Distributed Systems

Assignment 2

Group Members

Shashvat Gupta - 19CS30042

Sajal Chhamunya - 19CS10051

Satvik Bansal - 19CS10053

Distributed Queue with Partition and Broker Manager

1. Problem Statement

Design and implement a multi broker distributed logging queue system by dividing a topic queue into multiple partition queues with producers and consumers who can push or pop message logs specific to a topic in any partition or a round robin fashion if not specified. The queue needs to be persistent.

Also, develop client libraries for easy usage of the Queue API.

2. Libraries Used

Python is used as the programming language and the frameworks FastAPI and SQLAlchemy for HTTP API and Database ORM, respectively. Additionaly PostgreSQL is used for broker managers, along with Threading library for Health Checks.

3. Implementation Approach

There are multiple broker managers, who can interact with brokers but any consumer or producer can only interact with these broker managers. Out of these broker managers, one is assigned with write privileges while all the others have only read access. This implies that creation of new brokers, registration of consumers and producers, and job creation can only be done through the write access broker manager, while functions like getting the list of brokers, topics or jobs can be done by any broker manager.

The privilege assignment of the managers is done by adding an environment variable that is checked for read or write access and is used while creating a new manager. All of these managers share the same database and are indeed *replicas* of each other, with only the difference of privileges. This difference is implemented by disallowing some api end points for the managers with only read access.

Consistency is maintained by loading the database into the memory of these read only managers each time any read call is made. With less complex functions such as get a list of brokers or the list of topics, the whole database is queried. Otherwise by using the Write After Logging method, only the updates are loaded, in functions such as size or dequeue.

These Broker Managers would then coordinate between multiple brokers and handle the requests of consumers and producers, by directing them towards their requested broker. The design of the system is shown in the System Design section below.

To incorporate *persistence*, 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.

As multiple brokers can be created and can also be destroyed, *service discovery* is also implemented by sending a message to the broker manager, each time a broker is initialised, who can then change the data to reflect the addition of the new broker.

To check on the health of the connected components like brokers to the broker manager, each connected component sends a heart beat request to the broker manager in every 15 seconds. If a heartbeat is not received in the next 60 seconds, that is 4 heartbeats are missed, then the connected component is declared inactive/dead.

Lastly, we developed a client library to make it easier for programs to use the API of the distributed Logging Queue. This library is relatively simple and uses the requests package of Python.

4. System Design

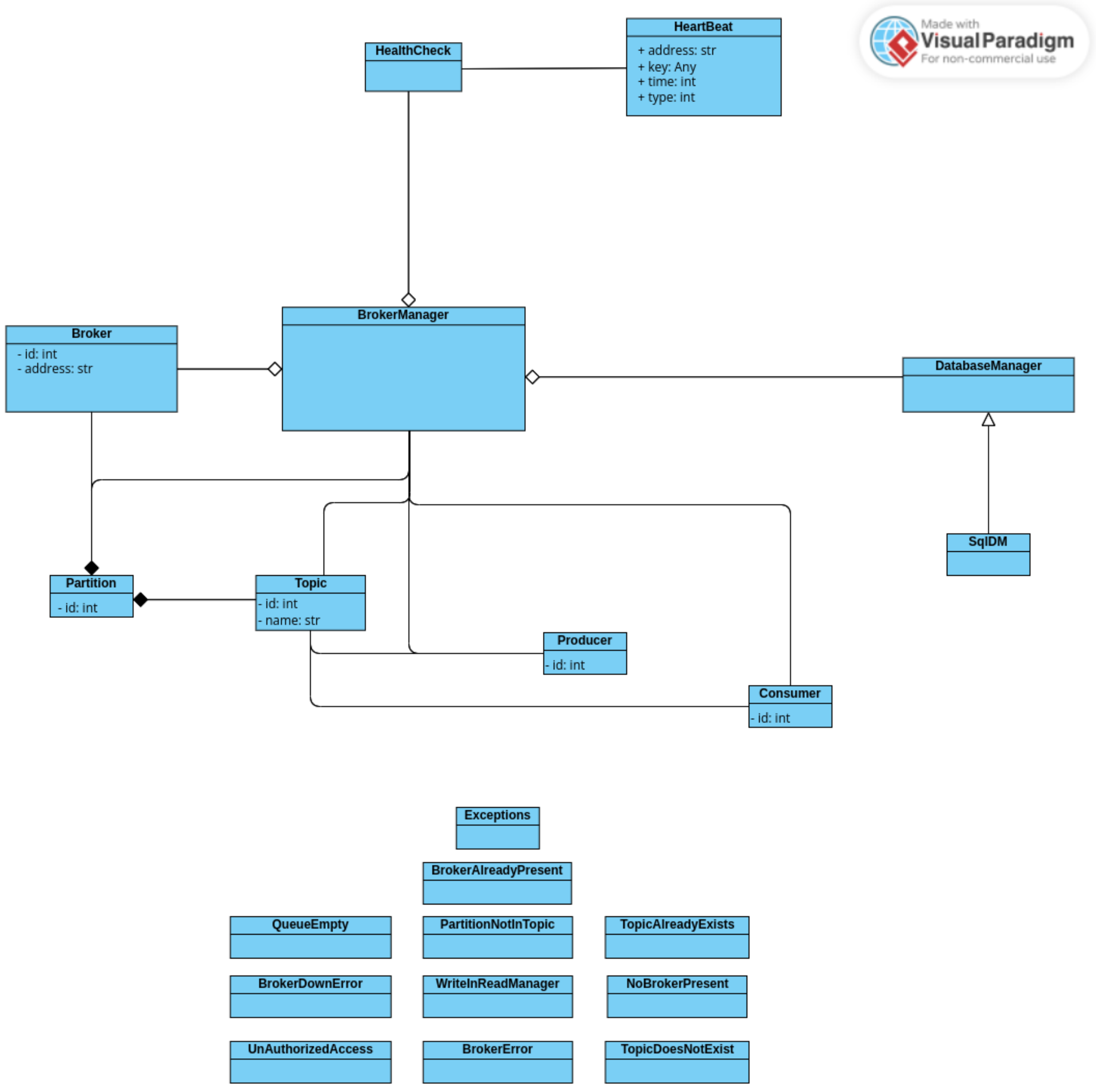

Fig 1: Class Diagram of Broker Manager

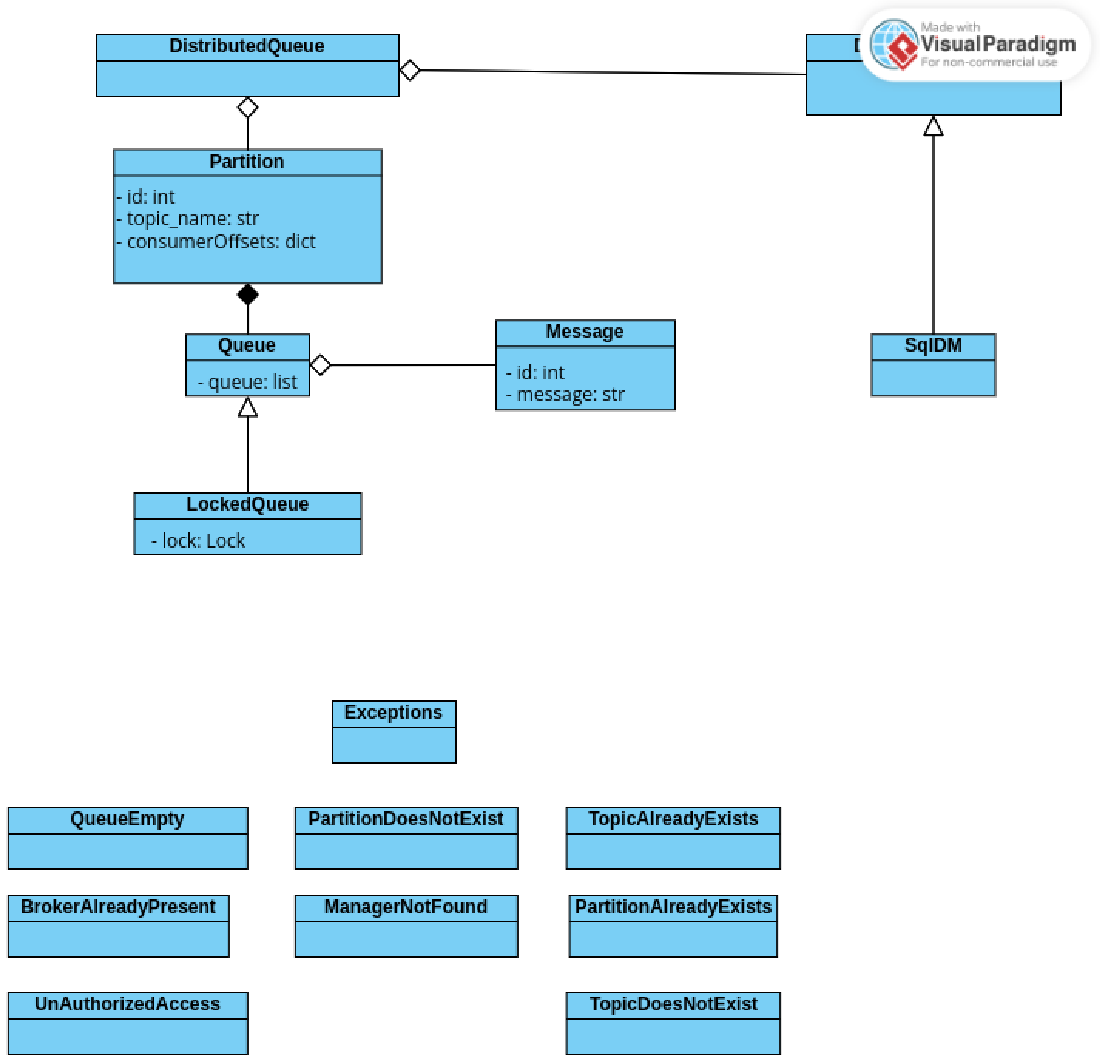

Fig 2: Class diagram of broker

5. Testing

We ran two types of tests, unit and application 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 brokers, managers, producers and consumers. We ran 5 brokers, 2 read only managers and a write access manager, along with 3 producer threads and 2 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.

6. Challenges Faced

The major challenges faced were designing a system that ensuring consistency across all managers, and how to notify every manager of an update. This was finally done by keeping a shared database and fetching complete data or the updates on requirement by the read only managers. Had we used a query for each function call the software would have become slow and sending multiple update pings to every manager would have increased the amount of server calls.