SYSTEM DESIGN 7

Message Queues:

A **message queue** is a form of communication between different components, services, or applications in a distributed system. It allows data (messages) to be exchanged asynchronously, ensuring that the sender and receiver do not need to interact with each other directly at the same time.

Key Characteristics:

1. **Asynchronous Communication**: The sender can send a message without waiting for the receiver to process it immediately.
2. **Decoupling**: Message queues decouple the sender and receiver, allowing them to operate independently.
3. **Reliable Delivery**: Messages are stored in the queue until they are successfully processed and acknowledged by the consumer.
4. **Scalability**: Queues can handle high loads by distributing messages among multiple consumers.

Why Are Message Queues Important?

Message queues are crucial for building modern, scalable, and fault-tolerant systems. Below are some of the key reasons why they are important:

1. Decoupling

- **Problem**: Tightly coupled components increase the complexity of changes and deployments.
- **Solution**: Message queues enable loose coupling, making systems easier to scale and maintain.

2. Asynchronous Processing

- **Problem**: Synchronous communication can lead to bottlenecks when a component is slow or busy.
- **Solution**: Message queues allow tasks to be processed in the background, improving system responsiveness.

3. Load Balancing

- **Problem**: Uneven workloads can overwhelm certain parts of the system.

- **Solution**: Multiple consumers can process messages from the queue, distributing the load evenly.

4. Fault Tolerance and Reliability

- **Problem**: Failures in one part of the system can cause the entire system to fail.
- **Solution**: Message queues store messages until they are successfully processed, ensuring no data loss.

5. Scalability

- **Problem**: Scaling systems can be challenging when components communicate directly.
- **Solution**: Message queues allow you to scale producers and consumers independently.

6. Prioritization and Ordering

- **Problem**: Certain messages or tasks might need to be processed before others.
- **Solution**: Many message queue systems support priorities and FIFO (First-In-First-Out) or custom ordering.

7. Interoperability

- **Problem**: Different applications may use different languages, frameworks, or platforms.
- **Solution**: Message queues act as a universal interface for communication across diverse systems.

Common Use Cases:

1. **Background Processing**: Sending emails, generating reports, resizing images.
2. **Event-Driven Architectures**: Processing events like user actions, IoT device updates.
3. **Data Pipelines**: Streaming data for ETL processes or machine learning pipelines.
4. **Microservices Communication**: Enabling seamless interaction between microservices.
5. **Task Scheduling**: Delaying tasks or scheduling periodic jobs.

Popular Message Queue Systems:

- **RabbitMQ**: Lightweight and feature-rich with support for AMQP.
- **Apache Kafka**: Distributed event streaming platform, ideal for high-throughput use cases.
- **Amazon SQS**: Fully managed message queue service.
- **Redis Streams**: Lightweight, in-memory queue with stream support.
- **ActiveMQ**: Reliable open-source message broker with support for multiple protocols.

Kafka :

1. Producers

A **Producer** is an application or service that sends records (messages) to Kafka topics.

- **How It Works:**
    1. Producers write data to specific topics.
    2. Data can be partitioned within a topic based on a key (e.g., user ID), ensuring all messages with the same key go to the same partition.
    3. Producers can specify the partition or let Kafka assign one (based on a partitioner algorithm).
    4. Kafka acknowledges the write based on the producer's `acks` configuration / min in sync replica config:
        - **acks=0**: No acknowledgement is required.
        - **acks=1**: Acknowledgment from the leader partition only.
        - **acks=all**: Acknowledgment from all replicas (strongest guarantee).

How Kafka Producers Know the Partition and Leader Node:

1. **Metadata Lookup**:
    - When a producer starts, it fetches the metadata from any Kafka broker it connects to.
    - The metadata includes:
        - List of brokers in the cluster.
        - List of partitions for the topic.
        - Leader node for each partition.
2. **Partition Selection**:
    - Based on the producer configuration:

        - **Keyed Partitioning**: If a key is provided, Kafka uses a partitioner to consistently map the key to a partition (e.g., `hash(key) % num_partitions`).
        - **Round-Robin or Default Partitioning**: If no key is provided, the producer uses a round-robin approach or an implementation-specific strategy to select a partition.
3. **Leader Node**:
    - Once the producer knows the partition, it retrieves the leader node for that partition from the metadata.
    - It sends the message directly to the leader broker of the selected partition.

What If the Producer Doesn't Know the Leader?

If the producer doesn't know the leader for a partition (e.g., due to stale metadata or a new topic being created):

1. **Initial Request**:
    - The producer sends the request to any **bootstrap broker** from its configuration.
    - The bootstrap broker is only used to fetch metadata and isn't necessarily the leader for any partition.
2. **Metadata Refresh**:
    - The producer queries the bootstrap broker for up-to-date metadata.
    - The broker responds with details about partitions, leaders, and replicas for the requested topic.
3. **Retry**:
    - Once the producer gets the updated metadata, it retries the request to the correct leader node for the partition.

---

2. Consumers

A **Consumer** reads messages from Kafka topics and processes them.

- **How It Works:**
    1. Consumers subscribe to topics and read messages from assigned partitions.
    2. Each consumer tracks its **offset**, the position in the partition where it last consumed a message.

3. Consumers can commit offsets explicitly (manual commit) or rely on automatic commits.
4. Kafka ensures that each message in a partition is delivered in order.

---

3. Consumer Groups

A **Consumer Group** is a collection of consumers that share the work of consuming messages from a topic.

- **How It Works:**
  - Each partition in a topic is consumed by one consumer within the group.
  - If there are more partitions than consumers, some consumers will consume multiple partitions.
  - If a consumer fails, the partitions assigned to it are reassigned to other consumers in the group (rebalance occurs).
- **Advantages:**
  - Parallelism: Multiple consumers can process data in parallel.
  - Scalability: Add more consumers to process more partitions.

---

4. Data Replication Algorithm

Kafka uses **Leader-Follower Replication** for fault tolerance and durability.

- **Replication Factor:**
  - Each partition has a replication factor, indicating how many replicas of the partition exist (e.g., replication factor = 3 means one leader and two followers).
- **Min In-Sync Replicas (ISR):**
  - Specifies the minimum number of replicas that must acknowledge a write for it to be considered successful.
  - Ensures durability and consistency under failures.
- **How It Works:**
  - Each partition has a **leader** that handles all reads and writes.
  - Other replicas act as followers, replicating the leader's data.
  - If the leader fails, a new leader is elected from the ISR.
- **Algorithm:**
  - Kafka uses the **Zookeeper consensus** or **Raft** (in KRaft mode) for maintaining metadata about leaders, replicas, and synchronization.

Replication Factor

The **replication factor** defines **how many copies of a partition** will be maintained across the Kafka cluster for fault tolerance.

- For example:
    - If the replication factor is 3, each partition will have **1 leader** and **2 follower replicas** spread across different brokers.

Min In-Sync Replicas

Min In-Sync Replicas (min.insync.replicas) defines the **minimum number of replicas (including the leader)** that must acknowledge a write for it to be considered successful.

- **Acknowledgment Behavior:**
    - If a producer sends a message with `acks=all`, the Kafka broker will wait for acknowledgments from at least `min.insync.replicas` before confirming the write to the producer.
    - If the number of in-sync replicas falls below `min.insync.replicas` (e.g., due to broker failures), the broker will reject write requests with `acks=all`.

How They Work Together

- **Replication Factor**: Determines **how many total copies** of the data exist.
- **Min In-Sync Replicas**: Ensures **a subset of those copies** are acknowledged as successfully written.

Example:

1. **Replication Factor**: 3
    - Partition replicas:
        - Leader on Broker 1
        - Follower on Broker 2
        - Follower on Broker 3
2. **Min In-Sync Replicas**: 2
    - At least 2 replicas (including the leader) must acknowledge the write for success.
    - If only 1 replica is in sync (e.g., during a failure), Kafka will not accept writes with `acks=all`.

---

5. Metadata Exchange

Metadata exchange ensures producers and consumers are aware of Kafka's structure (brokers, topics, partitions).

- **Key Processes:**
  - **Producers and Consumers Query the Cluster**:
    - When they connect, they query Kafka brokers to fetch metadata about topics, partitions, and leaders.
  - **Metadata Caching**:
    - Producers and consumers cache metadata locally to reduce the overhead of querying the cluster frequently.
  - **Leader Updates**:
    - If the leader of a partition changes, metadata updates are pushed to clients.
- **Mechanisms:**
  - Kafka brokers maintain metadata and serve it to clients.
  - Zookeeper (or KRaft in newer versions) ensures metadata consistency.

---

6. Rebalancing in Kafka

Rebalancing occurs when the assignment of partitions to consumers in a group changes.

- **Triggers for Rebalancing:**
  1. A new consumer joins the group.
  2. A consumer leaves the group.
  3. Topic partitions are added or removed.
  4. A broker or consumer fails.
- **Rebalancing Types:**
  1. **Static Rebalancing**:
     - Consumers disconnect and rejoin during rebalance, causing downtime.
     - Used in older versions of Kafka.
  2. **Incremental Cooperative Rebalancing**:
     - Consumers retain some partitions while others are reassigned incrementally.
     - Reduces downtime, introduced in newer Kafka versions.
- **How Consumer Rebalancing Works:**

1. A consumer group has a **group coordinator** (a Kafka broker).
2. Consumers send heartbeat messages to the coordinator to indicate they are alive.
3. If a consumer leaves or fails, the coordinator triggers a rebalance.
4. The **partition assignment strategy** determines which partitions are assigned to which consumers:
    - **Range Strategy**: Assigns consecutive partitions to each consumer.
    - **Round Robin Strategy**: Distributes partitions evenly.
    - **Sticky Strategy**: Tries to minimize partition movement between rebalances.

SCENARIO:

Suppose I have 4 Nodes, I have defined replication factor as 3 and min in sync replica as 3 Now suppose I have a Topic T1 and 12 Partitions How each leader and followers of each partition will be picked, as I have 4 nodes and replication factor is 3 I need to choose 2 follower nodes out of 3 Nodes. Now if any of the 2 nodes goes down , how data consistency or availability will be handled?

1. Partition Leader and Follower Selection

You have:

- **4 nodes**: N1, N2, N3, N4.
- **Replication factor**: 3 (1 leader + 2 followers per partition).
- **Partitions**: 12 for Topic T1.

Partition Assignment

Kafka assigns partition leaders and followers to nodes using a **round-robin algorithm** while ensuring:

- **Even distribution**: Leaders and replicas are evenly spread across all available nodes.
- **No overlap for replicas**: A single node does not host multiple replicas of the same partition.

Example Distribution

For **12 partitions** with a replication factor of 3 across 4 nodes:

1. **Leaders** are distributed round-robin:
    - Partition 0 leader: N1
    - Partition 1 leader: N2
    - Partition 2 leader: N3
    - Partition 3 leader: N4
    - And so on...
2. **Followers** are selected to ensure replication:
    - Partition 0: Leader = N1, Followers = N2, N3
    - Partition 1: Leader = N2, Followers = N3, N4
    - Partition 2: Leader = N3, Followers = N4, N1
    - Partition 3: Leader = N4, Followers = N1, N2
    - This continues for all partitions.

This distribution ensures a balance of leaders and replicas across nodes.

2. Handling Node Failures

Case 1: Two Nodes Go Down

- Let's assume N1 and N2 go down.
- Leaders for partitions hosted on N1 and N2 become unavailable.

Impact on Consistency and Availability:

1. **Min In-Sync Replicas**:
    - Since `min.insync.replicas=3` and only 2 nodes are available, no partition can satisfy this requirement.
    - **Writes with `acks=all` will fail**, as at least 3 in-sync replicas are required to acknowledge a write.
2. **Reads**:
    - **Data Availability**: If a leader for a partition is on N1 or N2, the partition becomes unavailable until one of these nodes recovers. Kafka won't automatically reassign a leader to another node unless configured to do so.
    - **Consistency**: Kafka cannot guarantee data consistency because writes are failing, and there may be a mismatch between replicas when nodes recover.

3. Recovery and Resilience

When the Nodes Recover:

1. **Replica Synchronization**:
    - Once N1 and N2 come back online, Kafka will sync their replicas with the current state of the leader (on N3 or N4).
    - If data was written while these nodes were down, it will backfill the missed data from the leader.
2. **Leader Election**:
    - Kafka will elect new leaders for partitions if necessary, ensuring all partitions have active leaders.

Best Practices for High Availability:

1. **Reduce Min In-Sync Replicas**:
    - Setting `min.insync.replicas=2` allows writes to continue even if 2 nodes are down.
    - This trades off some consistency for higher availability.
2. **Enable Unclean Leader Election (If Needed)**:
    - By default, Kafka does not allow unclean leader election (a non-in-sync replica becoming the leader) to prevent potential data loss.
    - Enabling this (`unclean.leader.election.enable=true`) can improve availability but risks losing the latest writes.
3. **Increase the Number of Nodes**:
    - Adding more nodes distributes the load and reduces the impact of node failures.

how kafka tracks for a topic if multiple consumer groups are present , which consumer group has consumes which data?

1. The Role of `__consumer_offsets` Topic

The `__consumer_offsets` topic serves as a **centralized store** for offset information across all consumer groups. It tracks the last committed offset for each **consumer group**, **topic**, and **partition** combination.

2. How Offsets Are Stored

- **Key Structure**: The key in the `__consumer_offsets` topic is composed of:
    - **Consumer group ID** (`group_id`).
    - **Topic name** (`topic`).
    - **Partition number** (`partition`).

- This combination ensures that each consumer group's offset for a specific topic and partition is unique.
- **Value Structure**: The value stores information related to the committed offset, such as:
    - **Committed offset** (the last processed message position).
    - **Metadata** (optional information about the commit).
    - **Timestamp** (the time of the commit).
    - **Leader epoch** (version of the partition leader).

---

3. Offsets Commit Process

When a consumer commits an offset:

1. **Commit Signal**:
    - The consumer sends a commit signal, either manually or automatically, indicating that it has successfully processed a batch of messages and reached a certain offset.
    - If `enable.auto.commit=true` (auto commit mode), the offset is committed periodically.
    - If `enable.auto.commit=false` (manual commit mode), the consumer explicitly commits the offset when ready.
2. **Partition Identification**:
    - Kafka uses the **consumer group ID** (`group_id`) to determine which partition of the `__consumer_offsets` topic the offset should be stored in.
    - This is done by calculating a **hash** of the `group_id` to decide which partition within `__consumer_offsets` will store the offset record.
3. **Key-Value Pair Storage**:
    - Kafka constructs the **key** using:
        - `group_id` (consumer group).
        - `topic` (the topic being consumed).
        - `partition` (the partition in the topic).
    - It then updates or appends a **value** containing the **committed offset** and other metadata.
4. **Writing to `__consumer_offsets`**:
    - Kafka writes this key-value pair to the determined partition in the `__consumer_offsets` topic.
    - If the consumer is committing for the first time or updating its last known offset, the record is either appended or replaced, depending on

whether an existing offset record for that `<group_id, topic, partition>` combination exists.

---

4. Offset Retrieval Process

- When a consumer restarts or a rebalance occurs, it needs to know the last committed offset for the partitions it's assigned to.
- Kafka:
    1. **Hashes** the `group_id` to identify the partition of the `__consumer_offsets` topic.
    2. Searches for the key corresponding to the `<group_id, topic, partition>` combination.
    3. Retrieves the last committed offset from the value.
    4. If no offset exists (e.g., the consumer group is new), the consumer starts from the earliest or latest offset, depending on its configuration (`auto.offset.reset`).