Distributed Queue - Part 2

# Design Details

### Overview

The base of our design process was that the broker managers, be it the read or the write manager or the broker all run as different processes on separate, probably distant machines. Thus, each has its own persistent storage and its own copy of in-memory data structures.

Secondly, we decided that the queue order across all messages will be preserved. That is the write manager, maintains an ordering among the messages it receives and stores across various brokers. When a consumer wants to consume messages from a given topic, this ordering is preserved. This adds to the complexity of serving requests but gives a FIFO guarantee.

Moving from the bottom level up. The last level of servers are the brokers. Brokers in our design are dumb processes, not doing any computation but simply maintaining a persistent storage storing topic-wise, partition-wise message queues. Supporting three major operations : connect to the broker manager, add message to given topic, partition and retrieve given message index from topic, partition.

Above this level, we have the broker managers. There is one write manager, and two read-only managers. Although the number of read managers do not really affect the operation of the system. But we do assume only one write manager. As already clarified, the terms “write” and “read-only” refer to the operations on the messages themselves and not on the broker managers themselves. Both write and read managers perform writes on the queue metadata. The read managers only update the consumer read index but in any case that does not change the complexity introduced. However, with this the read managers aren't really different from the write managers. In fact, they both run on the same APIs. With a little clever tweak, we can also make our system fail proof in case of write manager crashes

Since, our implementation does not assume any sort of central database, thus a critical feature or rather necessity was the syncing of these with the other managers. Here, there was a classic distributed trade-off decision: consistency vs. availability. The design we followed puts consistency over availability. So, whenever a manager updates its metadata by performing some operation, that is communicated to the others who then commit to their database as well. We assume no link failures and reliable links and no byzantine behaviour by any of the managers.

The top-most level API is the services endpoint visible to producers and consumers and we dupe it as the load-balancer that receives the requests from producer and consumer clients and redirects them to the appropriate broker manager. The consumer read requests are delegated to the read-only managers while others to the write manager.

### Implementation

1. **Brokers:**

As specified above the brokers in our scenario are dumb servers. They have their own persistent storage and accept requests from broker managers.

The following endpoints are served at the broker:

1. **“/” [Heartbeat–GET] :** This is the endpoint managers call periodically to get the state of the broker. If online, the broker responds with a success code.
2. **“/logs” [Enqueue–POST] :** This endpoint is called to insert a new log message in the queue. It takes as arguments the index, the topic name, the partition id, and the message itself. The index is necessary to be allotted by the manager to ensure queue ordering across the different partitions.
3. **“/logs” [Dequeue–GET] :** This endpoint is used to consume messages. It takes as input the index, topic name and partition id for the message.

**Tables:**

The database stored at the brokers consists of the following table(s):

| **Table#1 : Logs** | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | log\_index | Stores the topic-specific global message index |
| 2 | topic\_name | Stores the topic of the message |
| 3 | partition\_id | Stores the partition where the message is added |
| 4 | log\_message | Stores the message content |

1. **Broker Managers:**

The write and read only managers are only different in theory. Both serve the same endpoints listed below.

1. **“/” [Heartbeat–GET] :** the corresponding heartbeat endpoint for managers
2. **“/topics” [Add Topic–POST] :** the endpoint to add new topic to the queue
3. **“/producers” [Add Producer–POST] :** the endpoint called by producers to add themselves to a given topic; returns producer id and list of all partitions and their brokers thus informing the client if it decides to produce to a specific partition.
4. **“/consumers” [Add Consumer–POST] :** the endpoint called by consumers to register to a given topic; returns a consumer id but no partition based retrieval facility is provided unlike the producer case.
5. **“/messages” [Enqueue–POST] :** calls to add messages to the given topic queue, by registered producers are served at this endpoint; optionally takes a partition id as input to allow entering messages into a specific partition.
6. **“/messages” [Dequeue–GET] :** consumer calls this endpoint to retrieve messages from the given topic queue; messages are returned in a FIFO order with respect to their arrival at the write manager.
7. **“/unread\_messages” [Size–GET] :** the number of unread messages in a given topic queue for a given consumer id.
8. **“/brokers” [Add Broker–POST] :** the endpoint called by brokers to add themselves to the system; the design does not involve managers spawning brokers but brokers coming up on their own and calling the manager endpoint to get added to the system/network.
9. **“/live\_sync” [During Operation Sync–POST] :** the endpoint is invoked with updated data to inform other managers of new writes done; as the name suggests, maintains live synchrony amongst the managers.
10. **“/init\_sync” [Startup Sync–GET] :** the endpoint is called on the write-manager when a read-manager comes up after a failure or for the first time ever, it requests the write-manager to bulk send all the updates that it missed, using which the lagging read-manager updates its own database.

The broker health check(running as a separate background thread) is used to maintain the live status of each broker. In case any request is defined for a particular broker that is not online at the given moment, we do not consider a reassignment model for its member partitions, since in such a design, when the original broker recovers there are two partitions with the same id for the same topic and in such a system maintaining the queue order becomes extremely convoluted task.

So, to keep the design simple, if some producer requests to produce to a certain partition that is on a non responsive broker, the request is denied with an appropriate error message.

**Tables :**

The database at any manager level consists of the following tables:

| **Table#1 : Brokers :** stores the brokers and their respective addresses that have at any point connected to the system; note the assumption that willing brokers will call the add-broker endpoint to get themselves added; managers do not spawn brokers. | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | id | Stores the id of the broker; unique to an ip-port |
| 2 | ip | Stores the IP address of a broker endpoint |
| 3 | port | Stores the Port of the broker endpoint |
| 4 | is\_running | The status of the broker; detected & updated using the heartbeats to the brokers |
| 5 | updated\_at | Stores the timestamp when this entry was updated or inserted |

| **Table#2 : Topic :** stores the topics that have been added to the system; can be added either by the add-topic endpoint or the add-producer endpoint that is called for a non-existent topic. | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | name | The name of the topic; serves as the primary key as well. |
| 2 | updated\_at | Stores the timestamp when this entry was updated or inserted |

| **Table#3 : Consumers :** stores the consumers that have registered and the topic they have registered with; also stores the message index that has been already read to ensure FIFO-queue ordering. | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | consumer\_id | Stores the id of the consumer, upon register request |
| 2 | topic | The topic where the consumer has registered |
| 3 | idx\_read\_upto | The index of the message that the consumer has already read |
| 4 | updated\_at | Stores the timestamp when this entry was updated or inserted |

| **Table#4 : Producers :** stores the producers that have been registered along with the topic they have registered for. | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | producer\_id | Stores the id of the producer, upon register request |
| 2 | topic | The topic where the producer has registered |
| 3 | updated\_at | Stores the timestamp when this entry was updated or inserted |

| **Table#5 : TBPMap [ Topic-Broker-Partition ] :** maps what partition of which topic is stored on which broker; | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | id | Unique id of the entry |
| 2 | topic\_name | The topic whose partition is being stored |
| 3 | partition\_id | The partition id of the topic that is being stored |
| 4 | broker\_id | The broker where the given partition of the given topic is stored |
| 5 | updated\_at | Stores the timestamp when this entry was updated or inserted |

| **Table#6 : TPLMap [ Topic-Producer+Partition-LogIndex ] :** for each topic, maps which partition holds which global message index; this table is most important for preserving the queue ordering. | | |
| --- | --- | --- |
| **#** | **Column Name** | **Description** |
| 1 | topic\_name | The topic whose message index is being mapped |
| 2 | log\_index | The index of the message being mapped |
| 3 | partition\_id | The partition id where the message with the given log index is stored |
| 4 | producer\_id | The producer that produced this message |
| 5 | updated\_at | Stores the timestamp when this entry was updated or inserted |

**Code Structure:**

To ensure uniformity and modularity, the entire broker manager code is divided into multiple directories and class files, briefly explained below:

1. **app.py:** launcher for the Flask app
2. **/db\_models:** contains the SQLAlchemy enabled ORM classes for each of the above tables.
3. **/src:** holds almost the entire programming logic
   1. **views.py:** defines the endpoints served in app.py
   2. **\_\_init\_\_.py:** defines the app hyperparameters and contains the calls to be run on app startup; also spawns the health\_checker thread called heath\_checker\_daemon that makes regular health check calls to the brokers
   3. **/utils:** contains the utility functions and helper methods
      1. **prounter.py**: essentially a locked counter variable
      2. **consumer\_metadata.py, producer\_metadata.py:** defines a data structure that holds topic-specific consumer and producer information in memory.
      3. **log\_metadata\_queue.py:** a data structure that holds the topic-specific information of log\_index and partitions in a list.
      4. **partition\_to\_broker\_dict.py:** as name suggests, maintains the mapping from partition-ids to broker-ids.
      5. **topic\_to\_location\_dict.py:** maintains the mapping from topic name to PartitonToBroker dict defined above.
      6. **sync\_db.py:** utility functions that support syncing with the other managers
   4. **/models:** contains the following classes forming the main backend of the manager
      1. **master\_queue.py**: the Master\_Queue class holds all the metadata and the logic; and is at the highest level. The endpoint receiving functions call the Master\_Queue methods to get responses. Maintains a Master\_Broker object, and a dictionary of Topic objects indexed by topic name
      2. **master\_broker.py:** a sub-class held inside the Master\_Queue class, handles all operations relating to the metadata of the brokers and its management.
      3. **broker.py**: the Broker class is the sub-class of the Master\_Broker class, which maintains a Broker class object for each broker.
      4. **topic.py**: the Topic class contains all information pertaining to a given topic, including the consumer metadata, producer metadata and the log metadata.

1. **Load Balancer:**

This is a very thin, lightweight app added to enable exposing a single unified API to the clients. The requests received are forwarded to the write manager or read manager (for dequeue). For this purpose, a utility URLBook class maintains the addresses of the managers and also performs timely health checks to the managers.

If a given request is for a read\_manager then, any one of the *live* read\_managers is randomly picked to serve the request.

Otherwise it is redirected to the write manager.

In case the write manager is down, currently there is no backup write manager maintained. But any of the read-managers can also serve the same requests at any given moment that the write manager can. For the sake of this assignment, if the write manager crashes, the system is down and all requests are sent appropriate error messages.

1. **WAL:**

The PostgreSQL database management service used for the persistent storage management is already implemented with an in-built write-ahead-logging mechanism that is automatically enabled.

[ Refer: <https://www.postgresql.org/docs/current/wal-internals.html> ]

We implement a simple DB based exterior WAL in a separate version but do not include it in the main branch considering the overhead of doing two separate WA logging and recoveries. Besides, a WAL on WAL system is plain redundant.

# Testing

1. Currently, the preliminary testing involves spawning multiple brokers, 3 broker\_managers and the load\_balancer on known ports on the same host machine. Then using the Postman API Tester to send multiple calls to the load\_balancer.

# Members

| **Sr No** | **Name** | **Roll Number** |
| --- | --- | --- |
| 1. | Aaditya Agarwal | 19CS10003 |
| 2. | Debanjan Saha | 19CS30014 |
| 3. | Pritkumar Godhani | 19CS10048 |
| 4. | Akarsh Singh | 19EE10003 |
| 5. | Parth Tusham | 19CS30034 |
| 6. | Sayantan Saha | 19CS30041 |