Distributed Systems

Assignment 3

Group Members

Shashvat Gupta - 19CS30042

Sajal Chhamunya - 19CS10051

Satvik Bansal - 19CS10053

Consensus Module using Raft

1. Problem Statement

Part I

Implement a toy ATM network that simulates the behaviour of real-world ATMs. Each ATM can be simulated by a process that runs on a separate terminal window, accepts user commands and displays the appropriate output. Use an existing RAFT library for consistency of data across these processes.

Part II

In this part of the assignment, you will extend your broker to incorporate a NAT-like design using the RAFT protocol. This design aims to allow for fault-tolerant and consistent replication of partitions across multiple brokers without the need for additional physical hosts. Each partition should have multiple replicas across different brokers. Data consistency must exist between replicas of a partition using RAFT protocol.

2. Libraries Used

Python is used as the programming language and the frameworks FastAPI and SQLAlchemy for HTTP API and Database ORM, respectively. Additionally, PostgreSQL is used for broker managers and the Threading library for Health Checks. The raft-lite library is used to implement the RAFT consensus algorithm.

3. Implementation Approach

Part I

The system provides an abstraction called **ATMInterface** which corresponds to an ATM system. Each ATMInterface has a **RAFT** node associated with it which is an abstraction provided by the *raft-lite* used in the assignment. **ATMBloc** handles the business logic of the ATM which includes handling actions and providing and updating the status of balances through the RAFT Consensus algorithm. **Action** Interface has 4 implementations in the form of **Withdrawal**, **Deposit**, **Enquiry** and **Transfer** of charge. An action is

responsible for taking required inputs, handling transaction logic and updating balance status using ATMBloc and showing output in ATMInterface.

The state of balances is stored as a dictionary with names as keys and balances as values. In any transaction, the node pulls the current state from RAFT nodes, changes it according to transaction logic and updates the RAFT nodes with this new state. Since RAFT nodes work on the RAFT Consensus algorithm, all nodes have access to the latest current state and the ability to update it.

Part II

Multiple broker managers can interact with brokers, but any consumer or producer can only interact with these broker managers. Out of these broker managers, one is assigned write privileges, while all the others have only read access. This implies that creating new brokers, registering consumers and producers, and creating jobs can only be done through the write-access broker manager. In contrast, any broker manager can do functions like getting the list of brokers, topics or jobs.

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. These managers share the same database and are *replicas* of each other, with only the difference in 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 getting a list of brokers or the list of topics, the whole database is queried. Otherwise, only the updates are loaded using the Write After Logging method in functions such as size or dequeue.

To incorporate *persistence*, we store the system's current configuration in a database. Note that the system still runs on memory only. The system now also saves its state on the database. In 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 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 heartbeat request to the broker manager 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.

Each Broker has a RAFT node process associated with it. Abstraction of this process is used in a **RAFTAdapter** instance responsible for getting and updating the partition state stored in RAFT nodes. Hence, each partition has an instance of RaftAdapter associated with it. This is also used in **RAFTQueue,** an extension of older existing queues, as it gets the partition queue from RAFT nodes and appends new messages to the partition queues stored in RAFT nodes.

The state stored in RAFT nodes is a dictionary with the partition id as the key and a dictionary with the partition's state as the value. A partition state only requires two entities which are queue and consumer offsets. An example of a partition's state is as follows:

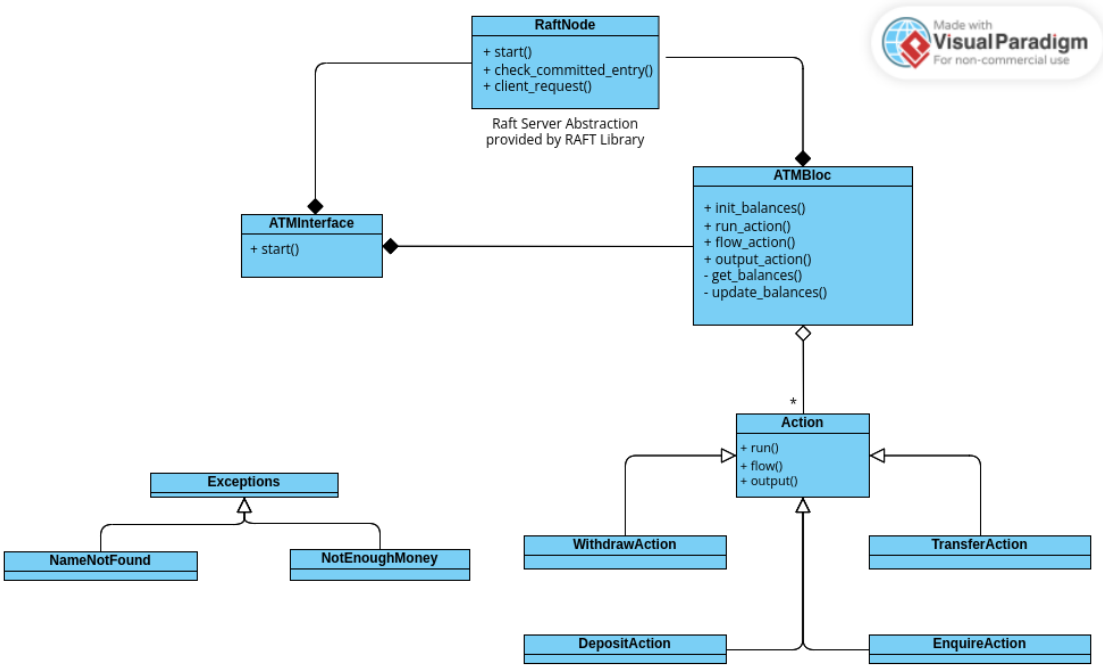
```
{
        'queue': [MESSAGE1],
        'consumerOffset: {0: 1}
}
```

RAFTQueue pulls its partition's queue, adds a message and updates the partition's state in RAFT nodes to enqueue a message. To dequeue for a consumer, the partition pulls the current state of the partition, reads the consumer's offset and passes the same to RAFTQueue, which returns the corresponding message, after which the partition updates the consumer's offset and updates the state of the partition in RAFT nodes.

Only a little gets changed for Broker Manager since it does not deal with RAFT nodes directly. However, whenever a new broker is added to the manager, all existing brokers need to be notified of the addition and the new node's raft node credentials so that existing brokers can connect to the raft process of the new broker. Also, the broker manager takes care of partition replication by storing brokers with an instance of a partition and selecting appropriate brokers for partition replicas. To service requests according to a partition, an appropriate broker with the partition's replica is decided in a Round Robin fashion.

4. System Design

Part I



Part II

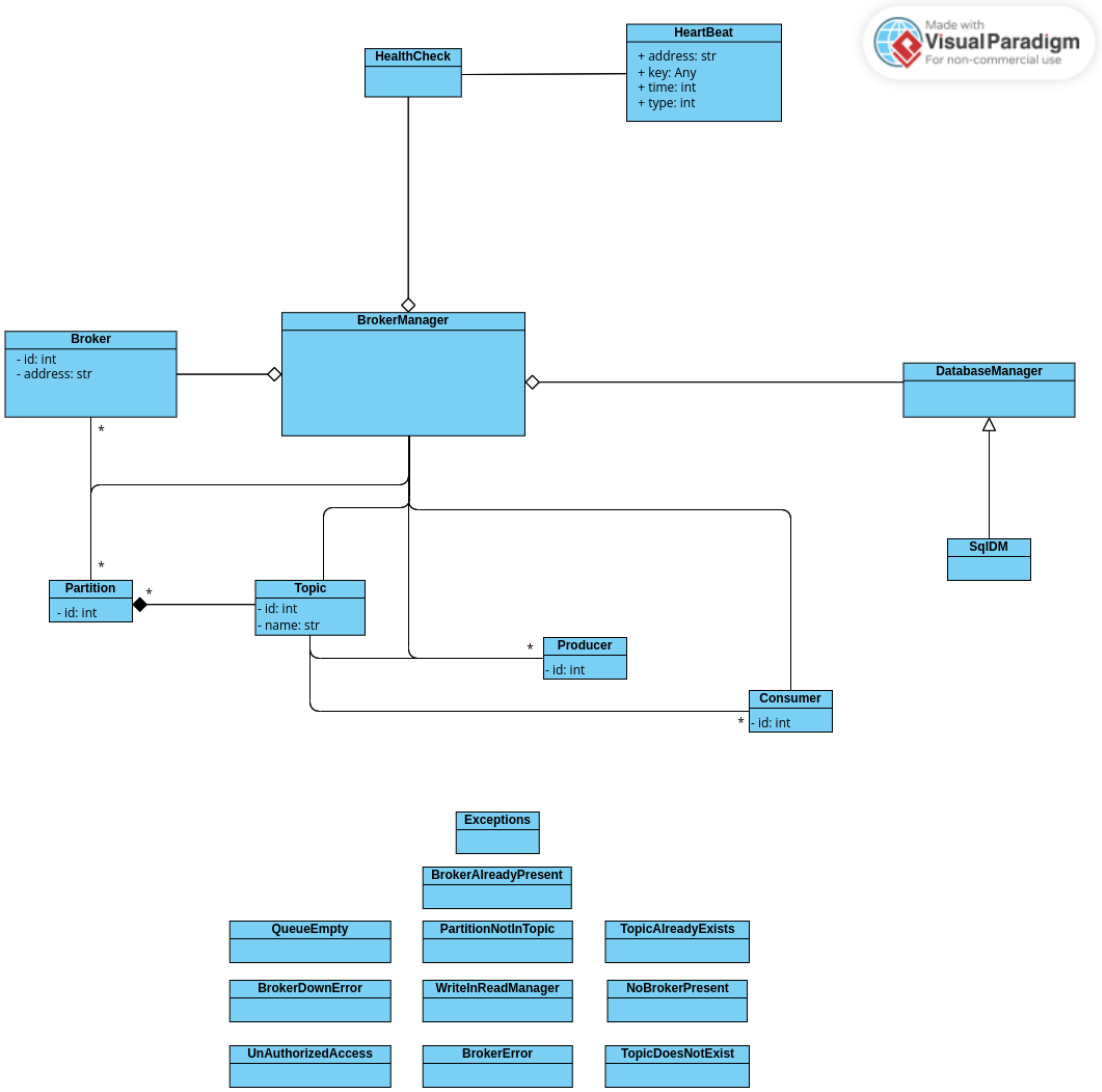


Fig 1: Class Diagram of Broker Manager

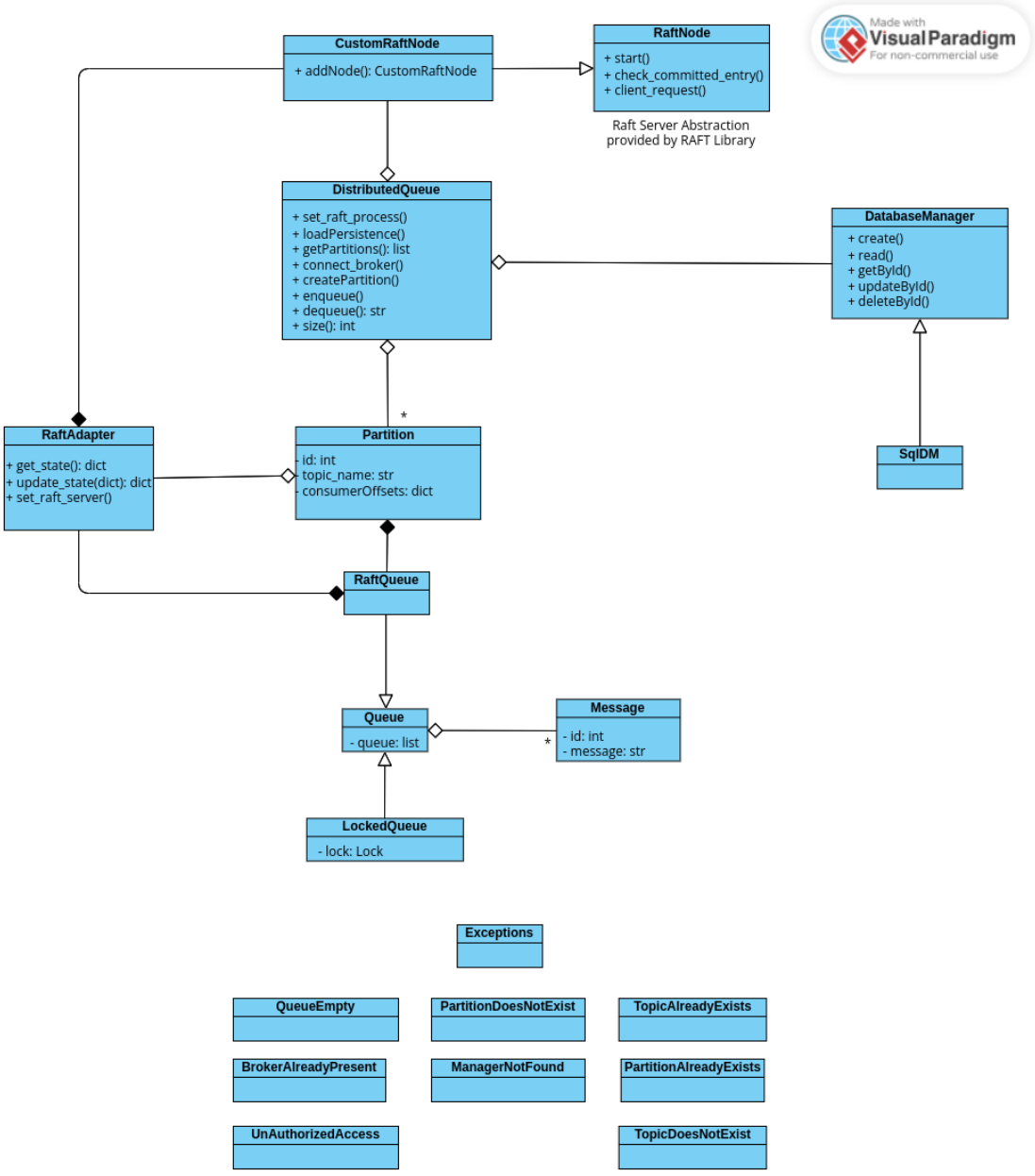


Fig 2: Class diagram of broker

5. Testing

We ran two types of tests, unit and application tests. In unit testing, we created separate scripts for producers and consumers 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, 3 producer threads and 2 consumer threads with pre-written log files. We conducted the above tests and 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 significant challenges faced included adding raft nodes to the existing network on addition of a new broker. This was finally overcome by storing current state in a database and restarting each raft node with the new broker connection. Also, it was challenging to handle replicas from broker manager's point of view. This was handled by storing all brokers in which a partition has a replica and then returning brokers in a round-robin fashion to service requests from a partition. This made the system stable and kept traffic on a single system minimum.

PART III

Q: Is the Broker Manager a single point of failure currently? If yes, how can we avoid that ?

A: The write broker manager is a single point of failure since, if the write broker manager goes down, there is no mechanism to elect a new write manager. Without any write manager, all write requests, like, enqueue etc, become inactive, and the system fails. Also, the heartbeat mechanism fails since there is not write manager to send heartbeats to.

Q: 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?

A: It would also be a good practice to use RAFT Consensus for managers. All read/write managers need to be part of a single cluster since read managers might not change the metadata, but they need access to the latest metadata to service requests.

Q: Would other read replicas detect the primary manager going down and take its role? Would the secondary have up-to-date information in this case?

A: Managers behave a lot like RAFT nodes themselves, in the sense that we can assume the leader of RAFT nodes to be the primary manager. Using this rule, whenever the primary manager goes down, raft nodes elect a new leader in its place. As soon as the new leader is elected, we can appoint it as the new primary manager and the manager sends

out a request to all brokers specifying himself to be the primary manager. If the primary broker were not processing an update when it went down, all other managers would have up-to-date information of the metadata.

Q: What other design choices can be made to improve the overall system's scalability and reliability while not giving up on consistency?

A: A different design choice could be using a raft node for each partition. This leads to more secure partition states, improved consistency and better scalability due to more raft clusters. Also, we can use RAFT consensus in the managers as discussed above. This would prevent the primary write manager from being a point of failure by quickly appointing a new primary manager. This improves reliability since the system has no single point of failure existing.