Message flow in kafka:

💡 0. Core Concepts Before We Begin

- Producer: Publishes messages to Kafka.
- Consumer: Reads messages from Kafka.
- Topic: Logical grouping/category to which messages are sent.
- Broker: Kafka server that stores data and serves clients.
- Partition: Each topic is split into one or more partitions for parallelism and scalability.
- Offset: A unique identifier for each message within a partition.
- Consumer Group: A group of consumers cooperating to read from a topic.

🛫 1. Message Creation & Publishing (Producer Side)

Step-by-step:

1. **Message Created: A producer creates a message, usually consisting of:**
    - Key (optional): Used for partitioning logic.
    - Value: The actual payload (event data).
    - Headers / Metadata (optional).
2. **Topic Decided: The producer chooses a topic (e.g., `"orders"`) to send the message to.**
3. **Partition Decision:**
    - If a key is provided, Kafka uses a hashing algorithm (e.g., `hash(key) % number_of_partitions`) to decide which partition.
    - If no key, Kafka may use round-robin or random distribution.

4. **Message Sent to Broker:**
    - Kafka client sends the message to the leader broker for that partition.
    - Kafka writes the message sequentially to a commit log file (append-only) on disk.

🧱 2. Message Storage (Kafka Broker Side)

- Each partition is a log file on disk.
- Message is written with a monotonic offset, unique within that partition (starting from 0).
- Kafka does not delete messages after consumption. Instead, it retains messages for a configurable period (e.g., 7 days).
- Brokers replicate partitions across other brokers for fault tolerance (controlled by replication factor).

📦 3. Message Consumption (Consumer Side)

Consumer Mechanics:

1. Consumer subscribes to a topic (e.g., `"orders"`).
2. Kafka assigns specific partitions to each consumer (within a consumer group).
3. Consumers read messages sequentially from the assigned partition using the offset.

🔁 4. Offset Management

- Offset is the position of a message in a partition.
- Consumers maintain an offset per partition to track how far they have read.
- Two types of offset storage:

    - Automatic (Kafka managed): Kafka stores it in a special topic (`__consumer_offsets`).
    - Manual (App managed): Developers store offsets in external systems or commit them manually.

- **Offsets can be committed:**
    - Automatically after consumption.
    - Manually, based on business logic (e.g., after processing is successful).

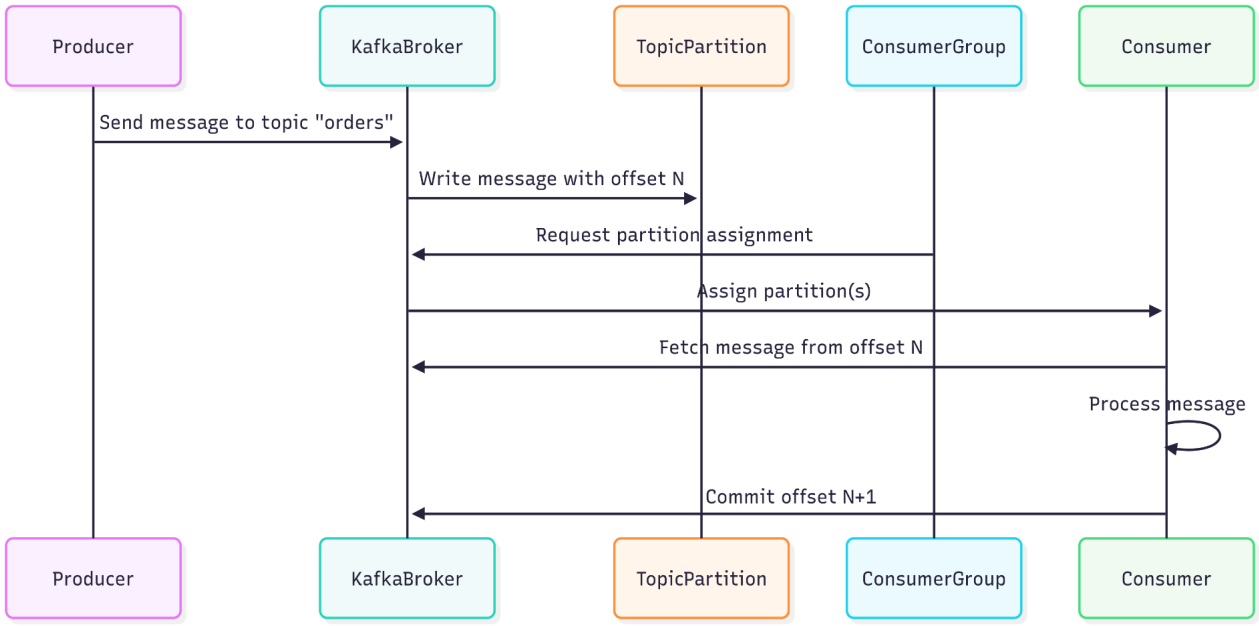
👥 5. Multiple Consumers (Consumer Groups)

Key Concepts:

- A consumer group allows multiple consumers to read from the same topic in parallel.
- Kafka guarantees each partition is read by only one consumer within a group.

Scenarios:

- If there are more partitions than consumers, some consumers get multiple partitions.
- If there are more consumers than partitions, some consumers remain idle.
- If each consumer is in a separate group, each one gets the full copy of all topic data (like pub-sub broadcast).



🔀 6. Partition Rebalancing

- When a consumer joins or leaves a group, Kafka rebalances partitions.
- Rebalancing involves pausing consumption temporarily and redistributing partitions.

🧠 7. Fault Tolerance and Durability

- Kafka's replication factor ensures data isn't lost if a broker crashes.
- Consumers can resume from last committed offset after crash/restart.
- Producers get ACKs based on configured acks=0/1/all for durability guarantees.

🔍 Intricacies You Should Know

Intricacy	Detail
Message Ordering	Guaranteed only **within a partition**, not across partitions.
Backpressure Handling	If consumer is slow, offset lag increases, but Kafka doesn't block producer.
Compaction	Kafka can **compact** messages with the same key (latest retained).
Retention Policies	Based on time (e.g., 7 days) or size (e.g., 1GB per partition).
High Throughput	Due to zero-copy, batch writes, and sequential disk I/O.
Exactly Once Semantics (EOS)	Supported with proper configuration: idempotent producers, transactions, and offset coordination.

✅ TL;DR (In One Line Each Stage)

1. Producer creates a message and sends it to a topic.
2. Kafka decides partition via key hashing or round-robin.
3. Broker writes the message to the partition log with an offset.
4. Consumer reads from the topic by partition and offset.
5. Offsets track progress; committed offsets let consumers resume.

6. Consumer groups ensure parallelism with one consumer per partition.

7. Kafka manages rebalancing, fault tolerance, and retention.

Apache Kafka vs RabbitMQ

Apache Kafka

Kafka is an open-source distributed event streaming platform developed by the Apache Software Foundation. It is written in Java and Scala. Kafka uses a data stream for the delivery of messages and is suitable for both online and offline message consumption. The aim of the project is to provide a high-throughput, low-latency platform for real-time data feed handling, streaming analytics, data integration, and mission-critical applications. It employs a pull-based approach and lets users request message batches from specific offsets. Although only being shipped with a Java client, it offers an adapter SDK that allows programmers to build their own system integrations.

Features of Kafka

- High throughput: Built to process millions of messages per second and handle large volumes of data.
- Zero downtime and data loss: Achievable with a replication factor > 1 for brokers.
- Ingest pipelines: Kafka can replicate events in a broker.
- Reliable: Due to its distributed, replicated, partitioned, and fault-tolerant nature.

RabbitMQ

RabbitMQ is an open-source distributed general-purpose message-broker software facilitating efficient message delivery in complex routing scenarios. It runs as a cluster of nodes where queues are distributed across them.

It initially implemented the **Advanced Message Queuing Protocol (AMQP)**. It uses a **push model** and prevents overwhelming consumers via the consumer-configured prefetch limit. It is commonly used for handling background jobs or as a message broker between microservices.

Features of RabbitMQ

- Built-in clustering: Multiple servers can be clustered into a single broker.
- Protocol support: Supports AMQP, STOMP, MQTT, and more.
- Tracing and debugging: Allows investigation when the system misbehaves.
- Broad language support: Clients available for most programming languages.

Note

Kafka and RabbitMQ serve the same core purpose—event/message handling—but with different strengths. Both are open-source, commercially supported pub/sub systems widely used by enterprises.

Kafka vs RabbitMQ: Comparison Table

S.No.	Parameter	Kafka	RabbitMQ
1	Performance	Around 1 million messages per second	Around 4k–10k messages per second
2	Payload Size	Default limit: 1 MB	No strict size constraints
3	Messages Synchronicity	Durable store, allows replay	Can be asynchronous/synchronous
4	Data Unit	Continuous stream	Message
5	Data Type	Operational	Transactional
6	Topology	Publish/Subscribe based	Exchange types: Fanout, Topic, Direct, Header-based
7	Data Flow	Continuous unbounded key-value pairs	Bounded distinct message packets

8	Event Storage Structure	Logs	Queues
9	Message Delivery System	Pull-based (consumer requests messages)	Push-based (messages pushed to queues)
10	Consumer Mode	Smart consumer / Dumb broker	Dumb consumer / Smart broker

---

When to Use Kafka

- When very fast message consumption is required.
- When message replay is needed.
- For high throughput requirements (100K+ events/sec).
- Suitable for real-time analytics, event sourcing, log aggregation, etc.

When to Use RabbitMQ

- When dynamic consumer addition is needed (without publisher changes).
- If message replay is not necessary.
- When supporting legacy protocols like AMQP 0-9-1, STOMP, AMQP 1.0, and MQTT.
- Ideal for legacy systems and complex routing logic.