

Assignment 2

March 07, 2023

- Abhishek Kumar Sah
 - Aditya Piyush
 - Jayesh Chandan
 - Saumyak Raj
-

Components

Brokers

Single link to manager, no direct contact with producers/consumers

Responsibilities:

- Store partition of several topics.
- All queue operations(enqueue, dequeue).
- send heartbeats

Load Balancer

Responsibilities:

- Route write/read requests to correct manager

-
- Use read-only managers efficiently

Managers

One primary : handles all producer requests

Multiple secondary(Read Only) managers : handles all consumer requests(through a load balancer) If partition id is not specified, use round robin to assign partition id.

Responsibilities (primary manager):

- Takes Write requests through the load balancer
- Store metadata
 - registered producers
 - consumers to topic
- Allow producers to register to topics or a partition of a topic.
- Route requests to correct broker.
- Add/Remove brokers.
- Conduct healthchecks on broker/clients.

Responsibilities (secondary manager):

- A load balancer routes requests to a cluster of read-only managers.
- Allow consumers to register to a topic or a partition of it.

Maintaining Sync between Managers

File structure

-
- ServiceConsumers
 - Producer.py
 - Consumer.py
 - Test
 - Test files
 - ServiceProviders
 - WriteManager.py
 - ReadManager.py
 - ManagerModel.py (ORM)
 - WMWrapper.py(API)
 - RMWrapper.py(API)
 - Brokers
 - Broker.py (DistQ)
 - BrokerModel.py (ORM)
 - BrokerWrapper.py (API)

STRUCTURE OF THE APPLICATION(File Structure)

Load Balancer

Designed with the help of nginx(used primarily for load balancing), the load balancer is responsible for distributing the traffic between the two servers. It is also responsible for the health check of the servers. The load balancer is configured to check the health of the servers every 5 seconds. If the server is down, the load balancer will not send any traffic to that server. The load balancer is also configured to send 50% of the traffic to server 1 and 50% of the traffic

to server 2. The load balancer is configured to listen on port 80. All the requests are sent to the load balancer and the load balancer distributes the traffic to the servers.

Managers

There are 3 types of managers in our application :

- Write Manager
- 2 Read Managers

Responsibilities of the write manager: the WriteManager is a class responsible for managing writes to the distributed messaging system. The responsibilities of the WriteManager include creating a new topic, registering a producer, registering a consumer, registering a broker, enqueueing a message, incrementing an offset, and receiving a heartbeat from a broker.

Specifically, the WriteManager is responsible for the following functions:

- **create_topic(topic_name):** creates a new topic with the given topic_name and returns a list of partition IDs
- **list_topics():** returns a list of topics
- **list_partitions(topic_name):** returns a list of partition IDs for the given topic_name
- **register_producer(topic_name):** registers a new producer for the given topic_name and returns a producer ID
- **register_consumer(topic_name, partition_id):** registers a new consumer for the given topic_name and partition ID (if provided) and returns a consumer ID
- **register_broker(endpoint):** registers a new broker with the given endpoint and returns a broker ID


-
- **enqueue(producer_id, partition_id, message):** enqueues a message with the given message to the given partition_id for the producer with the given producer_id. When no partition_id is provided, the message should be enqueued to a random partition.
 - **inc_offset(topic_name, consumer_id):** increments the offset for the given topic_name and consumer_id
 - **receive_heartbeat(broker_id):** receives a from the broker with the given broker_id.

Responsibilities of the read managers:

The responsibilities of the ReadManager include managing read operations for a distributed message queue system. Specifically, the ReadManager is responsible for the following:

- **list_topics()** - This method should return a list of available topics.
- **list_partitions(topic_name)** - This method should return the partitions available for a given topic_name.
- **register_consumer(topic, partition_id)**- This method should register a consumer for the given topic and partition_id.
- **dequeue(topic_name, consumer_id, partition_id)** - This method should dequeue a message from the given topic, partition, and consumer. When no partition is provided, the message should be dequeued from a random partition in a round robin fashion.
- **size(consumer_id, topic_name)** - This method should return the size of the queue for the given consumer and topic. Send a periodic heartbeat to the manager to indicate the availability of the ReadManager.

Additionally, the Flask interface includes functions to handle HTTP requests for each of the above operations. The interface provides endpoints to handle GET requests for listing topics, listing partitions, and retrieving the size of a topic for a given consumer. It also includes an



endpoint for dequeuing messages via a GET request. Lastly, the interface includes a command line argument parser to allow the Flask application to be run with different configurations.

All the three managers (one write, two read) are synchronised as they are accessing databases that are synchronised.

Databases

There are four tables in the database, namely: BrokerMetadata , ConsumerMetadata, ProducerMetadata, and PartitionMetadata

The BrokerMetadata table stores information about the brokers in the system and contains the following columns: broker_id - a unique identifier for the broker endpoint - the endpoint of the broker (represents the IP address and port of the broker) last_heartbeat - the last time the broker sent a heartbeat to the manager status - stores whether the broker is active or not

The ConsumerMetadata table stores information about the consumers in the system and it contains the following columns: consumer_id - a unique identifier for the consumer topic_name - the topic the consumer is subscribed to partition_id - the partition the consumer is subscribed to (is assigned in a round-robin fashion if the consumer is subscribed to the topic as a whole and not to a specific partition) offset - the offset of the consumer

The ProducerMetadata table stores information about the Producers and it contains the following columns: producer_id - a unique identifier for the producer topic_name - the topic the producer is producing to.

The PartitionMetadata table stores data about the partitioning the topics in system and it contains the following columns: id - a unique identifier for the partition topic_name - the name of



the topic partition_id - the partition ID of the topic (0, 1, 2, ...) broker_id - the broker ID of the broker that is hosting the partition size - the size of the partition

Brokers

From the original design which consisted of four models, the current design choice has reduced the number of models to two, namely:

-Topic_Names contains: topic_name, partition_id, {supports CreateTopic, ListTopic, CheckTopic}

-TopicMessages contains: message_id, topic_name, producer_id, message {supports enqueue, dequeue, size}