PA1 : Design Document

CS60002 Distributed Systems

TechStack

The main part of the assignment was based on configuring the HTTP endpoints for the distributed-topic-queue. Flask framework was chosen as the barebones of the assignment for a couple of reasons: a) working with bare HTTP servers is cumbersome, Flask provides with a clean workflow for the same, b) Flask is based on Python which enables all the varied python packages (like threading for locks or SQLAlchemy).

For ease of API testing, we also utilised the Postman API Tester, which enables a GUI based dashboard for sending requests and viewing responses. Here is a web published version of the same: https://documenter.getpostman.com/view/25366393/2s935hS7ur

For persistence of the queue, PostgreSQL was selected that provides the ACID guarantees of databases.

Part A

The first part of the assignment required a simple in-memory implementation of the topic-queue that supported the given API endpoints.
For this part, since the queue is in memory of the server, all operations done by various producers and consumers are executed on the same memory segment.

That is, it is highly likely that there may be data race conditions due to simultaneous writes by different producers or consumers (while registering).

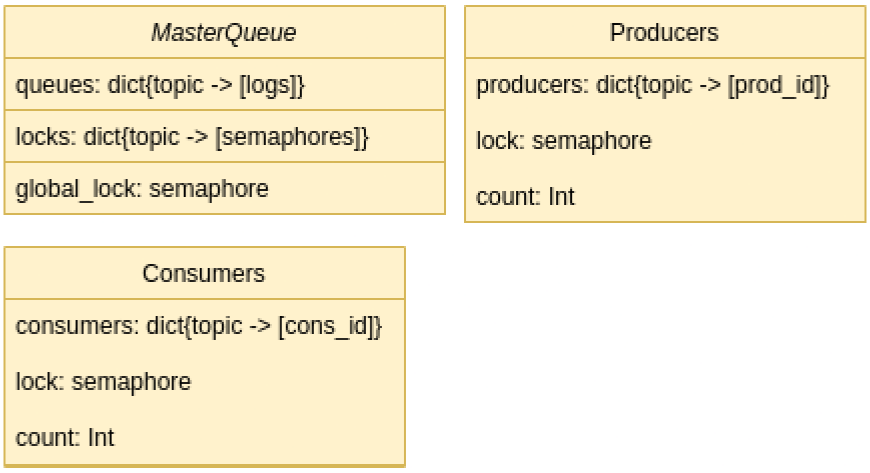
To prevent such race conditions and inconsistent state, the use of mutex or semaphore or any other prevalent locking mechanisms can be used.

Due to their flexible nature and ease of use, semaphores were used to ensure single execution of critical segments of code. The main critical areas of code were identified as:

1. Adding a topic
2. Producing a message
3. Registering a producer
4. Registering a consumer

For the sake of this part of the assignment there are several read operations. Since reads do not lead to typical race conditions and locking has negative effect on the latency of execution, it was warranted not to use locks on reading (that is, the famous readers-writers conundrum and its rather convoluted solution).

The following classes were considered to be the core queue data structure.

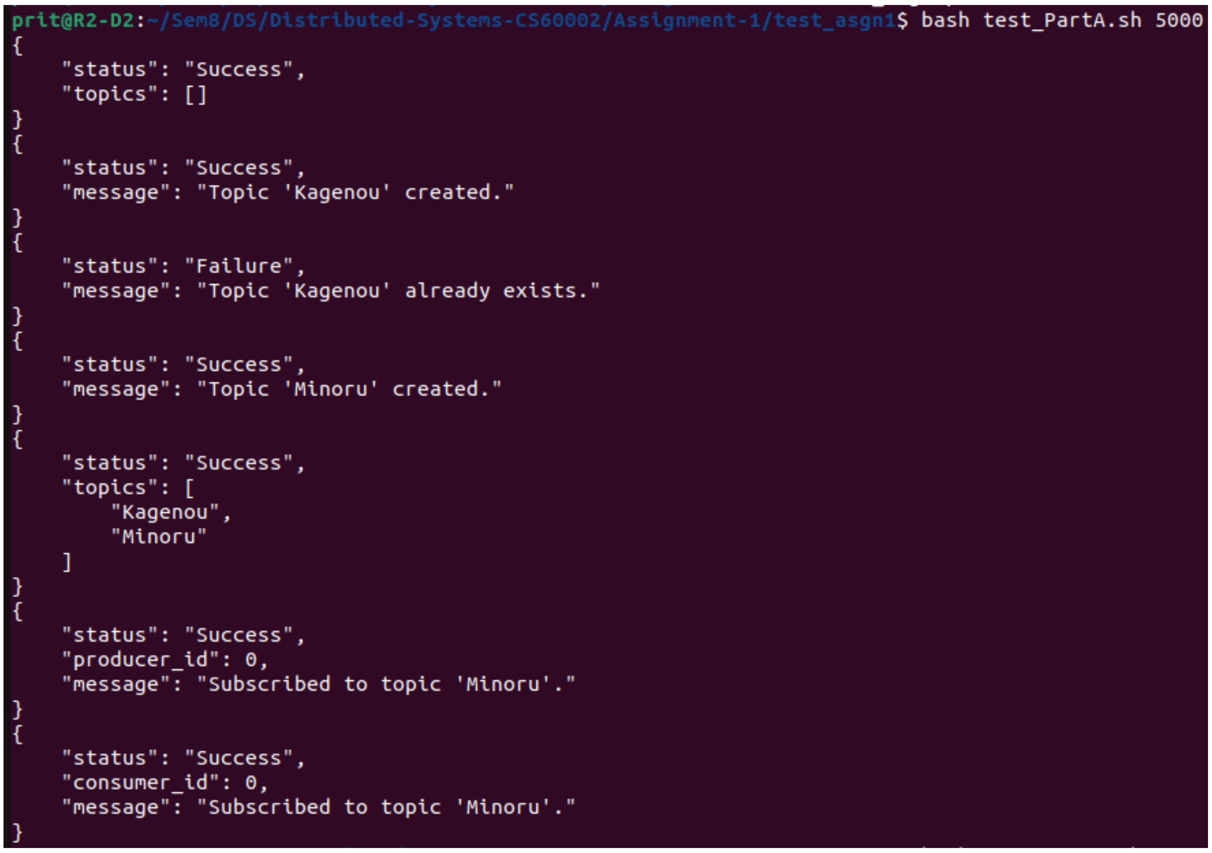

MasterQueue
queues: dict{topic -> [logs]}
locks: dict{topic -> [semaphores]}
global_lock: semaphore
Producers
producers: dict{topic -> [prod_id]}
lock: semaphore
count: Int
Consumers
consumers: dict{topic -> [cons_id]}
lock: semaphore
count: Int

Some key points about the design:

1. The MasterQueue class stores a dictionary that is keyed on the topic names and the value stored at the key represents the log messages committed to that topic.
2. The list of keys of the MasterQueue class itself serves as the list of topics.
3. To add (or remove in future versions) new topics, a central lock on the dictionary is mandated.
4. Moreover, writes in one topic should not ideally stop writes in another, hence, we also maintain topic-wise locks, dictionaried on the topic name. This is the lock a producer will acquire before logging a message to the given topic.
5. The Producers class is a structure to store the mapping of producers to topics. This data structure also has a central lock to be acquired when a producer registers.
6. By design, the producer_id of the producer is unique, even across topics. To keep track of the next id, we, thus, maintain a global counter. The purpose of this design choice is nothing but consistency with the behaviour that arises in PartB where we use SQL tables whose primary index is this id itself.
7. The Consumers class mimics the above functionality for the consumers. The discussion on producer_id applies to consumer_id as well for the reasons already mentioned.
8. A major design for the consumer is: new consumers will have access to all logs from the beginning of the production; and not only the once produced post their registration to the topic.

Testing

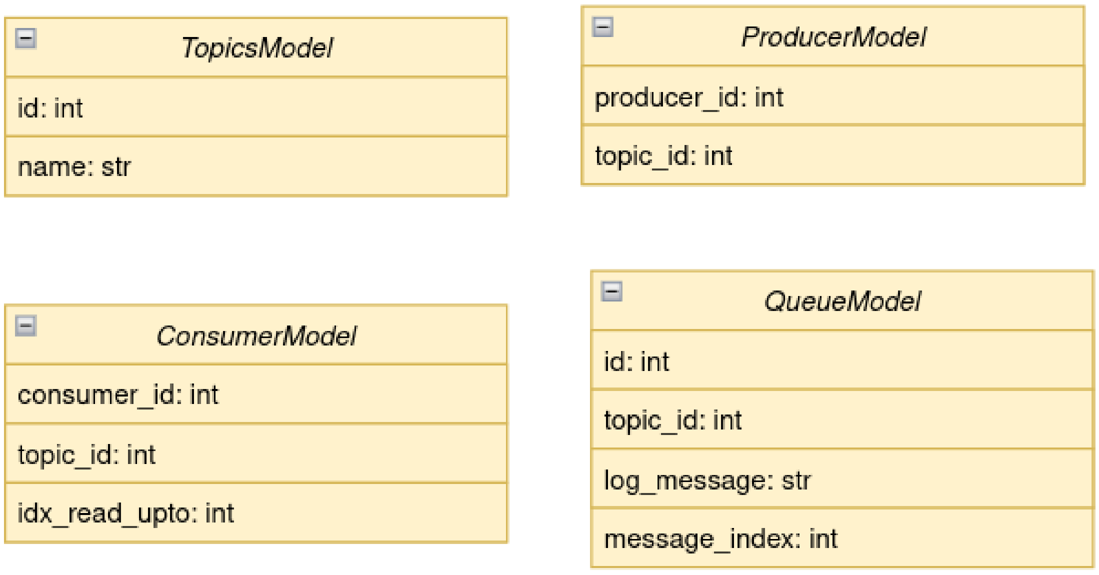
For partA, the given test script “test_partA.sh” was successfully executed.
The response for the same is mentioned in the Assignment-1 directory on the repository under the name: “test_partA_responses.txt”.



```
prit@R2-D2:~/Sem8/DS/Distributed-Systems-CS60002/Assignment-1/test_asgn1$ bash test_PartA.sh 5000
{
    "status": "Success",
    "topics": []
}
{
    "status": "Success",
    "message": "Topic 'Kagenou' created."
}
{
    "status": "Failure",
    "message": "Topic 'Kagenou' already exists."
}
{
    "status": "Success",
    "message": "Topic 'Minoru' created."
}
{
    "status": "Success",
    "topics": [
        "Kagenou",
        "Minoru"
    ]
}
{
    "status": "Success",
    "producer_id": 0,
    "message": "Subscribed to topic 'Minoru'."
}
{
    "status": "Success",
    "consumer_id": 0,
    "message": "Subscribed to topic 'Minoru'."
}
```

Part B

In this part, we were required to add persistence to the distributed queue. Hence we integrated PostgreSQL with our existing flask APIs and designed the following database schema:



The tables are - *TopicsModel*, *ProducerModel*, *ConsumerModel*, and *QueueModel.* The attributes/fields for each of these tables are described as follows:

1) TopicsModel:
- id: It is the primary key of this table which stores a unique integer for each topic
- name: It is the name of the topic stored in string format which has to be mandatorily unique for different topics

2) ProducerModel:
- producer_id: It is the primary key of this table which stores a unique integer for each producer which registers for a particular topic
- topic_id: it stores the integer id of a topic as a ForeignKey denoting the topic for which the producer is registering.

3) ConsumerModel:
- consumer_id: It is the primary key of ConsumerModel Table which provides a unique integer key for each consumer which registers for a particular topic
- topic_id: It is the integer id of a topic stored as a ForeignKey denoting the topic for which the consumer has registered.
- idx_read_upto: It stores the integer index/offset of the log messages in the queue upto which a consumer has read so far. In our design, we decided to provide the consumers all the log messages present in a particular topic specific queue. Hence, during creation of a consumer, we initiate the idx_read_upto value to 0. Further we use 0-based-indexing to fetch or count the remaining messages which are yet to be read for this consumer.

4) QueueModel:
- id: It is the primary key of this table which stores a unique id for a log message belonging to a particular topic
- topic_id: It stores the integer topic id as a ForeignKey for which we are storing a log message.
- log_message: It stores the actual log message in a string format.
- message_index: This field stores the integer value of the current index of the log message in the topic specific queue. In simpler terms, it stores the position of the current message in the queue of the specified topic.

For visualising how the QueueModel works, you can think of it as a single table for storing multiple topic specific queues. For each log message, we keep track of the topic of the message and along with that, we also store the position in which we are pushing the log message for this queue (by calculating the number of already inserted messages in the queue of that topic). This design helps us to query the database faster to insert and retrieve a log message as well as count the number of unread messages for a consumer.

Since operations of insertion of new message and producer registration are not atomic: for example in insertion of new log message, first we fetch the last inserted message index for each topic and then increment the message index to be inserted. Such sections of code are thus critical and require mutual exclusion to ensure consistency. However, here only three locks are used as opposed to topicwise locks used in PartA. This is primarily because the rest is handled in an ACID-compliant fashion by the SQL database itself.

Part C

This part of the assignment required to create a library that provides simple, user-friendly functions that would use the API endpoints created previously. This can be done by creating a module with classes containing sets of similar functions. An instance of the class can then be used to call these functions.

The module is being designed such that it handles the producer and consumer IDs internally to make it easier to use for the user.

As the endpoints created had the following features:

- Topics (/topics)
    - Get the list of topics (GET)
    - Register a topic (POST)
- Producer (/producer)
    - Register a producer to a topic (/register - POST)
    - Produce message to a topic (/produce - POST)
- Consumer (/consumer)
    - Register a consumer to an existing topic otherwise registers the topic first (/register - POST)
    - Consume a message from a topic (/produce - GET)
- Size (/size)
    - Get the number of messages left to be read (GET)

The module can be designed with classes as:

- MyProducer()
    - __init__(self, topics, broker): Constructor that calls /producer/register for the topics given
    - send(self, topic, message): calls /producer/produce for adding message to topic
- MyConsumer()
    - __init__(self, topics, broker): Constructor that calls /consumer/register for the topics given
    - has_next(self, topic): checks if there are messages left to be read from the topic
    - get_next(self, topic): calls /consumer/consume for getting a message from the topic
    - size(self, topic): calls /size for getting the number of messages left to be read

Both these classes would contain instance variables:

- topics: dict{topic → producer/consumer_id}
- broker: Broker through which APIs would be called

The Team

Sr No	Name	Roll Number
1.	Aaditya Agarwal	19CS10003
2.	Pritkumar Godhani	19CS10048
3.	Debanjan Saha	19CS30014
4.	Akarsh Singh	19EE10003
5.	Parth Tusham	19CS30034
6.	Sayantan Saha	19CS30041