Distributed Systems

Assignment - 1

Group Members:
Shashvat Gupta - 19CS30042
Satvik Bansal - 19CS10053
Sajal Chhamunya - 19CS10051

Distributed Logging Queue

Problem Statement

Design and implement a distributed logging queue with producers and consumers who can push or pop message logs specific to a topic. The queue needs to be persistent. Also, develop client libraries for easy usage of the Queue API.

Library Used

Python is used as the programming language and the frameworks **FastAPI** and **SQLAlchemy** for HTTP API and Database ORM, respectively.

Implementation Approach

Firstly, we created a Logging Queue that would support multithreaded interactions with the help of appropriate locks. The design of the system is shown in the System Design section below.
This version of the logging queue works directly in the system's memory. This implies that on system shutdown, the memory would get erased, and the system's current configuration vanishes. Hence, we store the current configuration of the system in a database. Note that the system still runs on memory only. The system now also saves its state on the database. In case of a shutdown, the memory is erased, but the database persists. On system startup, the system reads its configuration from the database and loads it into memory. Thus, we get persistence even on system shutdown.
Lastly, we developed a client library to make it easier for programs to use the API of Logging Queue. This library is relatively simple and uses the requests package of Python.

System Design

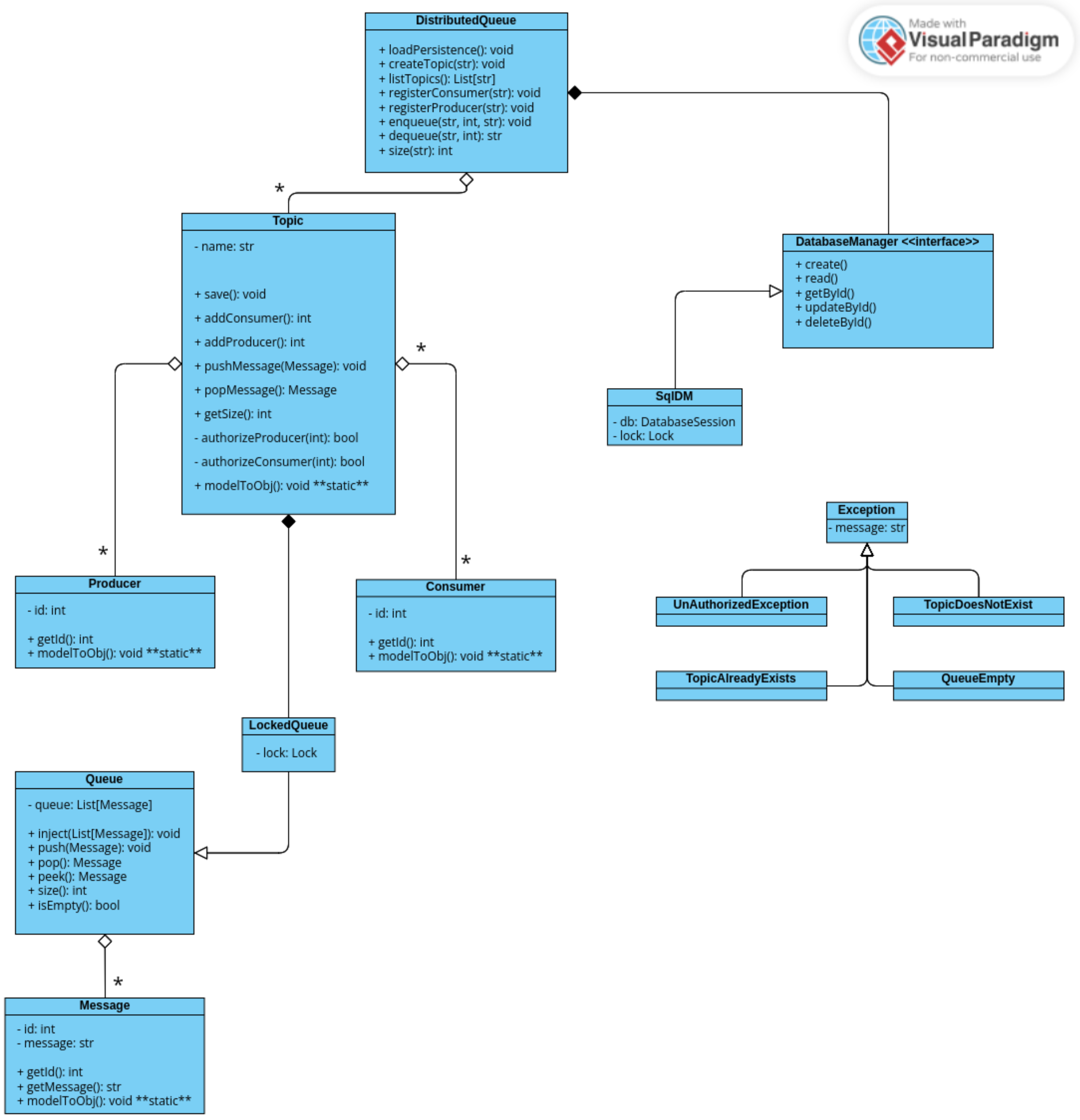

Fig 1: Class diagram of the Distributed Logging Queue System

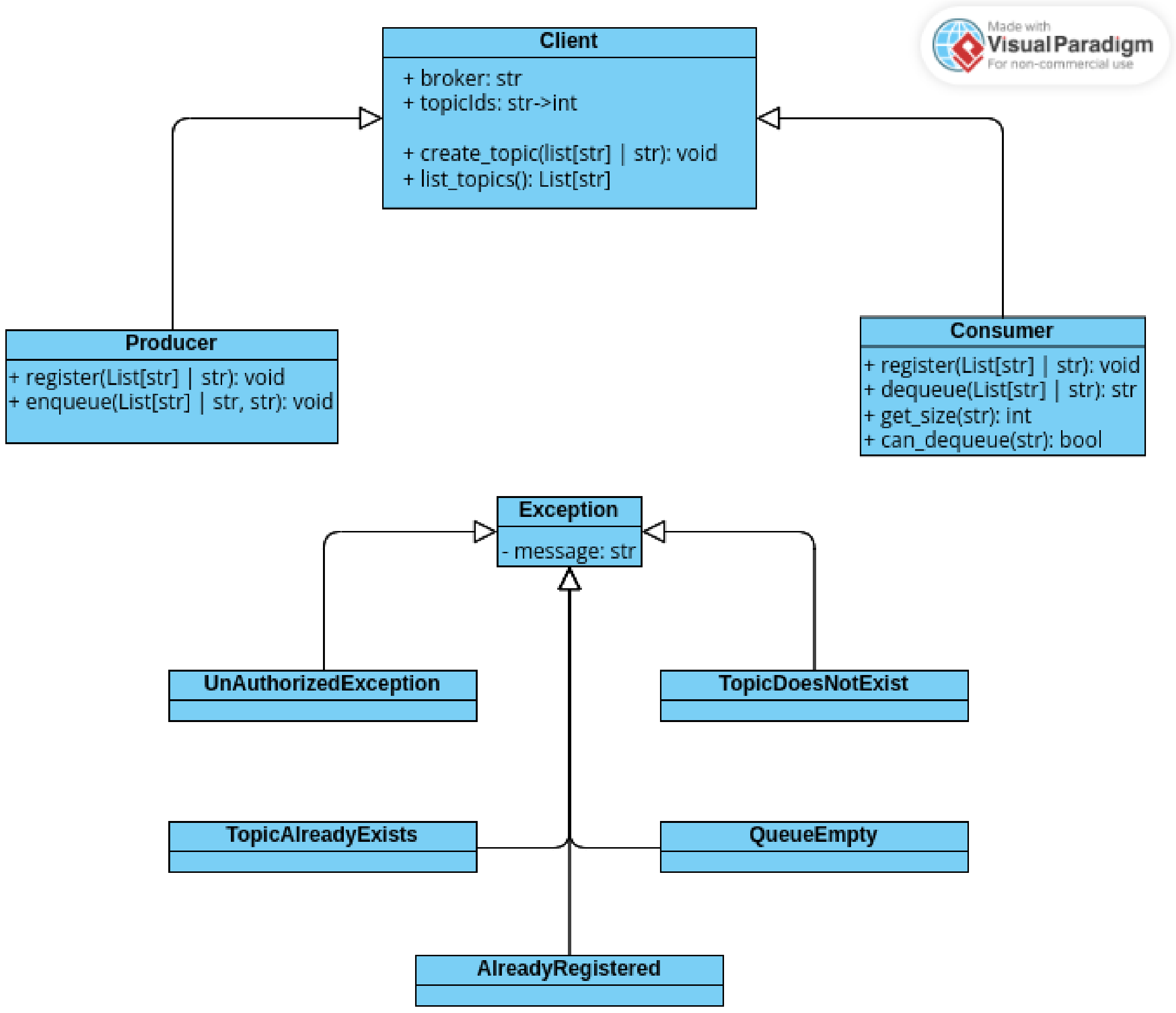
Made with Visual Paradigm For non-commercial use
Client
+ broker: str
+ topicIds: str->int
+ create_topic(list[str] | str): void
+ list_topics(): List[str]
Producer
+ register(List[str] | str): void
+ enqueue(List[str] | str, str): void
Consumer
+ register(List[str] | str): void
+ dequeue(List[str] | str): str
+ get_size(str): int
+ can_dequeue(str): bool
Exception
- message: str
UnAuthorizedException
TopicDoesNotExist
TopicAlreadyExists
QueueEmpty
AlreadyRegistered

Fig 2: Class Diagram of Client Library

Testing

We ran two types of tests, unit and application tests. Test cases and scripts for both are provided in myqueue/tests. We created separate scripts for producers and consumers in unit testing and tested each API endpoint using our developed client library. In application testing, we created custom threads for producers and consumers. We ran 5 producer threads and 3 consumer threads with pre-written log files. On conducting the above tests, we concluded that the system is bug-free, user-friendly and easy to use. To test persistence, we kept the application test running and stopped the server program in between. Then, we restarted the program and saw that the threads continued their operations.

Challenges Faced

Some challenges faced include designing a system to accommodate the requirements. Also, multi-threading and locking were hard to implement and harder to test. In our case, with a lot of trial and error, we successfully implemented and tested the system for the multi-threaded scenarios described above. Another challenge was storing and loading the system's configuration in the database to maintain persistence. We stored specific important models and their extended types to store the system's configuration in the database during the execution of the system itself. However, to load the configuration, we first load the configuration of the main broker, which in turn instigates each Topic (sub-system) to load their configuration. Each topic loads its configuration using the data linked to the topic in the database.