CS60002 Distributed Systems

Assignment 2: Distributed Queue with Partitions and Broker Manager

Maitrey Govind Ranade (18CS30026)
Kothapalli Dileep
Arpit Das
Gawai Laukik

Overview

We design all the specified modules in the assignment i.e. :

- Broker
- Broker Manager (read/write)

as Flask applications. Flask framework provides an easy way to build http endpoints, enabling **asynchronous** and **simultaneous** handling of the designed endpoints for various tasks by using **threading**.

For persistent data storage required by any process, it uses postgreSQL, with **write-ahead logs (WAL)** enabled to ensure data protection against crashes

Source Structure

```
./
├─ broker/                    # broker module + heartbeats
│  ├─ broker.py               # server for broker
├─ broker_manager/            # broker manager module
│  ├─ common/
│  │  ├─ debug.py
│  │  ├─ db_model.py          # contains database orm structures
│  │  ├─ manager_routes.py    # implements endpoints
│  │  ├─ manager_sync.py      # heartbeats for heath, ...
│  ├─ manager.py              # broker manager entrypoint
├─ consumer/
│  ├─ consumer.py             # consumer client
├─ producer/
│  ├─ producer.py             # producer client
├─ queueSDK/                  # sdk for making clients
├─ scripts/                   # scripts for docker containers
├─ tests/                     # testing scripts
```

Broker Manager

The broker manager module is a flask application contained in in the file `broker_manager/manager.py`. It can be run in a **read mode** (reads messages from the brokers for consumers) or a **write mode** (writes messages by simply passing an argument. Each broker manager (read/write) maintains the following tables for **metadata** storage:

- **Producer** maps id, topic, (optionally partition) and health
- **Consumer** maps id, topic, (optionally partition) and health
- **Topic:** maps topic id and name
- **Partition:** maps id, topic id, broker id
- **Broker:** contains broker ids and their ips and ports. Also maintains health status
- **Replica**: contains ips and ports and health status of replicas.

Since we use postgreSQL as our choice of storage at the brokers. Making more partitions will not improve message retrieval performance (due to a fixed schema). Our design supports any number of partitions as per assignment. But as a design choice, every topic has a default (implicit) partition on every broker (with id -1). If a producer does not specify a partition ID, its messages go to any random broker in the implicit partition. On the other hand if a consumer does not specify a partition ID, it consumes messages from all possible partition IDs.

We now describe the endpoints supported by broker manager in our design, with the relevant design choices:

Endpoints Open in Both modes:

- **[GET] /topics:** Get a list of created topics.
- **/topics/partitions:** Get a list of partitions of a given topic name (returns broker info of partitions)
- **/brokers:** returns healthy brokers.
- **/replicas:** List all the known healthy replicas of broker manager in the system.

Endpoints Open in Write mode:

- **/brokers/create/partition:** Creates partition for a given topic name and broker ID.
- **/brokers/remove:** Removes a broker with a given ID
- **/brokers/heartbeat:** Receives heartbeat from a broker (with broker IP and port), acts as a broker discovery service for write broker manager.
- **[POST] /topics:** Creates a new topic with a given topic name and returns its ID

- **/partitions:** Creates a partition given a topic name and a broker ID on the broker.
- **/producer/register:** Registers a producer for a given topic name and partition ID. If Partition ID is not given or -1, it registers for the common implicit partition for every topic on every broker.
- **/producer/produce:** Takes a message by producer and requests a broker to store it. If a partition ID is supplied, it will communicate and store to the broker handling particular partition. Otherwise it will choose a broker **uniform at random** from healthy brokers and put the message in its implicit partition for uniform distribution, leading to uniform load distribution by dequeue.
- **/consumer/health_poll:** (Only opened by primary) Endpoint for inter-broker manager communication. The consumers poll the read manager, the read manager redirects the polling to this endpoint for health metadata update for consumers.
- **/metadata/sync:** Read only managers poll this endpoint on primary to sync metadata with it.

Endpoints Open in Read mode:

- **/consumer/register:** Registers a consumer for a given topic name and partition ID. If Partition ID is not given or -1, it registers for all the partitions of the given topic name.
- **/consumer/consume:** Endpoint for consumer to dequeue. If a partition ID is specified, it will consume from the specific partition handled by a particular broker. Otherwise, it will choose a broker **uniformly at random** and dequeue the latest message for the given topic in that particular broker. This is done to maintain uniform load balancing. Updates health info using /consumer/health_poll.

In addition to all these endpoints, the primary broker manager runs a thread to periodically update health status if brokers, producers, consumers and replicas.

Replicas run a thread periodically to sync their metadata with the primary.

Broker

The broker module is a flask application contained in in the folder `broker/broker.py`. It supports the following endpoints

- **/store_message:** Endpoint for receiving a message from the primary (write) broker manager. It receives message and writes it into the database.

- **/retreive_messages:** Retrieve multiple messages with IDs greater than a given offset.
- **/consume:** Endpoint for consumer request as directed by the read manager

Design Choices:

- Since all the metadata is handled by the broker manager, the broker itself just stores the messages as they come, independently, without any foreign key relationship. It does not contain any information regarding the registration of producers or consumers.
- However, since messages are specifically independently ordered w.r.t. each broker (as they are operating parallel and independent nodes), the brokers itself maintain offsets for a specific consumer as directed by the broker manager. This helps avoid unnecessary syncs between different read only broker managers and ensures **consistency** without compromising horizontal scalability.

The brokers periodically send **heartbeats** to write broker manager. They include their IPs and ports in each beat. So the heartbeats of the brokers also double as a **discovery service** for the broker, serving two purposes at once. (health checks and discovery).

The broker process has its own database with WAL enabled containing a **message** table for storing messages and an **offsetcons** table for storing offsets as directed by the manager.

Other Components

Other components such as SDK for queue (in folder `queueSDK/`) and producer and consumer processes are updated from the previous assignment according to new newly designed endpoints.

Testing

- We do our testing using docker containers.
- Each node (broker, write broker manager, read broker manager) is launched inside its own docker container (with their own database).
- All the nodes communicate through the default docker subnet (172.17.0.0/16).
- IP of write manger is fixed to 172.17.0.2, as write manger also acts as a service discovery service for all other nodes spawned later on.
- Producer and consumer clients are launched directly on the host system.
- Node crashes are simulated by using the `docker kill` command.

The `scripts/` directory contains helper scripts for testing and launching containers.

- `create_image.sh`: creates the base docker image for all nodes
- `launch_container.sh <CONTAINER-NAME> <PROCESS-FILE>.py`: Launches a fresh container for a node with flask application `<PROCESS-FILE>.py`.
- `boot_container.sh <CONTAINER-NAME> <PROCESS-FILE>.py`: Starts a killed/stopped container for a node with flask application `<PROCESS-FILE>.py`.

To quickly test everything, just run the created testcase:

```
./tests/test.sh
```