Distributed Messaging Queue

Team No. 52

Team members:

Soumodipta Bose

Sudipta Halder

Padam Prakash

Ashutosh Gupta

Guided By:

Lokesh

Introduction

A distributed messaging queue is a system that allows multiple applications on same/different devices to communicate asynchronously with each other. It provides a way for one application to send a message to another application, without the need for a direct connection between the two. This project aims to develop a distributed messaging queue system that can be used by various applications to communicate with each other.

In a distributed system, where components may be located on different machines and may fail independently, a message queue helps to provide fault tolerance, scalability, and reliability.

The use of a distributed message queue offers several benefits, including

- Decoupling: Message queues allow applications to communicate without having to know the exact details of the receiver or sender. This helps to reduce dependencies and makes it easier to change or upgrade individual components without affecting the entire system.
- Scalability: By distributing the load across multiple message brokers, a distributed message queue can handle large volumes of messages and can scale horizontally as the number of clients and messages increases.
- Reliability: A message queue provides a reliable and persistent store for messages, ensuring that they are not lost in the event of a system failure or network interruption. This is important for systems that require high availability and fault tolerance.
- Asynchronous communication: A distributed message queue enables asynchronous communication, allowing applications to continue

processing messages without waiting for a response from the receiver. This can help to improve system performance and reduce latency.

System Design

The design of a distributed message queue system involves several components, including message producers, message consumers, and message brokers. The message producers are responsible for creating messages, which are then sent to the message brokers. The message brokers receive, store, and deliver messages to the message consumers.

The following are the key components of a distributed message queue system:

- Message Producer: A message producer is an application or process that generates messages and sends them to the message queue for processing. In a distributed message queue, the message producer sends messages to the message brokers, which then store and deliver the messages to the message consumers. Message producers can be located on different machines and can send messages asynchronously.
- Message Broker: A message broker is a software component that receives, stores, and delivers messages. It acts as a middleman between the message producer and message consumer. The message broker is responsible for ensuring that messages are delivered in the correct order, and that they are not lost or duplicated. In a distributed message queue, multiple message brokers are used to form a cluster, which provides fault tolerance and high availability.

- Message Consumer: A message consumer is an application or process that receives messages from the message broker and processes them. In a distributed message queue, message consumers can be located on different machines and can receive messages asynchronously. They can also subscribe to specific topics or channels to receive only the messages that are relevant to them.
- Topic or Channel: A topic or channel is a logical grouping of messages. In a distributed message queue, messages are sent to a specific topic or channel, and message consumers can subscribe to one or more topics or channels to receive only the messages that are relevant to them.
- Protocol: A protocol is used to define the rules for communication between message producers, message brokers, and message consumers. We have used REST Api for communication purposes.
- Cluster: A cluster is a group of message brokers that work together to provide fault tolerance and high availability. In a distributed message queue, multiple message brokers are used to form a cluster, which ensures that messages are delivered even if one or more brokers fail.

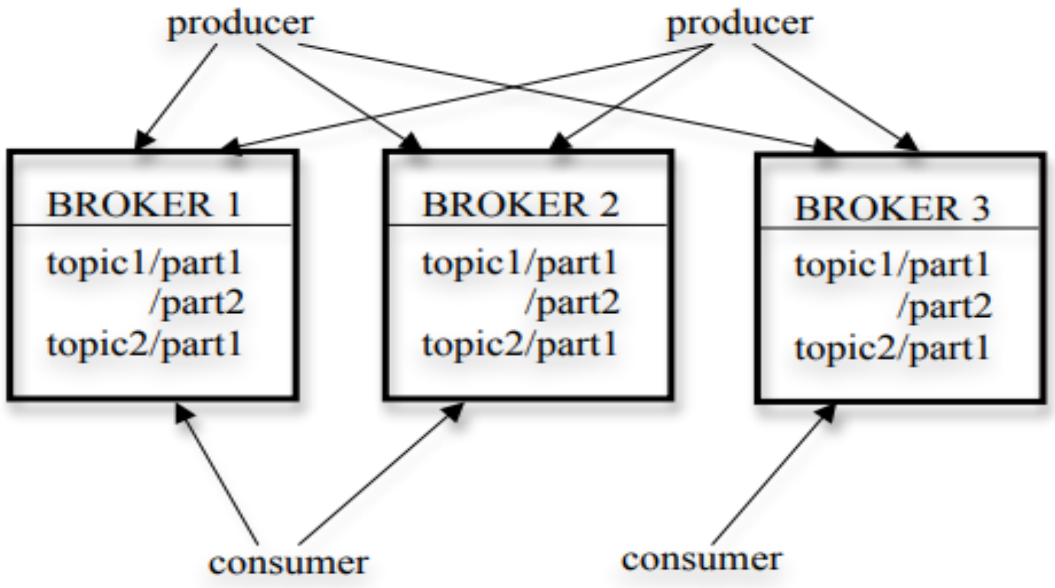


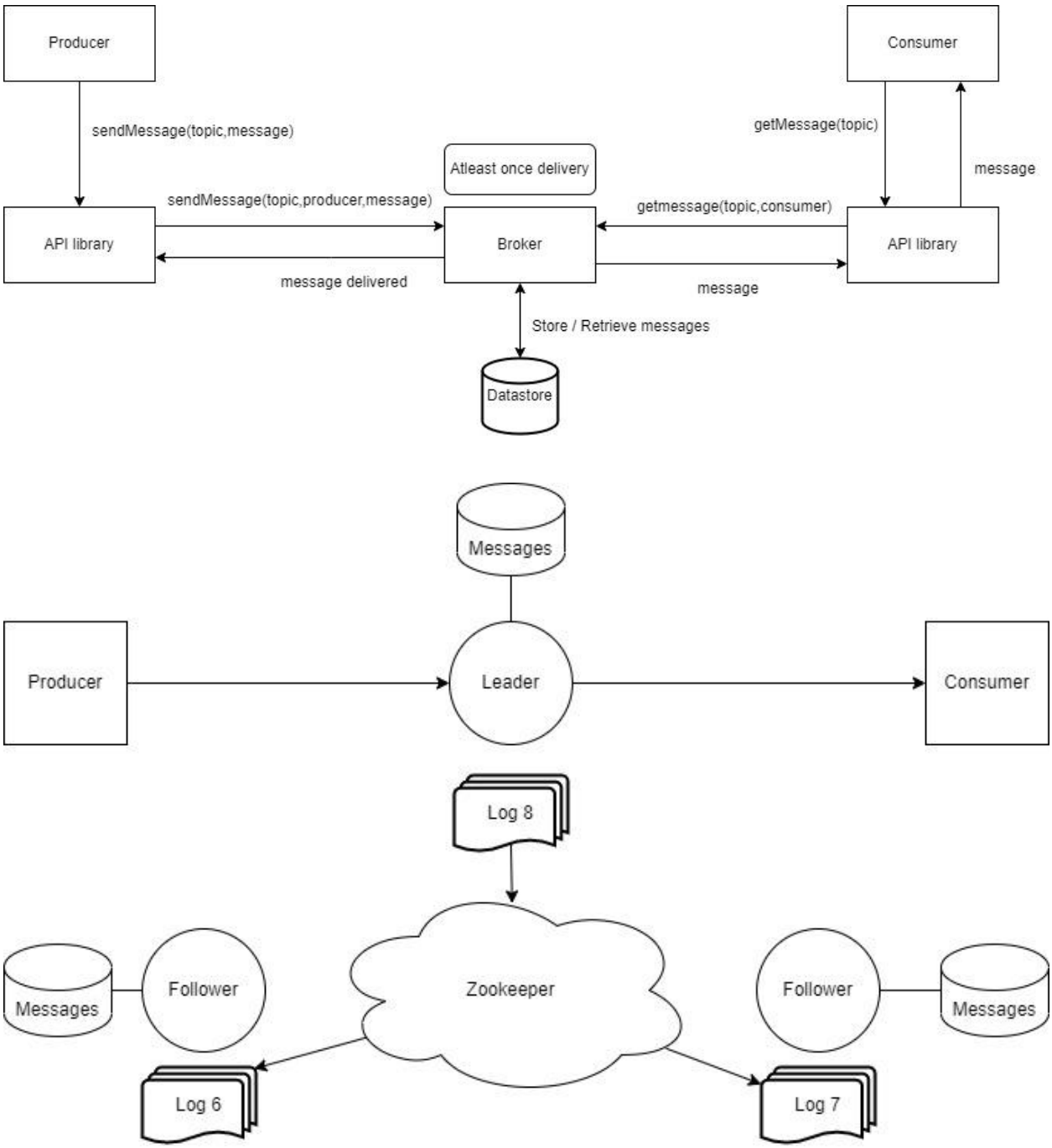
Figure 1. Kafka Architecture

Implementation Details

Technical Specifications

- Hardware:
  - A single computer , or VM with at least 4GB memory and at least 20GB of storage with 2 cores for Message Broker deployment.
  - Any normal computer for using the message queue.
- Software:
  - Language : Python
  - Frameworks : Flask
  - Database: SQLite

Design



We have implemented a distributed message queue using Python and ZooKeeper. The message queue is composed of producers, who can publish messages to topics, and consumers, who can consume messages from topics.

The code initializes a Flask app, sets up a ZooKeeper client, creates paths for various ZooKeeper nodes, and creates an ephemeral node for this instance in the election node. The ephemeral node is used for leader election.

When the instance becomes the leader, it sets the leader node in ZooKeeper. The Flask app has endpoints for health checks, clearing the database, publishing messages to topics, and consuming messages from topics.

The **publish** endpoint accepts POST requests with a JSON payload containing the name of the topic and the message to be published. The endpoint retrieves the topic ID from the database and inserts the message into the message queue for that topic.

The **consume** endpoint accepts POST requests with a JSON payload containing the name of the topic. The endpoint retrieves the topic ID, offset, and size from the database. If there are messages in the queue that the consumer has not yet consumed, the endpoint retrieves the next message and increments the offset for that topic. If there are no more messages in the queue, the endpoint returns a 204 status code.

There are also functions for initializing and clearing the database, and for becoming the leader and starting an election.

Replication

Replication refers to the practice of creating duplicate copies of messages and storing them on multiple nodes in the system. This helps to increase fault tolerance and reduce the risk of data loss in case of failures. When a node fails or goes offline, the replicated messages can be used to recover the data and ensure that no messages are lost.
We implemented this by creating a znode called logs and appending all the operations as SQL statements in each children node of the log. All the other brokers have a watcher setup on their end that gets notified when a new log is entered and automatically executes those logs, to stay up to date with the leader, to ensure reliability.

At Least once Delivery

At least once delivery is a guarantee that a message will be delivered to a consumer at least once, but potentially more than once. This means that in case of any failure during message processing, the message will be re-delivered to the consumer until the processing is successful. This approach ensures message reliability and prevents message loss. We have implemented it by using an acknowledgement message which ensures confirmation of delivery of message to the destination. We have followed the following for the same:

Producer -> Broker: The producer sends the message to the broker and the producer keeps on retrying to send the message incase of network failure until it receives an acknowledgement from the broker.
Broker -> Consumer: Broker first sends the message with its sequence number, only when consumer receives the message it tells the broker to consume and only then the message is consumed and offset is incremented.

Leader Election

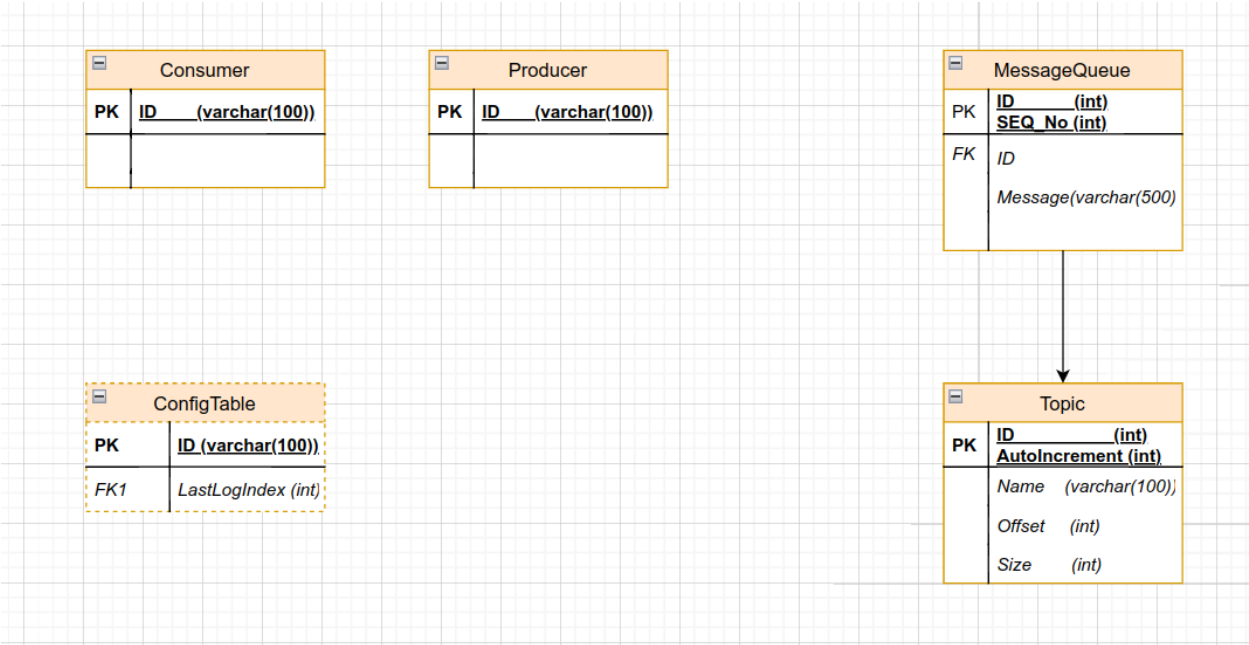
Leader election algorithm is a process by which a group of nodes (or processes) elect a leader from among themselves to coordinate and manage the overall group activities. In a distributed system, it is important to have a leader election algorithm to ensure that only one node acts as the leader at any given time, preventing conflicts and ensuring that the system operates correctly and the others replicate all the operations performed by the leader.
We have implemented this by setting up a znode in Zookeeper and informing the other brokers of the new leader using a Watcher service that gets triggered automatically when a broker is down.

Push based message Delivery

Push-based message delivery is a pattern where the server (message queue) actively pushes messages to the clients (consumers) as soon as they become available, without waiting for clients to request them.
The consumers are not constantly polling the server for new messages. Instead, they enter a sleep mode when there are no messages in the queue, and only wake up when there is a new message to consume. This approach can reduce unnecessary network traffic and improve performance by avoiding the overhead of constant polling.

When a new message arrives, all the consumers in sleep mode are notified, and they compete to consume the message from the queue. This ensures that the message is consumed as soon as possible and reduces the risk of duplicate messages being sent.

ER Diagram



Performance Analysis

Performance analysis of a distributed message queue depends on various factors such as the architecture of the message queue, the number of brokers or nodes, the type of messaging pattern, message size, message rate, and the number of consumers. For different metrics, performance analysis is as followed:

- Throughput: It refers to the number of messages processed per unit of time. A distributed message queue should have high throughput to handle large volumes of messages efficiently. It depends upon Hardware resources, Network latency, message size and volume, consumer load etc.
- Latency: It is the time taken by a message to travel from the producer to the consumer. Low latency is desirable in a distributed message queue to ensure that messages are delivered in a timely manner.

- Reliability: It is the ability of a distributed message queue to handle message delivery and processing failures without losing any messages.
- Consistency: It refers to the ordering of messages delivered to the consumers. A distributed message queue should ensure that messages are delivered in the same order as they are produced.

Our project is working fine in terms of these metrics.

Scope of Improvement

Our project can be extend to include following features further:

- Partition : Partition in a distributed messaging queue refers to the division of a message queue into multiple smaller queues, each of which can be independently managed and processed by a single consumer.
- Consumer group: a consumer group is a set of consumers that work together to consume messages from a topic or a set of topics. Each consumer group has a unique group identifier, and multiple consumer groups can consume from the same topic. When a message is sent to a topic, it is delivered to all consumer groups that are consuming from that topic. However, only one consumer within each consumer group will receive each message. This means that if you have multiple consumers within a consumer group, each consumer will receive a subset of the messages.
- Scalability: A distributed message queue should be able to scale horizontally by adding more brokers or nodes to handle an increasing number of messages and consumers.

We haven't incorporated these features due to less time and their complexity. In the case of a distributed messaging queue, Scalability, consumer groups, and

partitioning are good features that provide important benefits. However, implementing them is complex and time-consuming. Therefore, due to the time constraints, we have to prioritize other critical features and leave out these features for a later phase.

Conclusion

We have developed an implementation of Distributed Message Queue. A Distributed Message Queue (DMQ) is a distributed system that provides reliable and scalable messaging for applications. It allows different applications to communicate with each other asynchronously, decoupling the sender and receiver, and provides the ability to handle a large volume of messages with high throughput.

DMQs are widely used in modern distributed systems due to their ability to handle large amounts of data, provide reliable and fault-tolerant message delivery, and scale horizontally as the system grows.

We developed the implementation of DMQs with careful consideration of various factors, such as message durability, message ordering, message delivery guarantees, and distributed coordination.

In terms of performance, our implementation of DMQs can handle millions of messages per second, depending on the system configuration, message size, and workload. However, the performance of DMQs can be affected by factors such as network latency, disk I/O, and message size. Therefore, it's essential to tune the DMQ system based on the workload and requirements of the application.

References

- https://notes.stephenholiday.com/Kafka.pdf
- https://github.com/rabbitmq/internals