DISTRIBUTED SYSTEMS

Assignment 3

Report

Prepared by

Mayank Kumar
Shrinivas Khiste
Ishan Goel
Shashwat Shukla
Yashica Patodia

Department of Computer Science and Engineering,
IIT Kharagpur

April 15, 2023

Table of Contents

Table of Contents **iii**

1 Introduction **1**

2 System Architecture **3**

3 System Components **4**

3.1 Load Balancer . . . 4

3.2 Write Manager . . . 4

3.2.1 Functions . . . 5

3.2.2 Database Structures . . . 5

3.3 Read Manager . . . 6

3.3.1 Functions . . . 6

3.3.2 Database Structures . . . 7

3.4 Broker . . . 7

3.4.1 Functions . . . 8

3.4.2 Database Structures . . . 8

4 Features and Design Choices **9**

4.1 Handling Concurrent Requests . . . 9

4.2 Partitioning . . . 9

4.3 Round Robin . . . 10

4.4 Persistence . . . 11

4.5 Maintaining Data Synchronisation . . . 11

4.6 Broker Cluster Management . . . 11

4.7 Health Check . . . 12

4.8 Write Ahead Logging . . . 12

5 Raft Consensus **14**

5.1 Replication of Produce Log . . . 14

6 Testing **15**

6.1 Unit Testing . . . 15

6.2 Integration Testing . . . 15

6.3 System Testing . . . 15

6.4 Concurrency Testing . . . 16

6.5 Performance Testing 16
6.5.1 Average Response Time 16
6.5.2 Throughput 16

1 Introduction

The Raft algorithm is a consensus algorithm designed to manage a replicated log in a distributed system, ensuring that the system remains consistent and fault-tolerant. It is used to elect a leader and synchronize logs across multiple servers. Raft is considered more straightforward and easier to understand than other consensus algorithms, like Paxos.

Here is an overview of the Raft algorithm's consensus module:

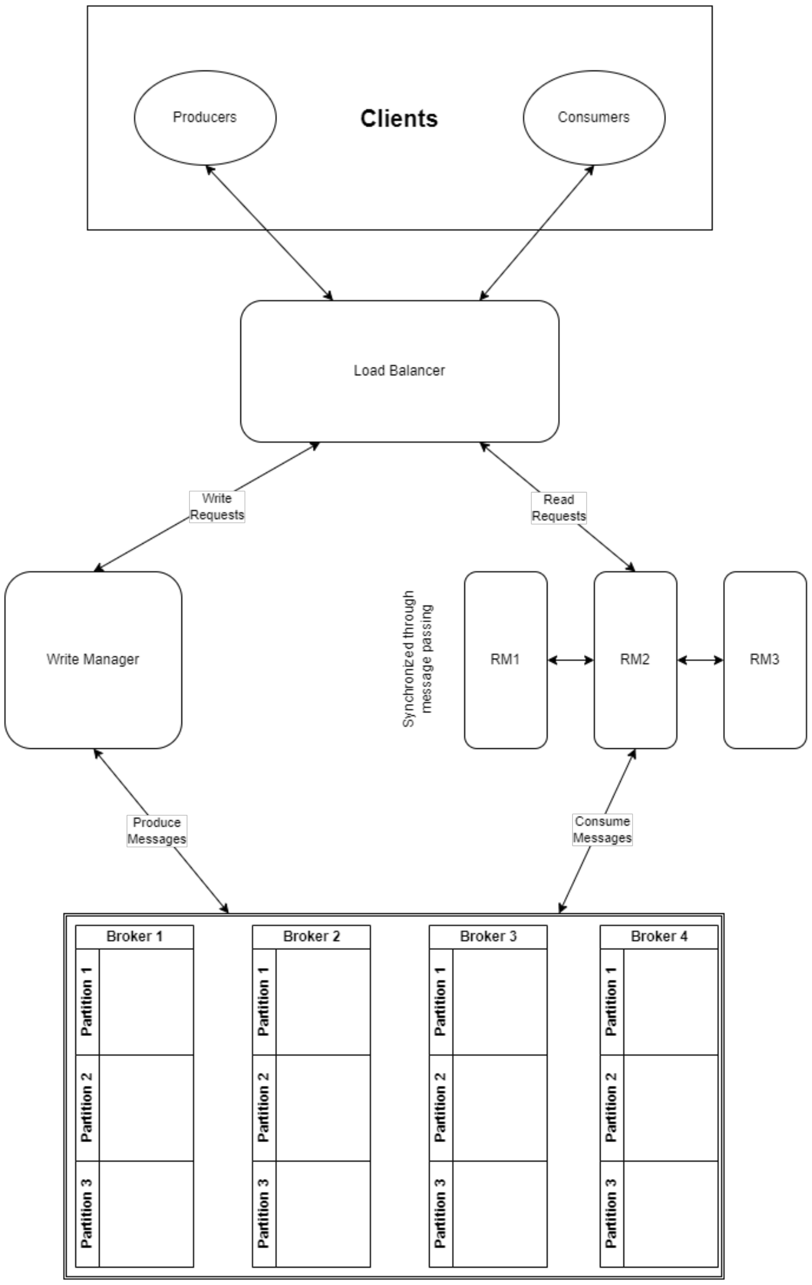
- Server States: Each server participating in the Raft consensus algorithm can be in one of three states: Follower, Candidate, or Leader. At any given time, there is only one leader, while the rest of the servers act as followers. Servers start as followers initially.
- Leader Election: In the absence of communication from the leader, a follower will time out and transition to the candidate state. The candidate then starts a new election term, votes for itself, and sends RequestVote RPCs (Remote Procedure Calls) to other servers in the cluster. The other servers will vote for the candidate if they haven't voted in the current term and if the candidate's log is at least as up-to-date as their own. If the candidate receives votes from a majority of the servers, it becomes the leader.
- Log Replication: Once a leader is elected, it starts accepting client requests and appends them as new entries in its log. The leader then sends AppendEntries RPCs to followers to replicate the new entries. The followers respond with an acknowledgment once they have written the entry to their logs. The leader keeps track of the highest index in the log that each follower has acknowledged.
- Safety and Consistency: To ensure consistency across the distributed system, Raft imposes several constraints on the logs of different servers. For example, if two logs contain an entry with the same index and term, then the logs must be identical up to that point. This helps maintain consistency and ensures that once an entry is committed to the log, it is not overwritten or removed.
- Committing Entries: When the leader receives acknowledgments for a log entry from a majority of the servers, the entry is considered committed. The leader then informs the followers about the committed index, and each server applies the committed entries to their state machine, ensuring that all servers have the same state.

- Leader Failures: If a leader fails, followers will time out and initiate a new election. The server with the most up-to-date log will likely be elected as the new leader, ensuring that committed entries are not lost. This process makes the Raft consensus algorithm fault-tolerant and capable of handling leader failures.

In summary, the Raft consensus algorithm uses a combination of leader election, log replication, safety constraints, and handling leader failures to maintain a consistent and fault-tolerant distributed system.

.

2 System Architecture



//Specific Info

3 System Components

Here is the description of the components of the system with details of their functionality, methods and attributes.

3.1 Load Balancer

This component redirects the write requests (from the producer) to the Write Manager and the read requests (from the consumers) to the multiple Read Managers in a round-robin fashion.

The following endpoints are handled:

1. **POST at /broker**: redirect the request to a Write Manager to add a broker
2. **POST, GET at /topics**: redirect the request to a Write Manager to add or list topics
3. **POST at /producer/register**: redirect the request to a Write Manager to register a producer to a topic
4. **POST at /producer/produce**: redirect the request to a Write Manager to produce a message at a topic
5. **POST at /consumer/register**: redirect the request to a Read Manager to register a consumer to a topic
6. **POST at /consumer/consume**: redirect the request to a Read Manager to consume a message of a topic

Appropriate Error Handling is done to report the type of error that has occurred (HTTP Error, HTTP Connection Error, NULL Response, etc) and which component is responsible for the error (Write Manager or Read Manager and which particular Read Manager).

3.2 Write Manager

This component is responsible for handling the write requests from the producer. This component has been implemented as a class (defined in models/write_manager.py) with all the functionalities and a Flask Server (defined in write_manager_app.py) that serves the endpoints that allow other components to access these functionalities.

3.2.1 Functions

1. **create_tables()**: Function to create all the tables required in the component if they do not already exist.
2. **drop_tables()**: Function to delete the tables to reset the Write Manager data.
3. **add_broker(port: int)**: Function to register a broker at the Write Manager. Stores the port maintaining the ID associated with the broker. It also forwards this message to the RM to maintain data sync.
4. **add_topic(topic_name: str)**: Function to add a new topic at the WM (Write Manager). This adds a topic to the Topic Database and initialises it with one partition assigning a random broker to this partition. The WM forwards the request to the particular broker (to initialise a queue for the partition) and also synchronizes the information about creating a topic with other RMs (Read Managers). The health check mechanism notes that the particular broker has been requested at the current time.
5. **list_topics()**: Function to list all the topics added at the WM. Retrieves the list from the database.
6. **register_producer(topic_name: str)**: Function to register a producer to a given topic. If the topic does not exist, it is created using the add_topic function. This function adds the registration into the Metadata returning a producer id. The Health Check mechanism adds the current time as the first heartbeat of the producer.
7. **produce_message(producer_id: int, topic_name: str, message: str)**: Function to allow a producer to publish a message to a topic. First, it is ensured that the producer id is valid and the producer is registered to the topic. Then, a partition of the topic is selected (in a round-robin fashion) and the request is forwarded to the particular broker to enqueue the message at the partition. The Health Check mechanism updates the last access time of the broker and the producer. If the number of messages of a particular topic exceeds a threshold, a new partition is created for the particular topic using the add_partition function.

3.2.2 Database Structures

These database structures maintain the persistence of the Write Manager. Locks have been used to ensure that concurrent requests can be handled without any errors. Appropriate Error Handling has been done to correctly catch, identify and report the type of error.

1. **BrokerDBMS** (database_structures/write_manager/broker_dbms.py): Handles the data of the brokers maintaining the broker port with an ID. Has functionalities

to add a new broker given a port, delete a broker given a port, get the current number of brokers and get a random broker port.

2. **PartitionDBMS** (database_structures/write_manager/partition_dbms.py): Handles the data of the partitions matching the broker id to the partition. Allows adding a partition given a partition name and broker id, getting the broker port from the partition by using an inner join with the Broker DBMS.
3. **TopicDBMS_WM** (database_structures/write_manager/topic_dbms.py): Handles the data of the topics noting the topic name, number of messages, number of partitions, the current partition offset used during the round robin and a list of partition ids. Allows adding a topic, adding a partition to a topic, listing topics, getting the current partition based on round-robin logic and getting the number of partitions assigned to a topic.
4. **ProducerDBMS** (database_structures/producer_dbms.py): Handles the data of the producers noting the topic name and associating it with an id. Allows adding producers given a topic, getting the number of producers, and checking if a producer id is valid and whether it is linked to the topic.

3.3 Read Manager

This component is responsible for handling the read requests from the consumers. This component has been implemented as a class (defined in models/read_manager.py) with all the functionalities and a Flask Server (defined in read_manager_app.py) that serves the endpoints that allow other components to access these functionalities.

3.3.1 Functions

1. **create_tables()**: Function to create all the tables required in the component if they do not already exist.
2. **drop_tables()**: Function to delete the tables to reset the Read Manager data.
3. **add_broker(port: int)**: Function to register a broker at the Read Manager. Stores the port maintaining the ID associated with the broker. This function is created to maintain data synchronisation between WM and RMs.
4. **add_topic(topic_name: str)**: Function to add a new topic at the RM. This adds a topic to the Topic Database. This function is created to maintain data synchronisation between WM and RMs.
5. **list_topics()**: Function to list all the topics added at the RM. Retrieves the list from the database.
6. **register_consumer(topic_name: str, sync: int)**: Function to register a consumer to a given topic. If the topic does not exist, an exception is thrown. This

function adds the registration into Metadata returning a consumer id. The Health Check mechanism adds the current time as the first heartbeat of the consumer. This information is also synced with the various other RMs. A sync variable is sent so that the RM can know whether the call is from synchronisation or a direct call. If it is a synchronisation call, then the message is not forwarded to other RMs.

7. **consume_message(consumer_id: int, topic_name: str, sync: int)**: Function to allow a producer to consume a message of a topic. First, it is ensured that the consumer id is valid and the consumer is registered to the topic. Then a partition of the topic is selected (in a round-robin fashion), the offset at the partition is retrieved and the request is forwarded to the particular broker to dequeue the message at the partition. The Health Check mechanism updates the last access time of the broker and the consumer. A sync variable is sent so that the RM can know whether the call is from synchronisation or a direct call. If it is a synchronisation call, then the message is not forwarded to other RMs.

3.3.2 Database Structures

These database structures maintain the persistence of the Read Manager. Locks have been used to ensure that concurrent requests can be handled without any errors. Appropriate Error Handling has been done to correctly catch, identify and report the type of error.

1. **BrokerDBMS**: Same as the WM BrokerDBMS.
2. **PartitionDBMS**: Same as the WM PartitionDBMS.
3. **TopicDBMS_WM**: Same as the WM TopicDBMS.
4. **ConsumerDBMS**: (database_structures/consumer_dbms.py): Handles the data of the consumers noting the topic name, and the offset as a dictionary linking the partition name with the offset and associating the consumer with an id. Allows adding consumers given a topic, adding a partition to a topic so that all consumer offsets to the partition get initialised to 0, getting the number of consumers, checking if a consumer id is valid and whether it is linked to the topic, getting the offset given a consumer id and a partition name

3.4 Broker

This component is responsible for handling the queues of all the partitions. It has been implemented as a class Broker (defined in models/broker.py) with all the functionalities, a Flask Server (defined in broker_app.py) that serves the endpoints that allow other components to access these functionalities, and another class MyBroker (defined in myqueue/broker.py) with the functionality to send requests to this Flask server. Appropriate Error Handling is done to catch, identify and report the correct errors.

3.4.1 Functions

1. **reset_dbms**: Function to reset all the tables in the component.
2. **create_topic(topic_name: str)**: Function to add a new topic at the broker. It checks if the topic already exists and returns an error if it does.
3. **list_topics()**: Function to list all the topics added at the broker. Retrieves the list from the database.
4. **enqueue(topic_name: str, message: str)**: Function to add a message to the topic queue. It checks if the topic exists and then gets the topic queue and appends the message to the end of the queue.
5. **dequeue(topic_name: str, offset: int)**: Function to dequeue a message from a topic queue at the given offset. It checks if the topic exists and if any message is left to be retrieved and then retrieves the message from the topic queue at the offset.

3.4.2 Database Structures

1. **Topic DBMS** (database_structures/topic_dbms.py): Handles the topic/ partition and the message queue for the topic/partition. Allows adding a topic, getting the topic list, and getting the topic queue, A TopicQueueDBMS class is also defined here that works on the same table but handles the topic queue enqueuing, retrieving at an offset and checking the remaining size of the queue.
2. **Message DBMS** (database_structures/message_dbms.py): Handles the message data at the broker level. The topic queues just store the ids of the messages. The message corresponding to the id is maintained here. It allows functionality to add a message and retrieve a message given an id.

4 Features and Design Choices

4.1 Handling Concurrent Requests

When a Flask Server receives a request from a client, it must be able to handle it effectively. To handle multiple requests simultaneously, the Flask Server can use a multi-threaded approach. The threaded = True parameter in the app.run() function call enables Flask to handle multiple requests concurrently by creating multiple threads within the server. Each thread can handle a separate request, and Flask can manage them all in parallel.

A Write Manager and Load Balancer are components that are often used in our distributed system and these will handle large amounts of traffic. By enabling the threaded = True parameter in these components, Flask can handle concurrent requests in parallel. This means that multiple clients can send requests to the Flask Server at the same time, and the server can respond to each request without having to wait for the previous request to complete.

To verify that the Flask Server can handle concurrent requests with the threaded = True parameter, a dummy application was created. This application had an endpoint that would sleep for a certain duration before returning a successful response. By testing this endpoint with the threaded=True parameter set, the server was able to handle multiple concurrent requests via Postman but with the threaded=False parameter set the server could not handle concurrent requests and waited until the previous request was completed.

4.2 Partitioning

In a distributed messaging system, messages are published to a topic, which acts as a logical grouping of messages. When a topic receives a large volume of messages, it can become difficult to handle them efficiently. To address this, the topic is divided into multiple partitions, which are essentially smaller subsets of the topic.

Each partition is assigned to a specific broker randomly, which is responsible for storing and managing the messages in that partition. By dividing the topic into partitions, the system can scale horizontally, as each broker can handle a smaller subset of the messages.

Initially, a topic has only one partition, which is sufficient for handling a small volume of messages. However, as the number of messages increases, the system automatically increases the number of partitions for that topic. This allows the system to handle larger volumes of messages without becoming overwhelmed.

By dynamically adjusting the number of partitions, the system can allocate resources more efficiently. For example, if a topic receives only a small volume of messages, it can have a small number of partitions, which means that fewer brokers are needed to handle the messages. This helps to prevent the wastage of resources, as brokers are only allocated to topics that require them.

Thus, partitioning a topic allows a distributed messaging system to handle large volumes of messages efficiently by dividing the workload among multiple brokers. By dynamically adjusting the number of partitions based on the volume of messages, the system can scale up or down as needed, while avoiding unnecessary resource usage.

4.3 Round Robin

When using a distributed messaging system, it's common to have multiple partitions in a topic to handle a large volume of messages efficiently. Each partition acts as an independent unit of the topic, and it's responsible for storing a subset of messages in the order they were received.

In such a scenario, the producer, who sends the message to the messaging system, doesn't need to specify the partition id to which the message needs to be published. The messaging system takes care of distributing the message to one of the available partitions based on a partitioning scheme defined by the system.

This approach is logical because it simplifies the job of the producer, who doesn't need to know anything about the internal details of the messaging system. The producer only needs to publish the message on the topic, and the messaging system takes care of distributing the message to the appropriate partition.

When a consumer subscribes to a topic, it receives messages from all partitions of that topic in a round-robin fashion. This means that the messaging system will send messages from one partition to a consumer, then switch to another partition, and so on until all partitions have been covered. This approach ensures that the load is distributed evenly among all partitions and consumers.

In summary, when using a distributed messaging system with multiple partitions, the producer doesn't need to specify the partition id while publishing the message, and the messages are consumed in a round-robin fashion from all partitions, ensuring that the load is distributed evenly among all partitions and consumers.

4.4 Persistence

Making a system robust to failures is essential to ensure that it remains functional even during unexpected events such as server crashes. In order to achieve resilience against unexpected system failures, it is essential to ensure that all data is persistently stored, i.e., the data should be stored in a way that it can survive a system crash or other unexpected events. In the given scenario, the data about messages, producers, and consumers are persisted using a PostgreSQL database.

PostgreSQL is a popular and open-source relational database management system (RDBMS) that allows for reliable storage and retrieval of data. By using a PostgreSQL database, the system is ensuring that all the data related to messages, producers, and consumers are stored securely and persistently.

Therefore, if the flask server crashes, the data will remain safe and accessible in the database. This means that even if the system experiences a failure, the data can be retrieved and used when the flask server is restarted. This approach ensures that the system can recover from a failure and continue to function seamlessly without any loss of data or functionality.

In conclusion, using a PostgreSQL database to store all data related to messages, producers, and consumers is an effective way to ensure the robustness and resilience of the system. By using this approach, the system can recover from unexpected events and continue to function without any loss of data or functionality.

4.5 Maintaining Data Synchronisation

There are two types of managers: a write manager and multiple read managers. The write manager is responsible for updating the data or adding new data to the queue, while the read managers access the data.

To ensure that the read managers always have access to the latest version of the data, synchronization between the write manager and read managers, as well as among read managers, is done via message passing. This means that when the write manager updates the data, it sends a message to each read manager notifying them of the change.

Furthermore, any metadata update that affects the read managers, such as changes in the topics, or the addition of brokers, is also sent to each read manager individually by the write manager. This ensures that all read managers are aware of any changes that may affect their functionality.

4.6 Broker Cluster Management

Broker Cluster Management involves managing the brokers, that is checking their status, and adding and removing brokers whenever necessary. We have handled the status management of the brokers in the Health Check Mechanism. The addition and removal of the brokers are handled by the Write and Read Managers.

Whenever a new broker is added, a request is sent to the Write Manager and the Read Managers. Also, whenever a broker has crashed, which can be realised by the health check mechanism, we have sent a request to the Write Manager and the Read Managers to remove the broker from their list, so that any future requests are not forwarded to that broker.

4.7 Health Check

We have implemented a health check mechanism for producers, consumers and brokers. The algorithm for the above three is as follows:

- Create a table in the database to log the last updated time for the three actors.
- The table has the following columns: Type (Producer, Consumer, Broker), ActorID, and LastUpdatedTime
- We update the last updated time when any of the functionalities are performed for the respective actors.
- We run the health checkup mechanism periodically and check whether any of the actors are inactive for an interval of more than the threshold difference.
- If so, we display the respective actor ids on the console.

When the broker is unreachable for 5 minutes, it is marked as failed. It no longer is used for assigning partitions to new topics, and we assume that the broker is inactive.

4.8 Write Ahead Logging

The Write-Ahead Logging (WAL) mechanism is a method of ensuring the durability of data changes in a database system. When a write operation is performed on a database, the WAL mechanism ensures that the changes are written to a log file before they are applied to the actual database. This log file is written to disk and can be used to recover the database in case of a system failure.

In PostgreSQL, the Write-Ahead Logging (WAL) mechanism works by writing a sequential stream of records, called WAL records, to one or more dedicated WAL files on disk. These files are separate from the main data files, which contain the actual database content. When a transaction begins to modify data in PostgreSQL, it writes the changes to memory buffers called "dirty pages." Before the transaction is considered complete, the changes are written to the WAL files on disk, along with a record of the changes that were made. This ensures that the changes are safely persisted to disk and can be recovered in the event of a system failure.

The WAL records contain information about the changes made to the data, such as the old and new values of modified rows, the type of operation performed (e.g., insert, update, delete), and other transaction-related metadata. The WAL files are typically stored on a separate disk or disk array to reduce contention with the main data files and improve performance.

When PostgreSQL starts up, it reads the WAL files to determine the state of the database at the time of the last checkpoint. If a failure occurs, PostgreSQL can use the WAL files to "replay" the changes that were made since the last checkpoint to recover the database to a consistent state. This process is known as crash recovery.

In addition to enabling crash recovery, the WAL mechanism also supports other features such as point-in-time recovery, replication, and high availability. By using the WAL files, these features can provide fine-grained control over the state of the database and ensure that data is safely and durably persisted to disk.

This Write Ahead Logging in PostgreSQL can be saved by setting the autocommit parameter of connection to True. By default, the WAL files are located in the PostgreSQL data directory for UNIX based system it is in */usr/local/pgsql/data/pg_wal/*, whereas for Windows it is stored inside *C:/Program Files/PostgreSQL/version/data*. These log files can then be viewed in the human-readable form with the help of *pg_waldump* command.

5 Raft Consensus

We use the Python library pysyncobj that provides a simple way to create fault-tolerant, self-healing, and consistent distributed systems using the RAFT consensus algorithm.

RAFT is a distributed consensus algorithm designed to be easy to understand and implement, enabling distributed systems to maintain a consistent state across multiple nodes or partitions.

Using the pysyncobj library, we created a distributed system where each node (or broker) runs an instance of the SyncObj class, which represents a single node in a distributed system. Each SyncObj instance manages its own state and communicates with other nodes to achieve consensus on the system's overall state.

5.1 Replication of Produce Log

Our broker class inherits Sync Obj class. Now any member method of this class can be synchronised among the brokers in order to achieve consensus by adding the decorator @replicated_sync. When the write manager/read manager sends any request to the broker it is sent along with the partition id. This request is replicated among all the brokers using this pysync obj library.

Now each of the individual brokers decides on whether or not to act upon it based on the presence of the partition id inside the broker's database.

6 Testing

To ensure that the code for the tasks described above is functioning correctly, a comprehensive testing process is required, that include several steps.

6.1 Unit Testing

Unit tests is used to verify that each component of the system is working as expected. The partitioning mechanism is evaluated by creating a mock topic and confirming that it creates a partition when the message limit per partition has exceeded. The broker manager is tested by generating mock clients, topics, and partitions and ensuring that the manager correctly stores metadata and mappings between clients, topics, and partitions.

Additionally, the round-robin algorithm is assessed to ensure that it equitably distributes requests among partitions. Each broker will be tested by creating a mock partition and verifying that the broker accurately manages the partition and persists partition data.

6.2 Integration Testing

In the integration testing phase, it is verified that different components of the system interact effectively. For example, verifying that partitions can be created and assigned to brokers correctly for partitioning. In the case of the broker manager, it is ensured that the manager correctly handles requests from producers and consumers and directs requests to the appropriate broker.

The service discovery mechanism is also assessed to confirm that clients can connect to the correct broker. Brokers are tested to verify that they can communicate with the broker manager and correctly manage their assigned partitions.

6.3 System Testing

System testing is used to verify that the entire system works as expected. For this phase, a realistic scenario is created, where producers and consumers send and receive data through the system. The system's scalability is tested by increasing the number of producers and consumers, and verifying that the system can manage the increased load. The system is also tested for fault tolerance by simulating failures in different components of the system and verifying that the system can recover from the failures.

6.4 Concurrency Testing

The concurrency of the entire queue system was tested by creating different consumers and producer threads.We can get a detailed idea of concurrency testing by seeing the throughput plots in the performance testing section. There we can see that what happens to the number of responses once input requests are increased by multi-threading.

6.5 Performance Testing

6.5.1 Average Response Time

The average response time, in the context of computer systems or web applications, refers to the average time it takes for a system or application to respond to a user's request or action. It is a common performance metric used to evaluate the efficiency, speed, and overall user experience of a system or application. The following table shows the average response time for different requests.

Type of Request	Average Response Time
Adding Topics	0.089
Registering Producers	0.010
Producing Messages	0.225
Registering Consumers	0.028
Consuming Messages	0.272
Request	0.247

Hence the average time for a request is 0.247. We here note that the average response time for registering a consumer is greater than that of producer because while registering consumers we sync all the read managers through message passing which corresponds to the extra timing.

6.5.2 Throughput

Throughput is the system's ability to handle high loads. In other words, the number of tasks accomplished per unit time. The goal of the distributed computing system is to have the best possible performance, in other words, to minimize the latency and response time while increasing the throughput.

Produce Message

The throughput oscilates somewhere around 60req/s once it reaches the stable state.

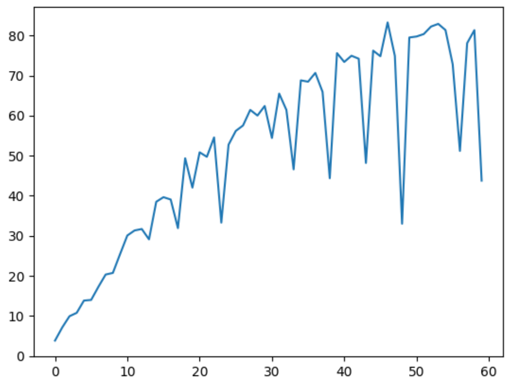


Figure 6.1: Produce Message

Read Message

The throughput oscilates somewhere around 60req/s once it reaches the stable state

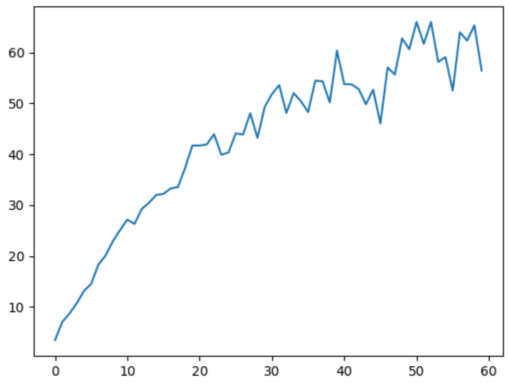


Figure 6.2: Read Message

Register Consumer

The throughput saturates somewhere around 100req/s once it reaches the stable state with little oscilations.

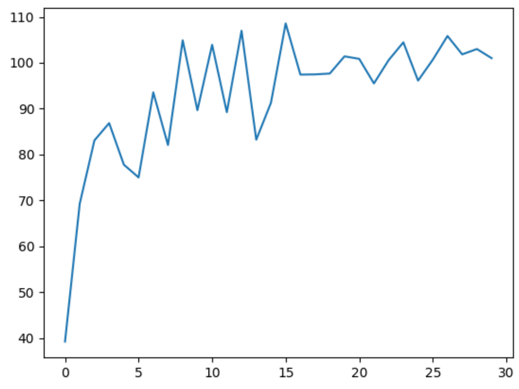


Figure 6.3: Register Consumer

Add Topic

The throughput saturates somewhere around 35req/s once it reaches the stable state with little oscilations.

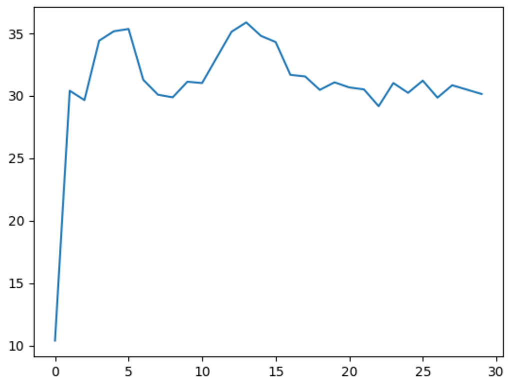


Figure 6.4: Add Topic