CS60002 Distributed Systems

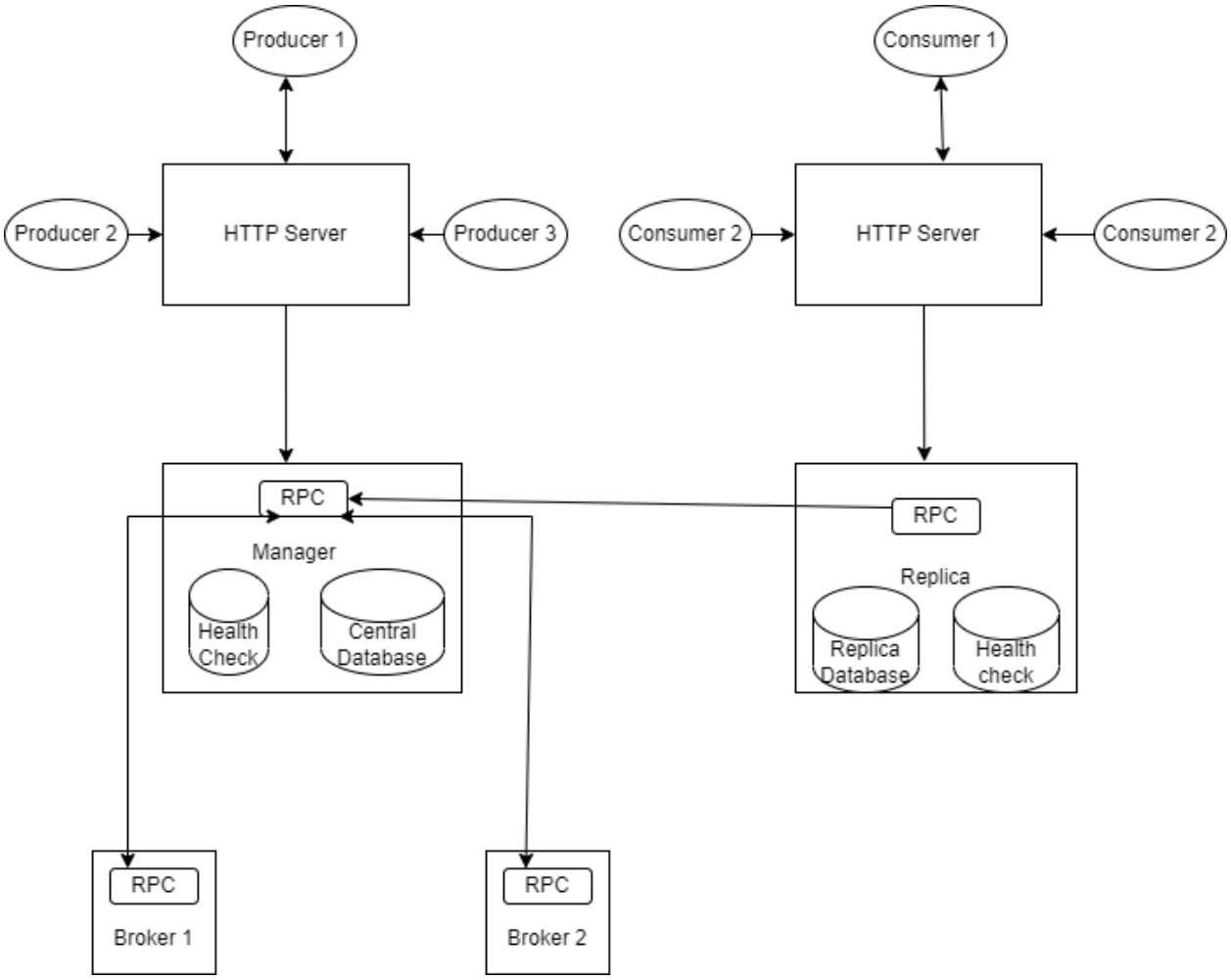
Assignment 3

Consensus Module using Raft

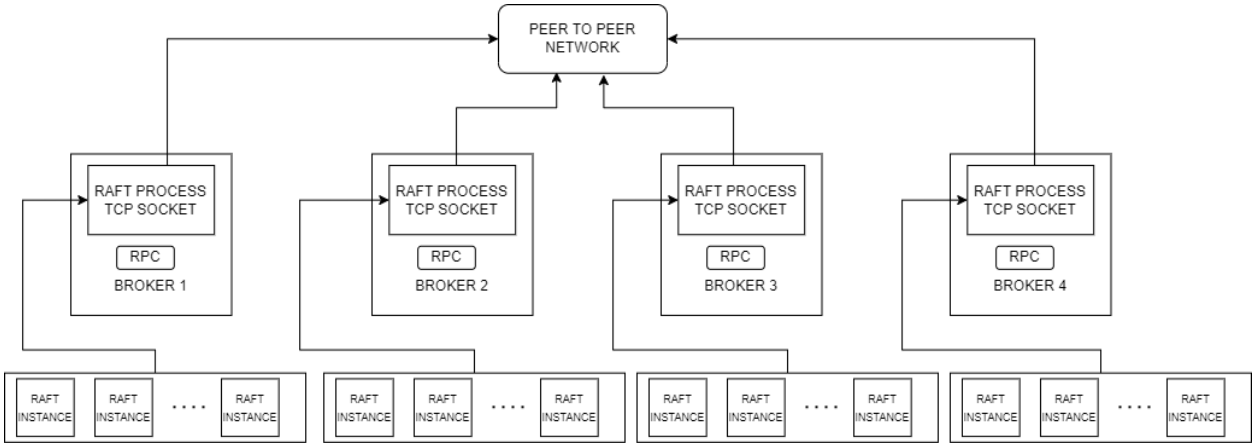
Subham Jain	(22CS60R16)
Aditya Chillara	(22CS60R18)
Venkat Sai Konduru	(22CS60R50)
V.V.S. Sri Harsha	(22CS60R77)
Tudi Jayadeep Reddy	(22CS60R83)

Contents

Architecture:



Broker Network:



Design Choices

How frequently put the data in the SQL DB?

The changes that happen at broker level are communicated to the Manager after a certain thresold so that the manager and update the DB.

How to implement the fetch message since different consumers might have to consume from different positions?

The in-memory **topics** data structures(as described below) has a **bias** key which keeps track of how many messages were deleted/completely consumed. Along with that each consumer keeps track of the position from which they need to consume messages. **bias** and **position** are used together to keep the message sequence for each consumer in order.

Do we have to use a queue?

One choice was to use the actual queue data structure to implement the Message queue, but having the problem of different consumers consuming from different positions made us go with dictionaries/ lists like data structure.

How to deal with the publish_message request if the topic has no subscribers?

One choice that we had to make was either keep the message in-memory for an indefinite period so that every new subscriber can start consuming from the start or to delete the message when all the consumers (who subscribed to the queue before the message got published) consume the message. As we went with the later approach we had to make out data structure as such that the producer cannot publish to a topic to which no consumers are subscribed to.

Optimizations done:

1. Made all the functions in replica asynchronous.
2. An asynchronous task runs in the background which gets updates from the manager on demand.
3. GRPC framework is being used for all communications, which leverages HTTP 2.0.
4. Upgraded from SimpleHTTPServer to Flask framework.

Why PySyncObj **for Raft?**

We have used the PySyncObj module for implementing the Raft consensus algorithm. PySyncObj is a python library for building fault-tolerant distributed systems. It provides the ability to replicate your application data between multiple servers. It uses raft protocol for leader election and log replication.

Replication Factor:
We have used a replication factor of 3. Data in each partition is replicated in two other brokers. This is done to avoid loss of data and any other inconsistencies if a broker is down.

How data is replicated?
Manager allocates the partitions and its replicas of a topic to the brokers in a round robin fashion as the topics are created by the producers. c

The Message Queue

The message queue can be divided into three components

1) Manager
2) Broker
3) Manager Replica

Manager

The manager in the system consists of 3 components

- Health Check DB
- Central DB
- gRPC server

Health Check DB

The DB stores relevant information about when a producer last requested something to the Manager through HTTP calls, and when a broker was called by the Manager which resulted in a successful access of broker.

Central DB

The Central DB stores the Producer, Topics and Messages pushed to the system.

gRPC server

The gRPC server exposes function which are called by the HTTP Server or by the broker.
List of function exposed by the gRPC server:-

- **RegisterBroker**
  Called by the Broker to register itself in the system

- **HealthCheck**
  Called by the broker/replica to check the status of Manager.

- **RegisterReplica**
  Called by the replica to register itself in the system

- **GetUpdates**
  Called by the Replica to get updates from the Manager about changes in the CDB.
- **SendTransaction**
  Called by the HTTP Server whenever it receives a request from the Producer.

- **ReceiveUpdatesFromBroker**
  This is called by the Manager code which in turn called “SendTransaction” function exposed by the broker to receive updates/changes done at the broker.

Broker

Broker in the system deal with Enqueue tasks which happen “in-memory”, there is no DB at broker level as broker data doesn’t have to be persistent.

The broker run a gRPC server which exposes multiple functions that can be called by Manager to direct the request from Producer to Broker.

The broker on performing an operation logs a query locally which then can be called by the Manager on receiving updates from the broker to make the CDB stable.

The broker publishes the following functions:-

- ResetBroker
- GetUpdates
- SendTransaction

ResetBroker

This is called by the broker itself when it gets registered as a broker in the Manager. The function sets the broker_id and clears any data available at the broker side.

GetUpdates

This is called by the Manager after every 5 seconds to receive all the queries executed at the broker level to be pushed in the Central Database available at the Manager.

SendTransaction

This is the primary function which is called by the Manager whenever producer requests something to the Manager.

Followings requests come to the Manager from producers:-

- List Topics
- List Partition
- Create Topic
- Register Producer
- Enqueue Message

List Topics

Manager does the necessary processing.

List Partition

Manager does the necessary processing.

Create Topic

Manager adds the topic to the Central DB if topic doesn't exists otherwise throws an error.
If the topic is added then all the brokers connected to the Manager are informed about the topic creation so that they can update there local data. This is done by calling the "SendTransaction" from manager.

Register Prodcuer

Manager creates a Producer and sends the producer id back to the producer. While creation, if any error occurs that is conveyed to the calling entity.

During registration of Producer, if producer has asked to subscribe to a topic which doesn't exists then the topic is created first and then the Producer is created. Both these information is shared with the all the brokers connected through the "SendTransaction" function exposed by the broker.

Enqueue Message

Upon receiving a Enqueue request from Producer the manager picks a Broker in a round robin fashion and directs that request to that broker. The broker in turn adds the message to the Topic.

Manger Replica

Manager replica handles all the request coming from the consumers.

The manager replica has it's own DB replica which is updated from Manager when the need comes i.e. a consumer needs to consume and there are no messages, or after a threshold.

HTTP Server API

The HTTP server API is written in python. A simple implementation of request handler is made using python built in module Flask.

Server address: http://127.0.0.1:8002
Following are the API Endpoints made available:

- /topics
    - GET request
    - Returns a list of topics available
- /consumer/consume
    - GET request
    - Params = consumer_id, topic (or) Params = consumer_id, topic, partition id
    - Returns a message after dequeuing from the Message Queue
- /size
    - GET request
    - Params = consumer_id, topic
    - Returns size of the queue of given topic in Message Queue
- /topics
    - POST request
    - Params = topic_name
    - Creates a new topic of given name
- /consumer/register
    - POST request

    - Params = topic
    - Registers consumer and returns corresponding consumer_id
- /producer/register
    - POST request
    - Params = topic
    - Registers producer and returns corresponding consumer_id
    - Creates the topic if it doesn't already exist
- /producer/produce
    - POST request
    - Params = topic, producer_id, message or Params = topic, producer_id, partition id
    - Adds the given message to the Message Queue

Persistent Storage Layer

The message queue is lost if the server crashes or restarts as the messages are stored in the memory. Hence, we have added a persistent storage layer to recover from such issues. There are three tasks:

1. Determine schema
2. Store and retrieve log messages
3. Tests to ensure that the functionality is as expected and the message queue is able to recover from failure.

Why Postgres?

PostgreSQL is an enterprise-class open source database management system. It supports both SQL and JSON for relational and non-relational queries for extensibility and SQL compliance. PostgreSQL supports advanced data types and performance optimization features, which are only available in expensive commercial databases, like Oracle and SQL Server. The features supported by postgres are user-defined datatypes, table inheritance, foreign key referential integrity and so on.

Determining Schema

The schema used:

1. topic : The table topic contains topic_name as primary key and bias.
    - Bias is a value that is incremented if a message in the message queue for a particular topic is consumed by all the subscribers.
2. producer : The table producer contains pid, topic_name
    - producer id - pid

- topic_name is the topic the producer registered to and is a foreign key referred to in the topic table.
- partition_id is the partitions the producer is in

3. consumer : The table consumer contains cid, topic_name and position.
    - consumer id - cid
    - topic_name of the topic consumer registered to and is a foreign key referred to in the topic table.
    - position is the message position of the consumer in the message queue.
    - partition_id is the partitions the consumer is in
4. message: The table message contains message, topic_name and subscribers
    - message is the message of the topic.
    - topic_name is the topic name of the message and is a foreign key referred to in the topic table.
    - subscribers are the consumers that are subscribed to the topic name which has the message.
    - partition_id contains the messages in each partition.

Producer Library

Producer Library written in python language consists of Producer class with 4 different functions:

RegisterProducer

The objective of this function is to register a producer for a particular topic name .This function takes the Topic name as argument and registers the Producer to that topic. On success it returns a Producer ID indicating that Producer is successfully registered to that topic.
On failure it raises myProducerError with an error message representing the error**.**

ListTopics

The Objective of this function to display the list of topics present in the distributed queue.The function takes no arguments. On success it returns the list of all the topics available that the producers have created.
On failure it raises myProducerError with an error message representing the error.

Enqueue

The Objective of this function is to enqueue a log message to a Particular Topic with Producer id.The function takes 3 arguments namely Producer ID,Topic name and Log message. On success it returns 0 indicating that the log message is enqueued into the queue.
On failure it raises myProducerError with an error message representing the error.

CreateTopic

This function takes Topic name as argument to create a topic name for the producer.On success it returns 0 indicating that the Topic has been created by the producer.
On failure it raises myProducerError with an error message representing the error.

Consumer Library

The Consumer is written in Python. It has the following functions:

RegisterConsumer

The function takes the topic name as argument and registers the consumer to the topic.
Inorder to subscribe/Register to multiple topics call this function multiple times with different topics.
On failure it raises a myConsumerError with the error message.

ListTopics

The function takes no arguments. On success it returns the list of all the topics available that the producers have created.
On failure it raises a myConsumerError with the error message.

Dequeue

The function takes one argument namely, topic name. On success the function returns the log message that is dequeued from the requested topic log queue.
On failure it raises a myConsumerError with the error message.

Size

This function takes one argument namely, topic name. On success the function Returns the size of the log queue for the requested topic.
On failure it raises a myConsumerError with the error message.

System Tests

systemtest1

Implemented 5 Producers and 3 consumers with 3 topics using the producer and consumer libraries where producers P1, P2, P3, consumers C1, C2, C3 are registered to topic T1, producers P1, P4, P5, consumer C1 are registered to topic T1 and producers P1, P2, consumers C1, C2, C3 are registered to topic T3.
The files producer_<ID>.txt contains the logs generated by the producer with corresponding ID.
The producers parallelly push the logs from log_file to the server with delay of 1 sec between

two enqueue requests from the same producer to the server. Simultaneously, the consumers which will consume the logs in order.
Crashed the server while the producers are logging and verified that the data is restored correctly from the database.

systemtest2

Registered a producer and a consumer to the topic DS. The producer logs a message using the Enqueue function and the consumer retrieves the message by the Dequeue function. Verified that the log message retrieved by the consumer is the same as the logged message by the producer.

Health Checkup:

Manager and read replica maintains a database that contains Producer,Consumer and Broker tables whenever the message has been created/consumed the timestamps of the message and ID of the user and broker is stored.

- Producer table consists of producer ID and timestamp of the message/topic that it has been created.
- Consumer table consists of Consumer ID and timestamp of the message that it has been consumed
- Broker table consists of Broker ID and timestamp of the message that it has been modified.

Write Ahead Logging:

Manager maintains the write ahead logging which logs every message that is requested by producer and consumer.
Every log message will contain Transaction ID which is unique for every transaction, broker ID, Request Type and topic/log message. If the transaction is successfully inserted into the database the manager will send the transaction id for the successful operation which is logged into the log file
Write Ahead logging consists of

- LogEvent: This function generates transaction ID and broker,request type, Topic.log message given by the manager logged into the log file.
- LogSuccess: This function takes transaction ID given by manager and logs the successful message updated to the Database.

Crash Recovery: Crash Recovery helps in recovering the messages which have not been updated to the database when the manager is down. It consists of:

- recoverLogs: This function takes broker ID as the argument sent by the broker and sends a list of unsuccesful log messages to the broker.