Assignment 1 Report

Group Members:
Abhilash Datta 19CS30001
Sunanda Mandal 19CS10060
Rohit Raj 19CS10049
Haasita Pinnepu 19CS30021
Matta Varun 19CS30028
Suguna Bhaskar Tirupathi 19CS10063

Implementation

Distributed Queue

The distributed server is a blueprint for a web server that implements a simple log queue system. The log queue system is used to store and distribute log messages to consumers who have registered to receive messages for a specific topic. The code uses the Flask framework to handle HTTP requests and responses, and SQLAlchemy as the ORM to interact with the database. It also uses an in-memory data structure, log_queue, to store log messages temporarily before they are written to the database. This in-memory data structure is implemented as a custom class InMemoryLogQueue. It defines several endpoints for different operations related to the log queue system. These operations include creating a new topic, registering a consumer or producer, enqueuing log messages, and retrieving the list of topics.

The setup_db_and_backup function is executed before the first request to the server and it creates the database tables if they don't exist and also performs a complete backup of the log queue system.

The /status endpoint returns the status of the server, which is always "success".

The /topics endpoint can be used to create a new topic by sending a POST request with the topic name in the request body. It returns the status of the operation and a message indicating whether the topic was created successfully. The same endpoint can also be used to retrieve a list of all topics by sending a GET request.

The /consumer/register endpoint can be used to register a new consumer by sending a POST request with the topic name in the request body. It returns the status of the operation and the consumer ID assigned to the consumer.

The /producer/register endpoint can be used to register a new producer by sending a POST request with the topic name in the request body. It returns the status of the operation and the producer ID assigned to the producer.

The /producer/produce endpoint can be used by a producer to enqueue a log message by sending a POST request with the topic name, producer ID, and log message in the request body. It returns the status of the operation.

Client Library (LogQueueSDK)

LogQueueSDK is a client library developed to provide an easy to use interface accessing the functionalities and engaging with the distributed queue server. Producers and Consumers can use this library to interact with the server while ignoring the underlying requests mechanism. All necessary utility functions were implemented and classified into Producer and Consumer classes.

The library consists of a base class "**Client**" that implements the common functionalities between Producers and Consumers such as "list_topics". **Producer** class and **Consumer** class inherit from Client and implement their own required methods. A producer would instantiate an object of the Producer class using two parameters - address of the broker and list of topics to register for (none by default). Similarly, a consumer would instantiate an object of the Consumer class using the same parameters. The clients would then continue using these objects to access the functionalities of the server. This provides for an easy to use and intuitive interface with the server.
Producer and Consumer also have a member variable topic_id_map which is a dictionary of topics registered to and correspondingly allocated ID. Following is a brief summary of classes and their methods.

Client

- *create_topic : returns bool*
- *list_topics : returns list*

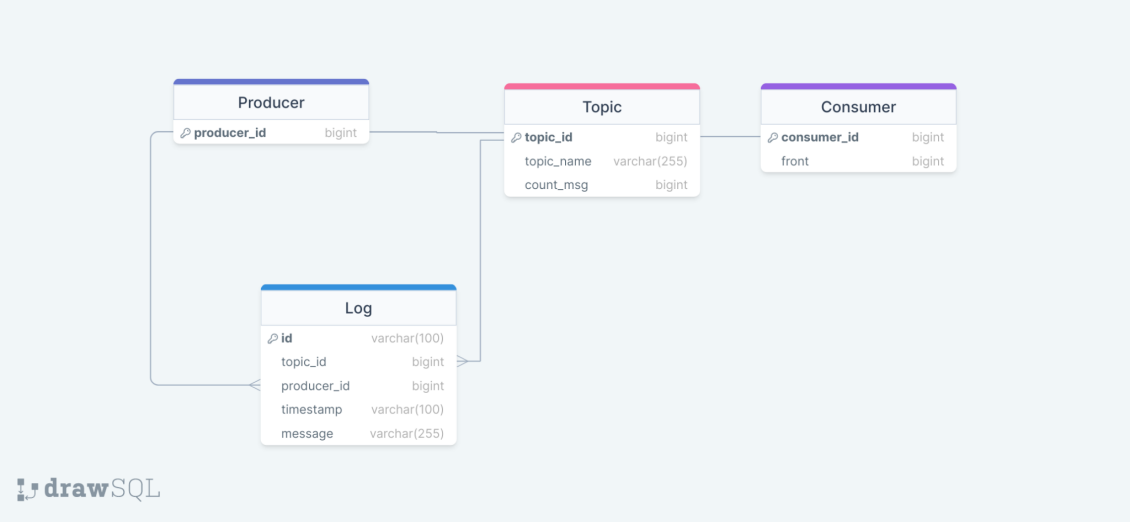
Producer

- *register : returns int (producer_id)*
- *send : returns bool*
- *can_send : returns bool*
- *stop : stops*

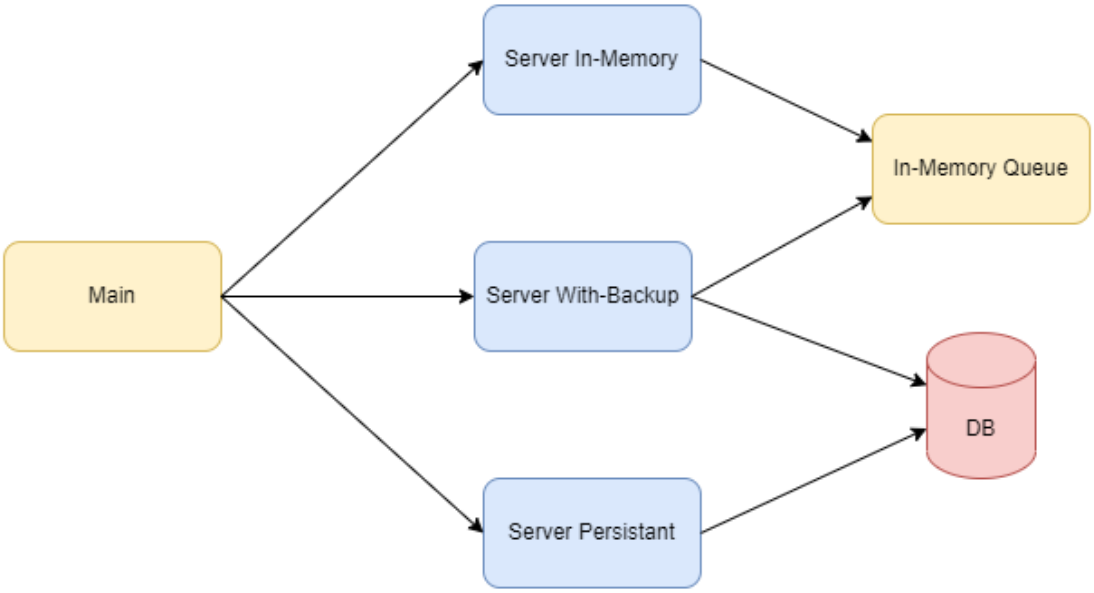
Consumer

- *register : returns int (consumer_id)*
- *dequeue : returns string*
- *size : returns int*
- *get_next : iterates over all topics and dequeues the messages*
- *stop : stops*

Database Schema



System Design



Challenges

It was the first time our group was working together on Github, One of the challenges we faced was version control and collaboration. Different team members were working on the same/different files simultaneously, leading to conflicts in code that needed to be resolved. And then the debugging process took quite some time, due to small errors like non-matching method names across files, etc. Testing the code then again got delayed due to version changes being done by the members.
Coming to the design challenges that we faced, we made the database server async, which was something new that we explored. And the exhaustive testing done, to check each and every failure case, handling exceptions and synchronization between threads were other difficulties that we encountered and resolved.

Hyperparameters

Some of the hyperparameters that we considered during implementation would be the number of servers running a script, the maximum size of a log, the maximum number of logs that can be stored in the database, and the number of columns in the database. Similarly, the number of logs that can be inserted per second is also to be considered.

Testing

We performed exhaustive unit testing to cover all possible test cases for the server_in_memory and server_persistent, some of the test cases would be:

1. For create_topic:
    a. success for new topic_name
    b. Failure for existing topic_name
2. For register_consumer:
    a. success on existing topic_name
    b. Failure if topic_name does not exist
3. For register_poducer:
    a. Success if topic exists
    b. Success if topic does not exist
4. For enqueue:
    a. Success
    b. Failure when topic_name doesn’t exist
    c. Failure when producer id doesn’t exist
    d. Failure when topic_name and producer_id doesn’t match
5. For dequeue:
    a. Success
    b. Failure when topic_name doesn’t exist
    c. Failure when consumer id doesn’t exist
    d. Failure when topic_name and consumer_id doesn’t match