topic name, and returning the new ID.

6. TopicTable Class: The TopicTable class is a table that contains multiple TopicQueue objects, each identified by a unique topic name. The class has the following methods:
    a. __init__ This method is called when a new TopicTable object is created. It initializes an empty dictionary topic_queues to store the TopicQueue objects and a variable last_id with a default value of 0.
    b. get_topic_queue This method returns the TopicQueue object associated with a given topic name.
    c. get_topic_list This method returns a list of all the topic names in the TopicTable.
    d. create_topic_queue This method creates a new TopicQueue object with the given topic name and adds it to the topic_queues dictionary.

7. TopicQueue Class:
   The TopicQueue class is a simple queue data structure that stores messages in a first-in-first-out (FIFO) order. It has the following methods:
        i. __init__ This method is called when a new TopicQueue object is created. It initializes the topic_name with the given value and an empty list topic_queue to store the messages.
        ii. .enqueue This method adds a message to the end of the topic_queue. get_at_offset This method returns the message at a given offset in the topic_queue. If the offset is greater than or equal to the length of the queue, None is returned.
    b. size This method returns the number of messages in the topic_queue. can_write This method returns True as a placeholder for checking if writing to the TopicQueue is possible.

8. MessageTable : This class contains all the message objects table entries and their corresponding ids.A class that represents a message table to store and retrieve messages
    a. __init__ : Initialize an empty message table.
    b. get_new_id_auto_inc : Get a new message id by auto incrementing the last_id counter.
    c. get_message : Get a message from the message table with a given message id. Args: message_id (int): The id of the message to retrieve. Returns: Message: The message if it exists in the message table, None otherwise.
    d. add_message: Add a message to the message table and assign a new id to it. Args: message (Message): The message to add to the message table. Returns: int: The id assigned to the message.

Database Structures

- The database structures are similar to their in-memory counterparts.
- In all the database tables, ID is given the type SERIAL for auto-incrementation.

9. ConsumerDBMS Class:
   create_table() creates the table with columns id, topic and ofset. Notice the use of ofset as column name instead of offset, as offset is a keyword in postgresql.
   register_to_topic() registers the consumer to the topic name taken as argument. It has been checked earlier that the topic name getting passed here already exists.
   get_consumer() fetches the Consumer object for a given consumer_id.
   increase_offset() increases the offset for the current topic queue so as to get the next message in it.

10. ProducerDBMS Class:
    create_table() creates the table with columns id and topic.
    register_new_producer_to_topic() registers the producer to the topic name taken as argument. It has been checked earlier that the topic name already exists or is created before getting passed here.
    get_producer() fetches the Producer object for a given producer_id.

11. TopicDBMS Class:
    create_table() creates the table with columns id, name and messages. It is ensured that each topic name should be unique. The message queue is maintained as an integer array of message ids so that any new consumer will have access to all the messages in the table.
    create_topic_queue() adds a new row to the table with given topic name, and the messages column for it is initially null
    get_topic_list() fetches the names of all topics in the table
    get_topic_queue() fetches the TopicQueueDBMS object for a given topic name

12. TopicQueueDBMS Class:
    This class is used to perform certain operations to the message queue in a topic
    enqueue() appends a log message to the given topic. If the message array is null, it creates the array with given log message as sole element.
    get_at_offset() fetches the message id of the message at given offset
    size() returns the length of the entire message queue

13. MessageDBMS Class:
    create_table() creates the table with columns id and message.
    add_message() inserts a new row with given message to the table.
    get_message() fetches the Message object for a given message id

Message Queue System

14. MessageQueueSystem Class:
    It takes an argument ‘persistent’ which uses the database when true and uses the in-memory structures otherwise. If persistent, psycopg2 library is used to connect to the database. The tables are not created here but once at the start (if does not exist)

before any request is made.

create_topic() adds a new topic queue with given name.

list_topics() returns the list of topics present in the table.

register_producer() registers a producer to given topic with unique id. If the topic does not exist, it is created before the registration.

register_consumer() subscribes a consumer to given topic with unique id. If the topic does not exist, it throws an error.

enqueue() publishes a new log message to the given topic's queue. It throws an error if the topic does not exist or the producer is not registered under it.

dequeue() fetches the first unread / unretreived message and increments the offset to next message. It throws an error if the topic does not exist, the producer is not registered under it or there are no more messages left to retrieve for the consumer. Notice the use of offset+1 for persistent instead offset. This is done because of the 1 based indexing of messages in persistent storage.

size() returns the number of log messages in the topic that are not yet retrieved by the consumer

Design and Implementation of Client Side

The design implementation of the client side included creating the libraries for the producer and the consumer side and some of the extra functions of listing the topics and creating the topics that was included in the extra class of ServerFunctions.
Following libraries were created inside the myqueue folder for the partC execution :

1. consumer.py :
    a. This file contains the implementation of client side class of the consumer aspect which includes all it's functionalities.
    b. First here we initialize the consumer class with all the required arguments, then in the initialization we register the consumer for the list of topics with which it is initialized with. This init function calls the Server side api call for the registering the consumer.
    c. **get_next(topic)** : this function encapsulates the main functionality of the consumer side of receiving the messages. This function first checks if the topic entered have been subscribed by the consumer or not, if not then it sends an error message of subscribing the topic. If the topic is indeed registered then it sends the call to Server side api call for **receiving** of the message and returns the required arguments according to response received on the request call.

2. producer.py :
    a. This file contains the implementation of client side class of the producer aspect which includes all it's functionalities.
    b. First here we initialize the producer class with all the required arguments, then in the initialization we register the producer for the list of topics with which it is initialized with. (Note : producer is not given with the functionality to create a new

topic). This init function calls the Server side api call for the registering the producer.

c. **send(topic, message)** : this function encapsulates the main functionality of the producer side of sending the message. This function first checks if the topic entered have been registered by the producer or not, if not then it sends an error message of registering for the topic. If the topic is indeed registered then it sends the call to Server side api call for **publish** of the message and returns the required arguments according to response received on the request call.

3. topic_lib.py :
    a. This file includes the implementation of the client side of the admin functions of the distributed queue
    b. First here we initialize the serverfunctions class with all the required arguments.
    c. CreateTopic(topic) : : this function encapsulates the main functionality of the admin side of creating the topics. This function first checks if the topic entered have already been created, then it sends an error message of creating the topic.

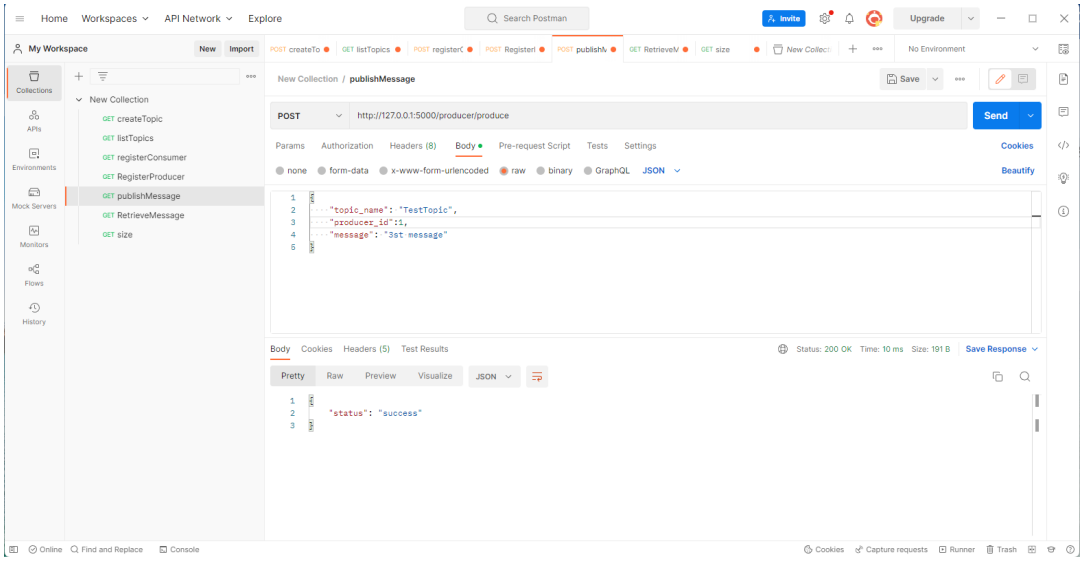
Design and Implementation of Server Side

- Server was implemented using Python based Flask framework
- All the endpoints have been tested using Postman and its collection has also been attached in the JSON format
- All the responses have been combined with corresponding response codes (2xx & 4xx) for easy categorization

Testing

1. Unit Testing for in-memory structures:
   All the functions used in the queue for in-memory structures were tested in unit_test1.py
2. Unit Testing for database:
   All the functions used in the queue for database structures were tested in unit_test1.py. Older tables are dropped if they exist and new tables are created for testing here.
3. Raw API Testing:
   Postman was used to test all the apis in the localhost server. APIs tested were :
   createTopic
   listTopics
   registerConsumer
   RegisterProducer
   publishMessage
   RetrieveMessage

size



Challenges

- Ensuring that the distributed logging systems works consistently both in presence and absence of the persistent storage layer.
- Chaining Exception: In try and exception same error were being raised hence it was becoming difficult to debug the errors.

Hyperparameters Assumed

1. DEFAULT_OFFSET: This hyperparameters begins from 0
2. IS_PERSISTENCE : True