Distributed Systems Assignment-1

(Implementing a Distributed Queue)

In this assignment, we have been tasked with implementing a distributed logging queue to store log messages generated by the company's various applications and services.

Software/Tools used

We have used the following tools to build the application:

- Backend: Python Flask, Postgresql for database
- Frontend: Python

Implementation

The application primarily consists of a server program and, a number of client programs that interact with the server program to subscribe, log and retrieve messages. The server program is built using the popular Flask framework and Postgresql as a database.

The Queue Data Structure

The server program maintains a global queue data structure to store and retrieve messages for both producers and consumers. Here we describe the salient features of the queue and important implementation details:

- ***Topic Level Parallelism:*** Flask being a multithreaded application it is crucial to maintain consistency of the queue across all request-response cycles without much compromise in performance. For this, we maintain separate queues for each topic i.e. all operations are topic-wise multiplexed. Hence we only need exclusive locks for each topic queue which lets the server handle requests for different topics parallelly. The queue is implemented as a python dictionary from the topic name to an array of messages. This ensures insertion, deletion, and retrieval in O(1) time (Worst case O(N) time).

- ***Maintaining different states of queue per consumer***: Clearly different consumers may be reading different messages for the same topic. To ensure this we have kept a single source of truth (the queue) and maintained different offsets for each consumer ID.

- ***Locks for consistency:*** Apart from the lock for each topic queue we have used a lock for each consumer offset, a global lock for the entire queue during the creation of a new queue for a new topic, and locks for the lists of subscribed users on a certain topic. We have tried to introduce locks for each resource that can have concurrent updates to ensure maximum parallelism.

Adding persistence with a Database

To add persistence to the messaging service we have added a database based on Postgresql. Before returning the response we update the global queue, and the database (if necessary) and return the response.

Below is the implementation of model.py

```
class QueueDB(db.Model):
   '''
   Id : PK int
   Nxt_id: int
   Value: string
   Topic: string as FK
   '''
   id = db.Column(db.Integer,primary_key=True)
   nxt_id = db.Column(db.Integer)
   value = db.Column(db.String,nullable=False)

class Topics(db.Model):
   id = db.Column(db.Integer,primary_key=True)
   value = db.Column(db.String,primary_key = False,nullable=False)
   start_ind = db.Column(db.Integer,nullable=True)
   end_ind = db.Column(db.Integer,nullable=True)
   producers = db.relationship('Producer',backref='topic',lazy=True)
   consumers = db.relationship('Consumer',backref='topic',lazy=True)


class Producer(db.Model):
   id = db.Column(db.Integer,primary_key=True)
   topic_id = db.Column(db.Integer,db.ForeignKey('topics.id'))

class Consumer(db.Model):
   id = db.Column(db.Integer,primary_key=True)
   offset = db.Column(db.Integer,nullable=False)
   topic_id = db.Column(db.Integer,db.ForeignKey('topics.id'))
```

QueueDB: This is implemented like a linked list where at each record we store the id of the next record.
Topics: Here the necessary data required for maintaining the topic is stored.start_ind denotes the first id in the QueueDB database, while end_ind denotes the id of the last element of the queue. We maintain one to many relationship with Producer and Consumer
Producer: We just store the id and the link to the Topic registered.
Consumer: We just store the id and the link to the Topic registered.

Library

Class MyQueue:

Methods:

```
createTopic(self,topicName:str) :
      To create topic
get_all_topics(self):
      To get all topics
createProducer(self,topicNames:list):
      To create Producer,returns Producer Object
createConsumer(self,topicNames:list):
      To create Consumer, returns Consumer Object
```

Class Producer:

Methods:

```
registerTopic(self, topicName: str):
      To register Topic
enqueue(self,msg:str,topicName:str):
      To add to queue
```

Class Consumer:

Methods:

```
registerTopic(self, topicName: str):
      To register Topic
dequeue(self,topicName:str):
      To dequeue from the queue,returns message dequeued
getSize(self,topicName):
      To get size of current queue, returns size
```

Testing

We tested the application using the following tests:

- We tested each endpoint using the curl
- We tested all the success and failure responses of each endpoint
- We ensured the correctness of the application by creating a number of producers and consumers where each producer produce messages and each consumer consumes messages independently and concurrently and finally checked if each consumer has received the messages in the correct order.

Challenges faced

The real difficulty in the assignment lies in designing the queue data structure, identifying all sources of consistency issues due to parallel access, and using the locks judiciously to ensure maximum performance. Also ensuring correctness by using test cases is rather difficult as multiple producers and consumers access simultaneously.