**CS60002 Distributed Systems**

Assignment 3: Consensus Module using Raft

Date : April 9, 2023

*Contributed by:*

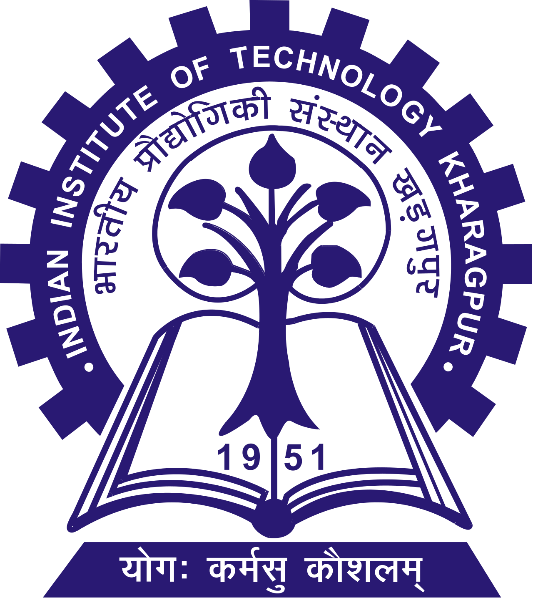
Amartya Mandal 19CS10009

Rupinder Goyal 19CS10050

Esha Manideep Dinne 19CS10030

Anish Sofat 19CS10011

Divyansh Bhatia 19CS10027



Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Spring Semester, 2022-23

# Assignment 2 Design Decisions

## Assignment 1 Modifications

In Assignment 1, we worked with a single broker. There was a single database that contained all data that include information about all producers, consumers and topics, as well as all messages in the system.

A modified version of this same system is used to implement each broker for this assignment. Here, instead of grouping messages by topic as earlier, we are grouping messages according to partitions. Each broker has a database that contains all partitions assigned to it (a broker might have multiple partitions of the same topic). The broker manager redirects requests to a broker as required, and the broker perform all required functionalities that include enqueue, dequeue, registering consumers, getting size and probing.

Probing is a new functionality implemented to check the first message in the queue without marking it as viewed. This functionality is required to compare the time stamps of all messages that are at the front of the queue for all brokers to obtain the actual first message.

## Design Decisions

### Partitions

* + Each topic is broken down into partitions horizontally.
  + Each partition of a topic might be on different brokers.
  + Multiple partitions of the same topic might pe on the same broker.
  + Creating a new topic automatically creates one partition for the same.
  + Every new partition created is assigned to the broker that handles the least number of partitions.
  + Any log message produced by the producer is assigned to a partition in a round robin fashion, unless a specific partition number is provided by the producer.

### Broker Manager

* + The Broker manager handles the metadata of producers, consumers, topics, brokers and partitions. It also stores the mapping between consumers and topics & partitions.
  + Producers can choose a specific partition to produce, in absence of which a round robin algorithm is used to choose partition.
  + The broker manager can add or remove brokers. Each broker is run using a docker, and the broker manager spawns a new broker in the docker port provided.
  + Current algorithm assigns every new partition to the broker with least partitions for uniform load.
  + Also, we send a request to launch a new broker whenever the number of partitions per broker passes a limit for load balancing.
  + Services like List Topics, Register Producer are handled by the broker manager; the rest including Create Topic, Create Partition, Register Consumer, Enqueue, Dequeue, Size are redirected to the respective broker (might be multiple brokers).
  + Health check is implemented using separate function in all brokers, that is called periodically on a separate thread to obtain a heartbeat from all brokers.
  + 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](https://www.postgresql.org/docs/current/wal-internals.html%20) ]

* + Read replicas (Load Balancing)

### Broker

We carry forward the design decisions we had implemented in Assignment 1 (Part B i.e., after implementing the persistent storage for handling server crashes), while implementing the brokers.

* All the log messages will be available to the new users as well(give some reasonable explanation).
* Database is implemented using PostGRESQL.
* Django handles all the low level detail communication with the Database.
* There are multiple models(tables), they are Topic, Producer, Consumer,
* LogMessage mainly.
* There are two other linking tables which link consumers with their
* subscriptions (Consumer Subscriptions) and another table which links log
* messages to the consumers who viewed the messages (consumerViews).
* There are various fields like TextField, DateTimeField etc.
* Consumer and LogMessages are linked using a ManyToManyField .
* LogMessages are sorted based on their creation time order(by using the
* ordering = [‘created’] in the Meta Class of LogMessage).
* All the functions implemented in Part A have been implemented again in the
* file queue\_funcs.py using the databases.

## Model Designs

### Broker Manager

The following database models are implemented to store the metadata for the broker managers.

1. Persistent Storage
   1. Producer Model

Stores the producer id (int) and topic (string) subscribed

* 1. Consumer Model

Stores the consumer id (int) and topic (string) subscribed

* 1. Topic Model

Stores all topics (string) present in the system

* 1. Broker Model

Stores the port (int) and database name (string) for each broker (Each broker has its own database that is assigned some unique name)

* 1. Partition Model

Stores the id (int), topic (string) of the partition, partition number (int) and the broker it is assigned to.

1. In Memory Storage
   1. Lock

To implement a threading to prevent deadlocks and handle multiple asynchronous accesses.

* 1. Topic

In memory storage for faster access. Contains details of all topics that include the name, produces and consumers subscribed, partitions and an index for round robin allocation to partitions.

* 1. Metadata

Maps all topic names to the topic

* 1. Broker

Maps every broker to the number of partitions it contains. Used for load balance.

* 1. Ids

Store three integer ids for brokers, producers and consumers to assign ids to them sequentially.

### Broker

The following database models are implemented to store the metadata for the broker managers.

1. Persistent Storage
   1. Log Messages

Stores all the log messages for each partition in the broker. Contains the message (string), creation time (DateTimeField) and partition (foreign key)

* 1. Consumers

Stores the consumer ids (int) of the consumers, and two many to many fields for mapping to their subscriptions and the messages viewed. The consumer model required for this assignment and the previous assignment is slightly different because, in this assignment, a specific consumer can be subscribed to multiple partitions in the same broker(which was not allowed as new consumer\_ids would be given in assignment 1)

c. Producers

Redundant model as the check whether a producer is registered to the topic or not is made at broker manager itself.

d. Topic

This model is basically modified to store the partitions present in the broker rather than just the topics. In the broker we modeled each partition of assignment 2 as a separate topic in terms of assignment 1. Instead of topic\_name being the only key, we made both topic\_name and partition\_no be the keys of this model

# Assignment 3 Modifications

## Modifications in broker manager

Any POST request received by the broker manager is no longer forwarded to the respective broker only. Instead, it is now forwarded to the RAFT process. The RAFT process lets all the RAFT objects know of the change.

Another major change in this assignment is using multiple replicas of each partition. We have created 3 replicas (this can be changed) for each partition that are assigned to distinct brokers. These are all synchronised.

* + Create Partition :

Creating a partition now creates 3 replicas of the partition, that are assigned to 3 distinct brokers. The 3 brokers with the least number of partitions assigned to it are chosen for the same. A message containing the topic name, partition id, the port assigned and the ports of the 2 peers, is encoded into a bit stream and sent to the raft process of the broker through a tcp connection. It then waits for an acknowledgement from each broker, on receiving which updates the database.

* + Register Consumer :

While registering a consumer, for every partition of the topic, we choose a leader, and send the request to it. A message containing the topic name, the partition number and the consumer id is sent to it after encoding into a bit stream, via tcp connection.

* + Enqueue :

Similar to the last function, a leader is chosen and a message containing the topic name, the message and the partition number of the partition to which it is to be added (this is decided via round robin scheduling if not provide by the user) is encoded and sent via tcp.

* + Dequeue :

Again similarly, a leader is chosen and a message is sent containing the topic name, the consumer id, the partition id (the partition having the oldest available message for this consumer is chosen using probe) is encoded and sent via tcp. The process then waits for a response from the same connection, which contains the dequeued log message, upon receiving which it returns the message.

## Modifications in broker

The major change in this assignment, is that instead of the broker manager calling the urls of each broker, it is called by the raft objects. Also a new urls and functionality is added to the broker :

* + GetData :

Whenever a raft process is initiated, this function is called. This returns all information about the port number of the raft objects and their peers corresponding to what is present in the persistent storage of the broker (postgres).

## Raft Processes

This class represents a process in the Raft consensus algorithm that can communicate with other processes in the same network. It is responsible for receiving and processing messages from other processes, and for handling requests from the broker such as enqueuing messages, dequeuing messages, adding consumers, and adding topics.

Functions:

* + *\_\_init\_\_(self, id, port, broker)*: Initializes the object with the given id, port, and broker values. The id value uniquely identifies the process in the network, the port value is the port number on which the process listens for incoming messages, and the broker value is the port number of the broker process that is responsible for managing topics and partitions. It checks for any data in the brokers persistent storage and updates itself accordingly.
  + *check\_broker(self)*: A thread function that continuously checks if the broker process is alive by sending a request to the broker at regular intervals. If the broker does not respond within a certain time limit, the function prints an error message and terminates the process.
  + *enqueue(self, key, message)*: Handles an enqueue request from the broker. The key value is a string that uniquely identifies a topic-partition pair, and the message value is a dictionary that contains the message to be enqueued. The function forwards the message to required RAFT Object.
  + *dequeue(self, key, message)*: Handles a dequeue request from the broker. The key value is a string that uniquely identifies a topic-partition pair, and the message value is a dictionary that contains the consumer ID and other information about the message to be dequeued. The function sends the message to the required RAFT Object and returns the dequeued message.
  + *add\_consumer(self, key, message)*: Handles an add-consumer request from the broker. The key value is a string that uniquely identifies a topic-partition pair, and the message value is a dictionary that contains the information about the new consumer to be added. The function forwards the message to required RAFT Object.
  + *add\_topic(self, key, message)*: Handles an add-topic request from the broker. The key value is a string that uniquely identifies a topic-partition pair, and the message value is a dictionary that contains the information about the new topic to be added. The function adds the topic to the dictionary of RAFT Objects.
  + *recv\_messages(self)*: A thread function that continuously listens for incoming messages on the port number specified during object initialization. When a message is received, it is decoded and the appropriate action is taken based on the message type.
  + *add\_topic(self, key, msg\_dict)*: A helper function that adds a new topic-partition pair to the dictionary containing the raftObjects. The key value is a string that uniquely identifies the topic-partition pair. Sends a request to the RAFT Object to add the partition to each broker required.

## Raft Objects

The class raftObj that extends the SyncObj class from the pysyncobj module. This class contains four methods that can be called by clients to perform certain tasks related to a message broker. The methods are:

* + \_\_init\_\_(self, selfNodeAddr, otherNodeAddrs, brokerPort): A constructor method that initializes a raftObj instance. It takes in three parameters: selfNodeAddr, which is the IP address of the current node; otherNodeAddrs, which is a list of IP addresses of the other nodes in the cluster; and brokerPort, which is the port number of the message broker.
  + enqueue(self,message): This method is used to add a message to a specific partition of a particular topic. It takes in a dictionary object named message that contains the following keys: topic, partition\_id, and message. The topic key holds the name of the topic, the partition\_id key holds the ID of the partition to which the message should be added, and the message key holds the actual message to be added to the partition.
  + dequeue(self, message): This method is used to consume a message from a specific partition of a particular topic. It takes in a dictionary object named message that contains the following keys: topic, partition\_id, and consumer\_id. The topic key holds the name of the topic, the partition\_id key holds the ID of the partition from which the message should be consumed, and the consumer\_id key holds the ID of the consumer that is consuming the message.
  + add\_consumer(self, message): This method is used to add a consumer to a particular topic. It takes in a dictionary object named message that contains the following keys: topic and consumer\_id. The topic key holds the name of the topic to which the consumer should be added, and the consumer\_id key holds the ID of the consumer.
  + add\_topic(self, message): This method is used to add a new topic to the message broker. It takes in a dictionary object named message that contains the following keys: topic and partition\_id. The topic key holds the name of the topic to be added, and the partition\_id key holds the number of partitions to create for the topic.

The *@replicated\_sync* decorator is used in the *pysyncobj* library to mark methods that should be replicated across all the nodes in a cluster using the Raft consensus algorithm. When a method is decorated with @replicated\_sync, the library ensures that the method is executed on all the nodes in the same order, ensuring that the state of the object is consistent across the cluster.

When a client calls a method on the leader marked with @replicated\_sync, the node broadcasts a message to all the followers in the cluster. The other nodes apply the same method to their own state, and respond to the leader node indicating that the method has been applied successfully. Once the leader receives a quorum of responses, it considers the method applied, and returns the result to the client.

This ensures that the state of the object is consistent across all the nodes in the cluster, even in the case of failures or network partitions. When a node re-joins the cluster after a network partition, it synchronizes its state with the other nodes using the Raft consensus algorithm.

## Part A

The ATM system is also implemented similarly, using the pysyncobj class for RAFT concensus. It is a simple implementation of an ATM (Automated Teller Machine) using the pysyncobj library. The SyncObj class is inherited to make the ATM object synchronized among multiple nodes.

The replicated\_sync decorator is used to mark the methods that need to be synchronized across all nodes in the cluster. When a method is marked with this decorator, it will automatically be synchronized across all nodes in the cluster, ensuring that all nodes have the same state.

The ATM class defines several methods that correspond to typical ATM functions: add (adds a new account), withdraw (withdraws money from an account), deposit (deposits money into an account), balance (checks the balance of an account), and transfer (transfers money from one account to another).

The while loop at the end of the code listens for user input and calls the appropriate ATM method based on the user's choice. The output of the method call is then printed to the console.

# Testing

## Setting up Dependencies

Run the following commands in the terminal.

- To setup and activate the virtual environment:

*assign2-venv\Scripts\activate*

- Install dependencies:

*pip install -r requirements.txt*

## Running the Project

- Run the app

*python loadbalancer.py*

- Run the write manager

*flask run -p 5001*

- Run read managers

*flask run -p %port*

\* Replace %port with port number for the manager replica

Setting up dockers for multiple brokers

- Setup docker

*python create\_broker.py %n*

\* Replace %n with the id number for the broker

For linux based systems, this will spawn the raft process, but Windows does not support fork. Hence, we need to run the raft processes separately as:

*python raftproc.py %raft\_port %broker\_port*

Replace %raft\_port and %broker\_port by the port of the raft process and the port of the corresponding broker

## Running tests

- For testing all functionalities

*python -i tester.py*

All functionalities are now available to be used by the user very conviniently

* *createTopic(topic\_name) :*

Create a new Topic

* *listTopics() :*

List all topics present

* *registerProducer(topic\_name) :*

Register a producer to this topic

* *registerConsumer(topic\_name) :*

Register a consumer to this topic

* *enqueue(topic\_name, producer\_id, message, partition\_no : default) :*

Add a log message

* *dequeue(topic\_name, consumer\_id) :*

View a log message

* *size(topic\_name, consumer\_id, partition\_no : default) :*

Get the number of messages not yet viewed by the consumer

* *getPartitions(topic\_name) :*

Get the number of partitions for this topic

* *partition(topic\_name, producer\_id) :*

Create a new partition for this topic

* *add\_broker(port) :*

Add a broker using this port