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.

Distributed Systems Assignment-2

(Distributed Queue with Partitions and Broker Manager)

Building on top of assignment-1 we now have to incorporate a Broker Manager who will distribute the load among the brokers (which we created in assignment-1) i.e. the brokers will handle all the tasks from creating topics to registration, enqueue, and dequeue of messages but, who will handle which request will be decided by the Broker Manager. The Broker Manager only maintains the necessary metadata for those decisions.

Software/Tools Used

We have used the following tools to build the application:

- Backend: Python Flask, Postgresql for database
- Frontend: Python
- Other Tools: Docker

Implementation

The application consists of a Broker Manager and several Brokers. We implemented the Brokers in assignment-1 and we have made **NO** design changes in the Broker implementation except that the API requests to enqueue and dequeue operation now include a partition number. The Broker Manager actually consists of a single Write Manager and several Read Managers. Even though both are the same application, the behavioral distinction is created using an *IS_WRITE* environment variable.

Write Broker Manager

Here we describe the salient features of the Write Manager and several design choices we made during its development.

Metadata

As we mentioned earlier the Write Manager only maintains the necessary metadata for load distribution. We maintain the following metadata:

- Topic Metadata: For each topic created the Manager maintains its ID, number of partitions, and a round-robin index (explained later). It also maintains which topic, and partition pair is allotted to which broker.

- Client Metadata: For both producers and consumers it only stores metadata of clients who are registered on an entire topic. For these clients, it maintains a global Client ID which maps to several local Client IDs for the different partitions of the topic in different brokers. The client is only aware of the global Client ID.

- Broker Metadata: For each active broker it maintains several pieces of information like broker ID, broker URL, last heartbeat timestamp, docker name, etc.

Algorithmic aspects

Here we describe the several design choices we made while deciding how a certain function will work in the Write Manager:

- *Deciding the number of partitions and their broker allotment of a topic:* The Write Manager after getting a create topic request will randomly decide the number of partitions between 1 and the number of active brokers and then for each partition, it will select a broker at random to allot it the corresponding partition by sending it a create topic request. In case we detect a broker crash we will recover that broker in a separate thread and send a “Service currently unavailable message” immediately.

- *Implementing partitions in a Broker:* The broker doesn’t know anything about partitions. While the Manager sends create topic request to the broker it sends the topic name as topicName#PartitionID and itself maintains which topicName#PartitionID is allotted to which broker. While enqueuing and dequeuing the broker will construct the topic name from the topic name and partition ID sent in the request header.

- *Implementing Client registrations:* If the client is registered on a particular topic partition the manager just sends the request to the corresponding broker. On the other hand, if he’s registered on an entire topic, the manager decides which partition to choose by using a round-robin load distribution scheme and then sends a request to the selected broker.

- *Load Distribution:* We are giving sort of ownership of the partition to consumers, and this ownership changes in a round robin fashion.This ensures that on no broker will there be too many requests compared to other brokers.This is implemented by assigning an index(say i) for each consumer and a rrindex(say j) for each topic.Then the consumer will access the (i+j)%len th partition and after that j is incremented.

- *WAL implementation:* While we are implementing the database using postgres.Postgres has an inbuilt WAL mechanism where we do a series of updates and then after that call db.session.commit when the entire commit takes place.p

- Health-check Mechanism: We have implemented a function that will periodically ping each broker and record the timestamp in the corresponding broker metadata. In case it doesn't get any response it will recover that broker in a separate thread and carry on pinging the other brokers.

- Broker crash recovery: If a broker is detected inactive by the health check mechanism or, a 404 response the broker is recovered in a separate thread. // Explain the recovery mechanism like how it gets all previous data etc

Database for persistence

We are using postgresql for persistent storage.Each broker is assigned an independent postgres server.
Below is the metadata database schema

```
###################### FOR TOPIC METADATA ########################



class TopicDB(db.Model):
   topic_id = db.Column(db.Integer,primary_key=True,nullable=False)
   topicName = db.Column(db.String,primary_key=False,nullable=False)
   numPartitions = db.Column(db.Integer,nullable=False)
   rrindex = db.Column(db.Integer,nullable=False)
   #topicMetaData_id =
db.Column(db.Integer,db.ForeignKey('TopicMetadataDB.id'))
class TopicBroker(db.Model):
   id = db.Column(db.Integer,primary_key=True,nullable=False)
   topic = db.Column(db.String,primary_key=False,nullable=False)
   partition = db.Column(db.Integer,primary_key=False,nullable=False)
   brokerID = db.Column(db.Integer,primary_key = False,nullable=False)
```

```
class globalProducerDB(db.Model):
   glob_id = db.Column(db.String,primary_key=True,nullable=False)
   rrindex = db.Column(db.Integer,nullable=False)
   topic = db.Column(db.String,nullable=False)
   localProducer =
db.relationship('localProducerDB',backref='globalProducer',lazy=True)
   brokerCnt = db.Column(db.Integer,nullable=False)

class localProducerDB(db.Model):
   id = db.Column(db.Integer,primary_key=True,nullable=False)
   local_id = db.Column(db.Integer,primary_key=False,nullable=False)
```

```
   broker_id =
db.Column(db.Integer,db.ForeignKey('broker_meta_data_db.broker_id'),nullab
le=False)
   glob_id =
db.Column(db.String,db.ForeignKey('global_producer_db.glob_id'),nullable=F
alse)
   partition = db.Column(db.Integer,nullable=False)
######################### FOR CONSUMER METADATA
################################

class globalConsumerDB(db.Model):
   glob_id = db.Column(db.String,primary_key=True,nullable=False)
   rrindex = db.Column(db.Integer,nullable=False)
   topic = db.Column(db.String,nullable=False)
   localConsumer =
db.relationship('localConsumerDB',backref='globalConsumer',lazy=True)
   brokerCnt = db.Column(db.Integer,nullable=False)

class localConsumerDB(db.Model):
   id = db.Column(db.Integer,primary_key=True,nullable=False)
   local_id = db.Column(db.Integer,primary_key=False,nullable=False)
   broker_id =
db.Column(db.Integer,db.ForeignKey('broker_meta_data_db.broker_id'),nullab
le=False)
   glob_id =
db.Column(db.String,db.ForeignKey('global_consumer_db.glob_id'),nullable=F
alse)
   partition = db.Column(db.Integer,nullable=False)
```

```
############################## BROKER META DATA #################



class BrokerMetaDataDB(db.Model):

   db_uri = db.Column(db.String,primary_key=False,nullable=False)
broker_id = db.Column(db.Integer,primary_key=True,nullable=False)

   url = db.Column(db.String,nullable=False)

   docker_name = db.Column(db.String,nullable=False)

   last_beat = db.Column(db.Float,nullable=False)

   docker_id = db.Column(db.Integer,db.ForeignKey('docker_db.id'))

   localProd =
db.relationship('localProducerDB',backref='broker',lazy=True)

   localCons =
db.relationship('localConsumerDB',backref='broker',lazy=True)



############################### DOCKER METADATA
#############################################

class DockerDB(db.Model):

   id = db.Column(db.Integer,primary_key=True,nullable=False)

   brokers =
db.relationship('BrokerMetaDataDB',backref='docker',lazy=True)



############################### FOR Manager
#############################################
```

```
class ManagerDB(db.Model):

   id = db.Column(db.String,primary_key=True,nullable=False)

   url = db.Column(db.String,nullable=False)
```

Read Broker Manager

Both the read and write managers have the same implementation. It will behave as a read manager if the IS_WRITE environment variable is set to False. Otherwise, it's a write manager. The read managers only handle the dequeue operations. The write manager selects a read manager at random while handling a dequeue operation.

Synchronization Maintenance b/n Read and Write Managers

There exists a shared database between the write manager and the read managers to maintain sync between them. The write manager is the only one who writes to the database and the read managers can only query for reading data. This method has the following pros and cons:

Pros:

- Simple implementation
- Ensures common knowledge of any update at the same time between the read managers.

Cons:

- Performance and scalability are poor as there exists a central database.
- In the event of a database failure entire system collapses.

Dockers for deploying Managers and Brokers

Since using VMs locally using a virtual box would turn out to be quite demanding in terms of computation, we are using docker for containerizing each manager and broker.The details regarding the IP addresses and ports are stored in the metadata database of the manager.

Library

Library implementation was very similar compared to the previous assignment. The only differences were, that instead of createProducer and createConsumer, we only have one createClient which is used for the creation of both the producer and the consumer.

Testing

We tested the application using the following tests:

- We tested all the success and failure responses of each endpoint using the library
- 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.

Challenges faced

- Getting familiarized with docker and shell scripts to extract docker metadata information
- Implementing crash recovery by running a background process

Distributed Systems Assignment-3

(Consensus module using RAFT)

The main objective of this assignment is to provide fault-tolerance and ensure consistency among the brokers and the managers. The assignment is divided into 3 parts and below for each of the three parts we discuss the implementation details, design choices, and trade-offs.

RAFT Library

We use the PySyncObj library for RAFT. PySyncObj is a Python library for building fault-tolerant distributed systems. It provides the ability to replicate application data between multiple servers. The library is very user-friendly, and internally handles all the functionalities of RAFT which includes leader election, and persistent, and synchronized RAFT logs. For more details refer to the repo: PySync

PART-I

This part of the assignment was mostly for the learning purpose of RAFT. It consists of two classes:

- User: Stores all the user-related information i.e. a mapping from username and account number, and provides the interface for checking the validity of users and accounts.
- Account: Stores all the account-related information i.e. the amount held in the account, and provides the interface for all account-related operations like withdrawal, deposit, transfer, etc.

The implementation maintains consistent information across all the ATMs, handles network partitioning, and also provides fault tolerance.

PART-II

This part of the assignment aims to provide fault tolerance and consistent states across all the brokers. The key data structures and ideas remain the same in our broker implementation however, we had to completely change the way of implementation in order to incorporate the RAFT library correctly. The key changes are described below:

- ***Monolithic Design:*** The broker now contains a single class containing all the necessary data structures. This was required to ensure using a *single PORT* for the RAFT process. Also, to ensure that all the state updates are performed in the correct order respecting all the dependencies between them we constructed a single method that will apply the different updates to the state (Note that it still handles concurrent requests).

- ***Maintaining consistent IDs:*** We need to maintain consistent IDs for the replicas (e.g. topicID, consumerID, producerID, etc). In previous assignments, each broker would maintain their own set of IDs which can be different. To ensure that they aren't we moved the ID allocation to the Write Manager which with each request will send the ID to the corresponding broker to use for the corresponding data.

- ***Multiplexing updates across brokers:*** Since all the brokers are RAFT nodes the library sends all the updates to all the brokers. To ensure that brokers only apply those updates that are meant for them we pass an ID_LIST from the Write Manager with each update which contains which brokers should apply the update.

- ***Fault Tolerance:*** The Write Manager provides fault tolerance by trying each of the brokers corresponding to each partition. In case it detects a broker down it tries to recover that broker in a separate thread.

- ***Handling of Concurrent Requests:*** We ensured that concurrent requests are handled by using suitable locks as scarcely as possible so that the performance hit is minimal.

Finally, we tested the implementation by running multiple concurrent producers and, consumers to stress test the application. We also noticed that each broker after its crash-recovery gets back all the states correctly.

PART-III

This is the final part of the assignment which aims to make the managers consistent and fault tolerant. We consider the following questions and build up the solution:

1. *Is the Broker Manager a single point of failure currently? If yes, how can we avoid that?*

   Proposed Solution: The Broker Manager till assignment-2 is a single point of failure. Moreover, it contains a central DB which is a huge bottleneck for the system. To avoid this we use multiple synchronized managers with independent DB and elect one of them as the Write Manager via a Voting process. In case the current Write Manager goes down the voting process reelects a WM from the remaining alive managers. So, it's no longer a single point of failure.

2. *Would you use Raft for managers as well? If yes, would there be one Raft cluster for all read/write managers or separate clusters for read and write managers?*

   Proposed Solution: Yes, we would use RAFT for managers as well. In our opinion, there is no difference in implementation and data structures between WM and RM except for the responsibilities. So, we consider each of the managers as an RM and, give one of

them the special privilege of WM using RAFT leader election. Therefore, a WM is also an RM.

3. *Would other read replicas detect the primary manager going down and taking its role? Would the secondary have up-to-date information in this case?*

   Proposed Solution: The RAFT managers are always synchronized with each other thanks to RAFT. Since PySyncObj automatically performs reelection in case the leader goes down, we used the library as is.

4. *What other design choices can be made to improve the overall system's scalability and reliability while not giving up on consistency?*
   Proposed Solution:
   - Caching is one of the strategies that can be used to increase the system's scalability. This is partially fulfilled in our implementation as we have stored all the data structures in memory.
   - Using Asynchronous IO can improve network latency a lot.
   - Better load-balancing strategies can be used to increase the system's scalability. To this effect, we have used a modified round-robin algorithm for load-balancing. For more details refer to assignment-2 documentation.

Limitations & Future Scope

- Incorporating Asynchronous IO to improve upon the network latency of requests to brokers from WM.
- A better Caching Policy can be used to improve the performance of the system.
- We have used a static load-balancing technique. We can also go for Dynamic load-balancing for our system.