PA3: Distributed Queue with Consensus

Distributed Systems CS60002

Design

Before beginning the part with consensus and replication, we will describe the changes made in the base implementation of Programming Assignment-2 (PA2). Firstly, we drop the global, topic-specific FIFO order preservation guarantee that was central to the development and design in PA2. Second, now producers and consumers are concerned with specific partitions in specific topics. These partitions are replicated 3 times across the brokers and this leads to the need for consensus amongst these partitions.

The model is similar to the PA2 design with three levels of servers – brokers, managers, load-balancer.

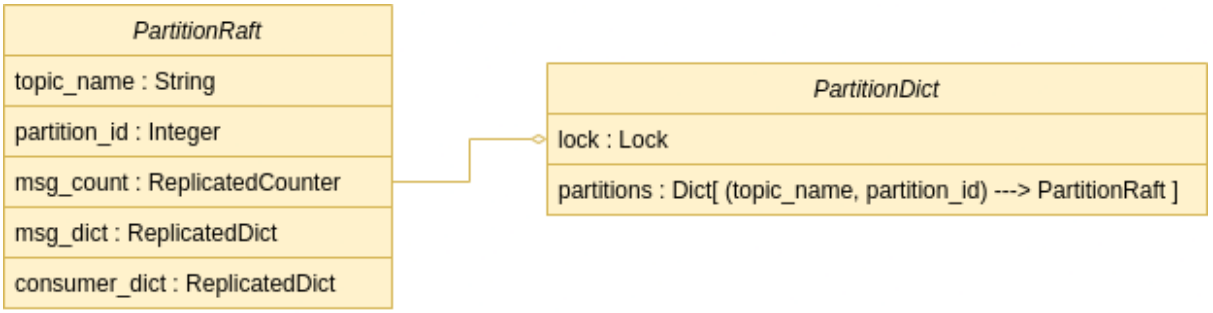
The most basic layer of servers are the brokers. Unlike the PA2, brokers are not dumb servers in this scheme. They maintain active in-memory data structures and have persistent storage. The database for brokers maintains two tables, Log and PartitionRaft. The Log table stores the main message store while the PartitionRaft stores the details of RAFT objects of individual partitions in this broker. These details include partner nodes information to reconnect the RAFT after a shutdown.

Table#1 : Log		
#	**Column Name**	**Description**
1	msg_offset	Stores the index of the message in this partition
2	topic_name	Stores the topic of the message
3	partition_id	Stores the partition where the message is added
4	msg	Stores the message content
5	msg_replicated_bitmask	Stores the status of DB updates across replicated partition

Table#2 : PartitionRaft		
#	**Column Name**	**Description**
1	topic_name	Stores the topic whose partition is represented

2	partition_id	Stores the partition represented
3	msg_count	Stores the number of messages in this partition
4	replica_id	Stores the replica id (0,1 or 2 in this case)
5	raft_host	Stores the RAFT node address of this partition
6	raft_partners	Stores the address of partner RAFT nodes

Each partition is represented by an object of the PartitionRaft class. This utilises the SyncObj class of the pysyncobj library, a popular open-source RAFT library in Python. The PartitionRaft class has three synced members (synced using pysyncobj 's SyncObj) – a message counter *msg_count*, a message dictionary *msg_dict* and a consumer dictionary *consumer_dict.* The consumer_dict maps the consumer_id's to the offset of the messages read up to now in the current partition.



The msg_dict is an interesting data structure. It is a map of the index of the message and the message content. Index is obtained by the msg_count counter. The number of messages may be typically very high. Maintaining all those messages in-memory is not desirable or necessary. To keep the msg_dict from growing, we flush messages that are pushed in the database by all the replicas from the msg_dict. To do so, a bitmap of 3 set bits is maintained; any partition that unsets the last set bit will remove it from msg_dict. This ensures that the databases are consistent across replicated partitions. And the msg_dict serves as a caching data structure for messages.

The broker's in-memory data structure includes a PartitionDict dictionary that maps individual partitions (topic name and partition number) to the corresponding PartitionRaft objects.

Broker Endpoints				
#	**Endpoint**	**URL**	**Method**	**Description**
1	Heartbeat	/	GET	Returns 200 if alive
2	Raft Status	/raft_status	GET	Returns the SyncObj status

3	Message	/logs	GET	Retrieve message
			POST	Add message
4	Partition	/partitions	GET	Returns all partitions on this broker
			POST	Add a new partition
5	Consumer	/consumers	POST	Register a new consumer to partition

The next level is the level of the managers. The managers maintain the producer, consumer and topic metadata and the registry of which producer/consumer are registered to what topics.
The managers also maintain synced data structures for that purpose – producer_dict, consumer_dict, topic_dict, broker_dict. Each of these are self explanatory and are kept in RAFT consensus with other managers using pysyncobj 's SyncObj class.
Besides the typical push message, pop message, add topic, register endpoints, the manager maintains the endpoints to spawn new brokers and partitions. Producers can thus create new partitions before posting messages. Managers also support endpoints to get brokers for given topics and partitions.

Manager Endpoints				
#	**Endpoint**	**URL**	**Method**	**Description**
1	Heartbeat	/	GET	Returns 200 if alive
2	Raft Status	/raft_status	GET	Returns the SyncObj status
3	Message	/logs	GET	Retrieve message
			POST	Add message
4	Partition	/partitions	GET	Returns all brokers for this partition
			POST	Add a new partition
5	Consumer	/consumers	POST	Register a new consumer
6	Broker	/brokers	GET	Get a list of all brokers
			POST	Create a new broker
7	Topic	/topics	POST	Add a new topic
8	Producer	/producers	POST	Register a new producer

The final level, the load-balancer, exposes the final endpoints to the clients. It takes regular heartbeats from the managers and maintains the current RAFT leader in the managers

RAFT network. All requests are then forwarded to the RAFT leader. The leader check is periodically called to keep the leader running at all times.

It is imperative to note that for RAFT the quorum requires at least 2f+1 votes to be able to tolerate f faulty ones. If at any time, in any of the RAFT networks, either the managers or the broker-level partitions, if sufficient number of nodes are not alive so as to maintain the quorum, the updates to in-memory structures will be blocked.

Code Structure

The repository is subdivided into logically separate components. A brief description is given below.

1. README.md : environment setup instructions
2. requirements.txt : dependencies for the project
3. install_db.sh : install and configure Postgres Database [for Debian-based distros only]
4. broker/
    a. README.md : contains the information to run the brokers
    b. run.py : creates a Flask app, initialises the database, populates in-memory structures and launches the app on the given host and port.
    c. setup_db.sh : contains the database setup commands
    d. db_models/ : contains the database models
    e. src/
        i. views.py : contains the endpoint and their definition
        ii. app.py : contains the code to initialise the app and the database instances
        iii. raft.py : defines the classes and methods used by Raft synchronisation
        iv. msg_handler.py : handles the periodic database dumping from in-memory data structures
        v. utils.py : defines a few helper functions
5. manager/
    a. README.md : contains information to run managers
    b. run.py : creates a Flask app, populates in-memory structures and launches the app on the given host and port.
    c. src/
        i. views.py : contains the endpoint and their definition
        ii. app.py : contains the code to initialise the app and the database instances
        iii. raft.py : defines the classes and methods used by Raft synchronisation
6. load_balancer/
    a. README.md : contains information on launching the load balancer
    b. load_balancer.py : the app definitions and code for the load balancer

7. unit_tests/
    a. README.md : contains information to start a manager cluster
    b. run_manager_cluster.py : launches 3 managers on different ports
    c. broker/
        i. README.md : contains the detailed commands to run unit tests
        ii. test_*.py : each of the 6 files tests a specific broker functional endpoint
    d. manager/
        i. test_manager.py : tests the manager functional endpoints
    e. load_balancer/
        i. test_load_balancer.py : launches a manager and brokers to test the load balancer
8. stress_tests/
    a. README.md : detailed instruction on performing stress tests
    b. stress_threaded.py : spawns multiple threads and performs add and get messages functions
    c. stress.py : non-threaded version for the above tests

Testing

Unit Tests

All endpoints are tested using individual unit test files.
The folder unit_tests/ contains level-wise unit tests. Refer to individual README files on how to run the unit tests.

Stress Tests

Bulk operations are performed from multiple threads on the complete system running, to test concurrency, consistency and availability.

Stop-and-Go Tests

Random nodes are stopped (Keyboard Interrupt) and re-spawned to test recovery and fault tolerance.
Once again, once that many nodes are stopped as can be handled by the Raft, i.e., quorum should be possible at all times.

Members

Sr No	**Name**	**Roll Number**
1.	Debanjan Saha	19CS30014
2.	Aaditya Agrawal	19CS10003
3.	Pritkumar Godhani	19CS10048

4.	Parth Tusham	19CS30034
5.	Akarsh Singh	19EE10003
6.	Sayantan Saha	19CS30041