CS60002 Distributed Systems

Assignment 1: Implementing a distribured queue.

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

Design Approach

According to the given specification of the http server endpoints in the assignment, each producer_id or consumer_id has its own corresponding topic assignment. Hence it makes sense that each of the queues holding messages for a specific topic is individually accessible, regardless the state of other topic queues.

Part A

The queing datas-tructures are maintained in memory for the broker, and has the following design choices:

- The design allows asychronous requests through the specified endpoints. Writeable data structures are locked (using python multihreading) by any request for any operation involving a write. This preserves an ordering of messages from the POV of the broker. Also it guranettees the correctness and atomicity of requests such as produce/register, consumer/register, producer/produce.
- **There are no readlocks.** Since the implementation is in python, python's inbuild **GIL** helps protect the integrity of reading its own datastructures. In short, reads are non blocking, whatever is available in memory is returned immediately.
- Each Topic queue has its own lock so multiple producers can simultaneously write different topics.

Tradeoffs:

- The consistency of datastructures provided by python comes at a cost of not being able to multiprocess. Since python does not have IPC routines like C, this is multithreaded but can't be spawned over multiple processes, which is a requirement in production servers.

Part B

To provide failure recovery when the broker goes down, all the endpoints commit to a postgres database and the following endpoints are added/ or modified:

- The produce/produce takes an additional parameter, “prod_name”. This is an identifier assigned by the client, it is helpful in restoring the state in case of a failure.
- The consumer/consume endpoint returns an additional parameter , “offset”, which denotes the number of messages dequeued by a particular “consumer_id”

The database has following tables:

- Message: contains the message, it’s topic and the producer client (“prod_name”)
- Consumer: containing a mapping of consumer_ids, topics_ids and offsets (for dequeuing)
- Topic: contains a mapping of topic_ids and topics_names.
- Producers: a mapping of producer_ids to topics.

Part C

- This part implements an SDK module that Allow functions for producer enqueue to the registered topics.
- For consumer it implements a module that parallel polls all the queues that a particular consumer is registered to. (Consumer.run())
- It also handles functionality of restoring a consistent view in case the the server crashes (covered in discussion)

Testing

- Basic endpoint testing by curl command
- Further, the implementation is tested by producing logs by multiple producers at variable intervals.
- The results are valided by checking the following
    - For a single topic, messages produced in that topic = messages consumed.
    - The order of messages consumed is same across all consumers for a particular topic.
    - See “tests/verify.py” in the source directory that tests this.

- Finally data recovery is sending a SIGINT manually to the broker_part_b.py while testing

Discussion

Recovery from failure is a tricky part in this assignment; random failures can lead to different states of the clients and database. This is particularly troublesome when:

- The broker crashes after committing update to database but before returning the status message. Now client thinks that request failed but infact it is written to database.
- Sometimes due to a crash a success message is returned after calling commit, but due to server crash database rolls back.

We modified the requests slightly by adding more information, so that when crash happens, the clients can update database accordingly to ensure a consistent view.

Hyperparameters

- **Consumer** (in queueSdk, consumer SDK)
  - *timeout* : time by which if no new messages are dequeued then the consumer terminates
  - *poll_time*: time interval in which a particular consumer calls consumer/consume for a particular topic
  - *poll_time_after_error*: if some error occurs while consuming, this is the time consumer waits before going to regular polling
- **Producer**
  - *max_timeout*: maximum tine between two requests of same producer.