Assignment 2 Report

March 07, 2023

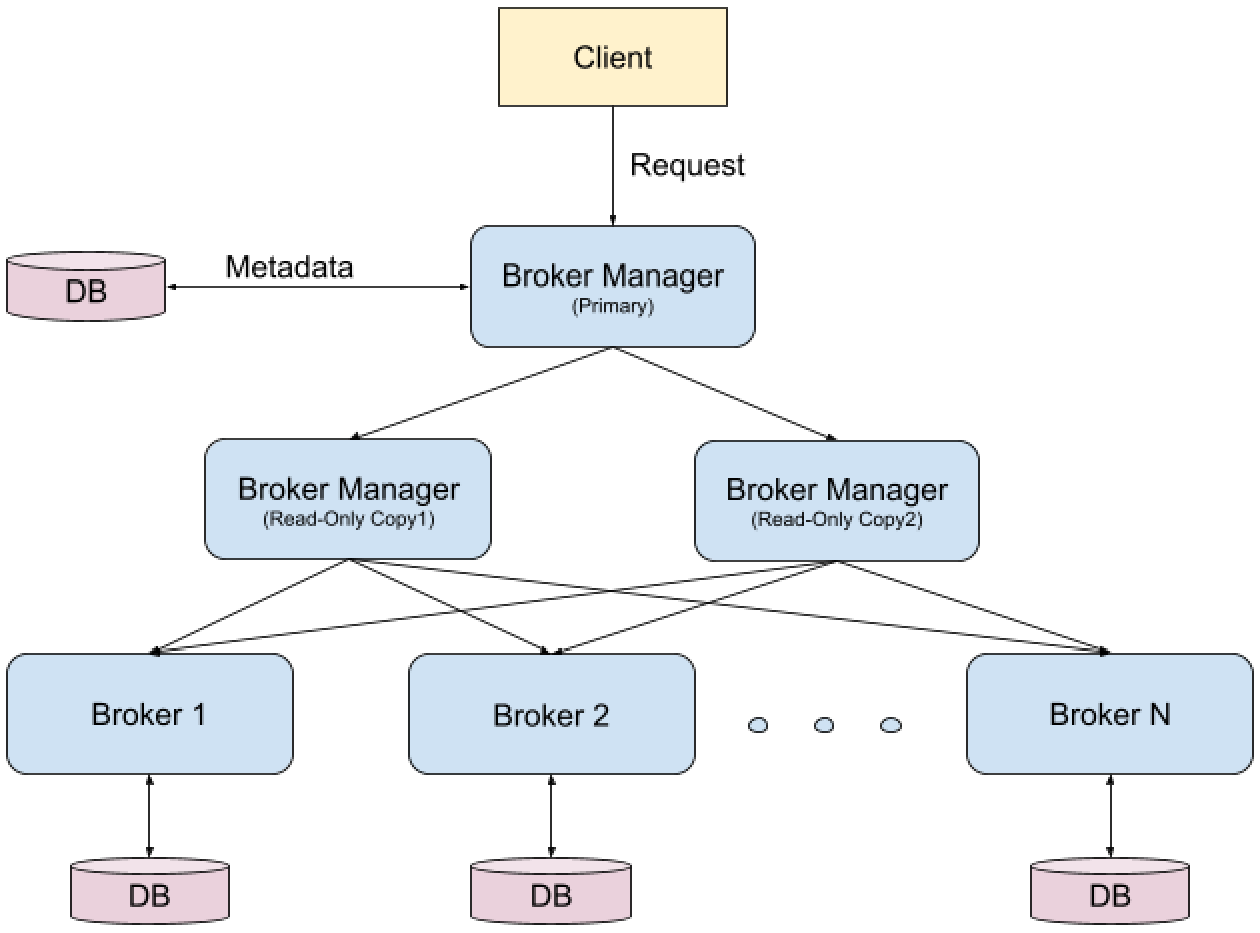
Group Members:

Abhilash Datta	19CS30001
Sunanda Mandal	19CS10060
Rohit Raj	19CS10049
Haasita Pinnepu	19CS30021
Matta Varun	19CS30028
Suguna Bhaskar Tirupathi	19CS10063

Overview

In this assignment, we have extended the basic broker implementation from Assignment 1 to a more sophisticated, multi-broker distributed queue system by dividing a topic queue into multiple partition queues. Partitioning is a common technique used in distributed queue systems to improve scalability and fault tolerance by allowing multiple brokers to handle different partitions of the same topic in parallel.

System Design



Broker Manager(Primary)

Implementation

This Broker Manager service manages topics, consumer and producer registration, and partition allocation. The Broker Manager is responsible for creating and maintaining metadata for the topics it manages.

The Broker Manager provides an API for creating topics, listing existing topics, and registering consumers and producers for specific topics. It also keeps track of the last partition ID for each topic and uses it to assign new partitions to producers.

The code uses Flask, a popular Python web framework, to define the routes and handle the HTTP requests. It also uses SQLAlchemy, an ORM library, to interact with a database to store the metadata for the topics and partitions.

There is also a function **getServerAddress**() to select one of the available Broker Manager copies to handle a request. The Broker Manager copies are identified by their IP and port number, and their health is periodically checked to ensure they are still alive.

Health Check

This script defines several functions that scan for active brokers and broker managers on a network using multiprocessing. It then defines a run_thread() function that continuously calls doSearchJob() every 10 seconds to search for active brokers and broker managers.

The **scanBroker**(port) function attempts to send an HTTP GET request to the specified port on the IP address 127.0.0.1 (i.e., the local machine). If the response status code is 200 and the response JSON contains the message "broker running", the function returns the port number. Otherwise, it returns None.

The **scanBroker_network**() function creates a multiprocessing pool with 50 processes and applies the scanBroker() function to each port in the range [5000, 5009]. It then appends any non-None results to a list and returns the list.

The **scanBrokerManager**(port) function is similar to scanBroker(), except it looks for a message indicating that a broker manager copy is running.

The **scanBrokerManager_network**() function is similar to scanBroker_network(), except that it uses scanBrokerManager() instead of scanBroker() to scan for active broker managers.

The **scanBrokerManagerPrimary**(port) function is similar to scanBrokerManager(), except that it looks for a message indicating that the primary broker manager is running.

The **scanBrokerManagerPrimary_network**() function is similar to scanBrokerManager_network(), except that it uses scanBrokerManagerPrimary() instead of scanBrokerManager() to scan for the primary broker manager.

Finally, the **doSearchJob**(brokerOrManager = 0) function calls the appropriate function based on the value of the brokerOrManager parameter: scanBroker_network() for 0, scanBrokerManager_network() for 1, and scanBrokerManagerPrimary_network() for 2. It then returns the IP address 127.0.0.1 and the list of active ports.

The **run_thread**() function simply calls doSearchJob() every 10 seconds in an infinite loop.

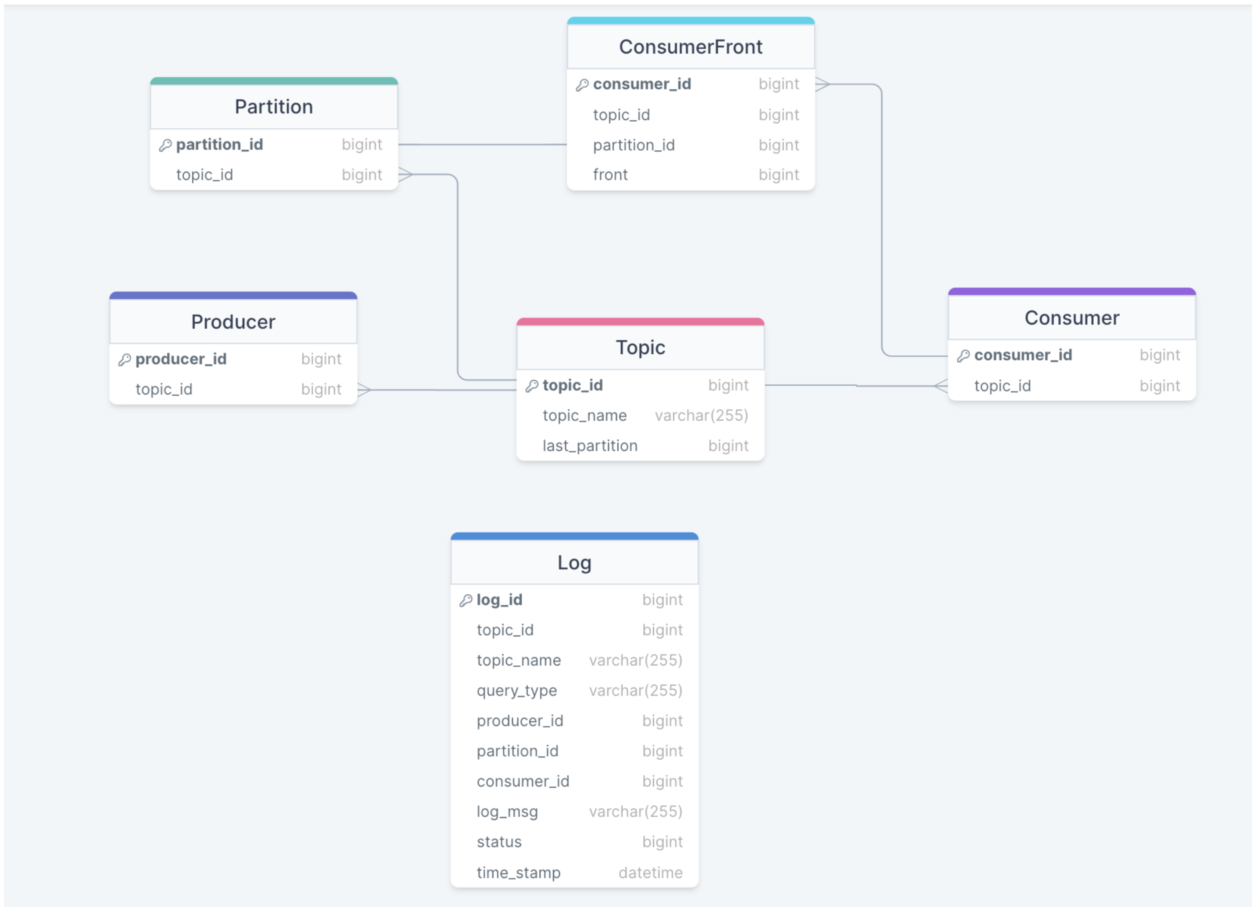
Request Forwarding

The write broker manager or the primary broker manager acts as a leader of the system and receives any request sent to the system. If the request involves POST requests, the leader handles itself. Otherwise, it forwards the requests to read-only brokers.

Handling partitions

Whenever a new topic comes, a partition is created in every broker online. Any new broker created after that won’t have partitions of previous topics.

Database Schema



Broker Manager (Read-Only)

The code for a broker manager creates and manages brokers in a distributed network, allowing producers to produce and consumers to consume messages from brokers. The code defines a Broker class with methods to check a broker's health and declare it dead or alive. It also defines a getServerAddress function that returns the address of a live broker. The code makes HTTP requests to the brokers to create topics, produce, and consume messages. The code also defines a function to set up the broker manager by backing up data from servers and setting up brokers in the network.

Brokers

The persistent server exposes endpoints for creating a topic, producing messages to a topic, consuming messages from a topic, and getting the size of a topic. The server uses SQLAlchemy for database access and defines the before_app_first_request function that creates the necessary database tables before the first request is processed.

The server exposes the following endpoints:

"/" : A simple endpoint that returns a welcome message when accessed.

"/status" : A GET endpoint that returns a JSON object indicating that the broker is running.

"/topics" : A POST endpoint that creates a new topic with a specified name and partition ID.

"/producer/produce" : A POST endpoint that enqueues a log message to a specified topic and partition ID.

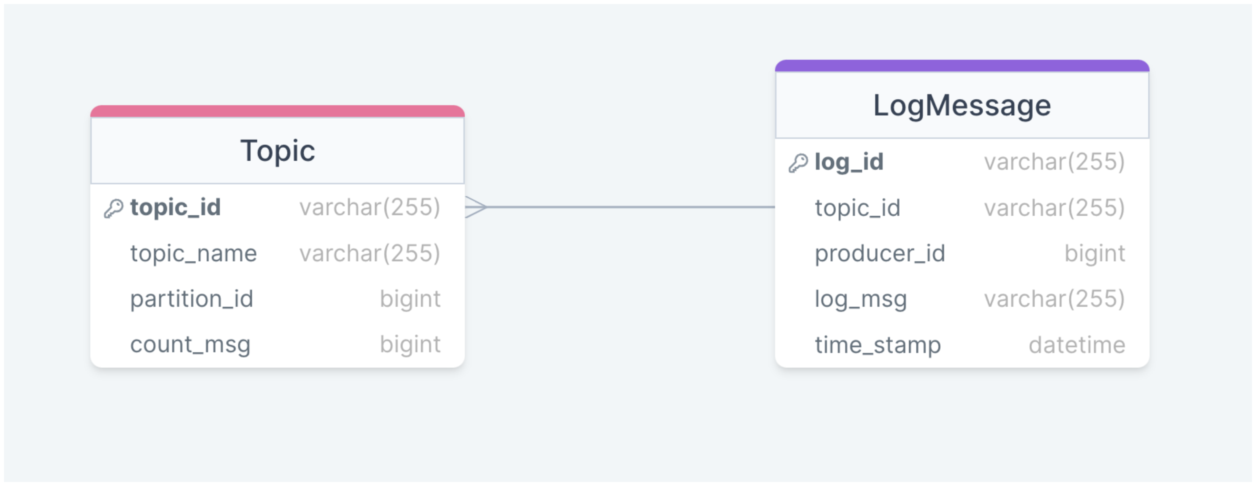
"/consumer/consume" : A GET endpoint that dequeues the next log message from a specified topic and partition ID.

"/size" : A GET endpoint that returns the number of messages in a specified topic and partition ID.

Each endpoint uses the dbBroker.AsyncDAL.DAL class for database access, which handles creating, reading, updating, and deleting data from the database. The DAL class uses SQLAlchemy's async session for asynchronous database access.

The server also defines two global variables, topic_lock and log_lock, which are used to synchronise access to topics and log messages.

Database Schema



Challenges Faced

1. Detecting Brokers

   Scanning the subnet IPs for possible brokers is a highly resource-consuming task and all the brokers need to be hosted in the same network. So we resorted to hosting brokers in different ports and running independently. Dynamically adding or removing brokers is also handled.

2. Deadlocks

   Due to its asynchronous nature and improper opening and closing of global locks, the system had gotten trapped into deadlocks.

3. Consistency among Broker Managers

   Maintaining metadata consistency is challenging as the calls are asynchronous. So updating fronts, and registering new producers and customers often lead to inconsistency. This is handled through a common database with the write-manager acting as the leader.

4. Write Ahead Logging

   Adding enough verbosity to understand how the system is working.