**MESSAGING SYSTEM:**

\* **point to point** (particular message received by max one receiver, once received it'll be deleted)

\* **pub-sub** (one topic subscribed by multiple consumers. consumer can sub to multiple topics.

durable logs of stream, we can configure the expiry time / size of a topic)

**KAFKA** --> high throughput low latency event streaming platform (real-time data feed).

\***Distributed logging** (ex. logs from various micro services)

\***Stream processing** (ex.data streams in real-time in trade app transactions)

\***Data integration (**ex. hospital health record, lab results, insurance claim integrates all records)

\***Pub/Sub Messaging** (ex.user tweets -pub / notifies all sub)

**Benefits**: Reliability , Scalability , Durability and Performance

**MAJOR USES** :

\* streaming pipelines that move data between different applications.

\* streaming applications that are capable of processing streams of data.

\* fault tolerant storage system that stores streams of records.

**WORKING OF KAFKA**:

\* Kafka stores log/event in topics in partition (logic in producer / default partition method round robin).

\* order in partition maintained with use of a key.

\* The partitions enable the topic to scale beyond a single server and act as the unit of PARALLELISM.

\* IN each partition the events are stored in offset which are IMMUTABLE(append only manner)

\* there will be a replica for each topic's partition in other brokers that the leader topic's partition runs. So that

this will provide fault tolerance even if a broker goes down.

\* TCP protocol is used for communication between clients and Kafka nodes.

**INSTALL KAFKA ON COMPUTER**:

\* Install JDK & Kafka.

\* Start Zookeeper. [bin/zookeeper-server-start.sh config/zookeeper.properties] default port : 2181

\* Start Kafka server. [bin/kafka-server-start.sh config/server.properties] default port : 9092

**EVENT** --> a thing that has happened [NOTIFICATION + STATE]

\* IoT

\* Business process change

\* User Interaction (user clicks a link)\*\*

\* Micro service output.

\* Kafka uses key value pair.

**CLUSTER --> TOPIC --> PARTITION --> OFFSET --> MESSAGE**

**TOPIC IN KAFKA**

\*Fundamental unit of event organization.

\*named container for similar events.

\*can duplicate data between topics.

\*DURABLE "LOGS" of events. (append only - never insert in between)

Can only seek by offset, not indexed.

\* We can make Kafka message expire by size or duration.

\*Messages in Kafka are "IMMUTABLE".

\*default number of partition is 6.

**PARTITION IN KAFKA**

\*If the key value is null,

It would be distributed in round robin method to the partitions.

the message will be distributed evenly.

(IF order doesn't matter)

\*If Key is not null,

run key through hash function and mod num of partitions.

same key has same partition.

(If we need order)

**BROKERS IN KAFKA**

\* An computer, instance or container running the Kafka process.

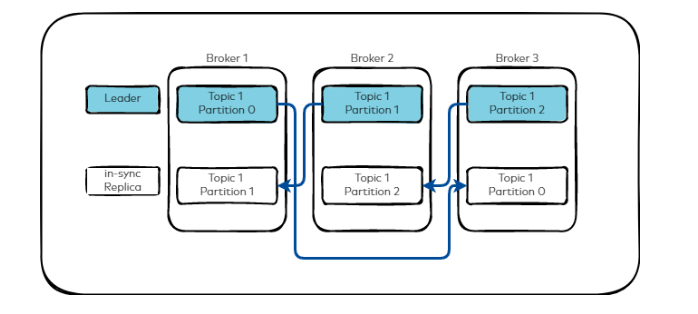
\* If you’re connected to one broker you’re connected to entire cluster(brokers).

\* Manage partitions.

\* Handle write and read requests.

\* manage replication of partitions

\* Intentionally kept simple.



**REPLICATION IN KAFKA (TOPIC LEVEL)**

\* copies of data for fault tolerance.

\* One lead partition and N-1 followers. (can have 0 followers also If we setup)

\* IN general writes and reads happen to the leader.

\* An invisible process to most developers.

\* Tuneable in the producer.

\*Each partition has one leader and one or more follower replicas.

\*Followers can be located on any broker different from the leader's broker. (to maintain fault tolerance)

**ZOOKEEPER**:

\* Zookeeper is a distributed centralized service that coordinates/manages large sets of hosts.

\* Zookeeper is used to provide a configuration service, naming registry, synchronization, and group services in distributed applications.

**NOTE:**

\*\*\* Kafka only provides a total order over messages within a partition, not between different partitions in a topic.

\*\*\*Cluster → Broker → Topic → Partition → Replica (Leader/Follower) → Offset → Message (Key-Value Pair)

\*Java native language of Apache Kafka.

\* **Throughput** refers to the rate at which a system, process, or network successfully completes a task or produces an output within a given period.

**PRODUCER IN KAFKA**

\*API surface of producer library is lightweight.

\* we choose topic, connection pooling, network buffering, partition, security config, and other things in producer.

\* Acks:0 -> No acknowledge from broker. (possible data loss) (speed highest performance) (ex: logs)

\* Acks:1 -> Acknowledgement from leader to producer (limited data loss) (higher performance)

\* Acks: all -> Acknowledgement from leader & all followers to producer (no data loss) (ex: transactions)

**CONSUMER IN KAFKA**

\* many consumers read from one topic.

\* scaling consumer is automatic. (just have to provide customer group ID)

\* After scaling it'll Automatic Rebalancing --> Even Load distribution, Fault tolerance, Scalability.

\*If the number of consumers is more than the number of partitions then some consumers will remain idle as they have no partitions to read from.

\*If the number of partitions is greater than the number of consumers, then each consumer will receive messages from multiple partitions.

\*If the number of consumers is equal to the number of partitions, then each consumer reads messages in order from exactly one partition.

**CONSUMER GROUPS**:

Each message published on a topic will be delivered to one consumer instance within each subscribed consumer group. These consumer instances may either be in separate processes or on separate machines.

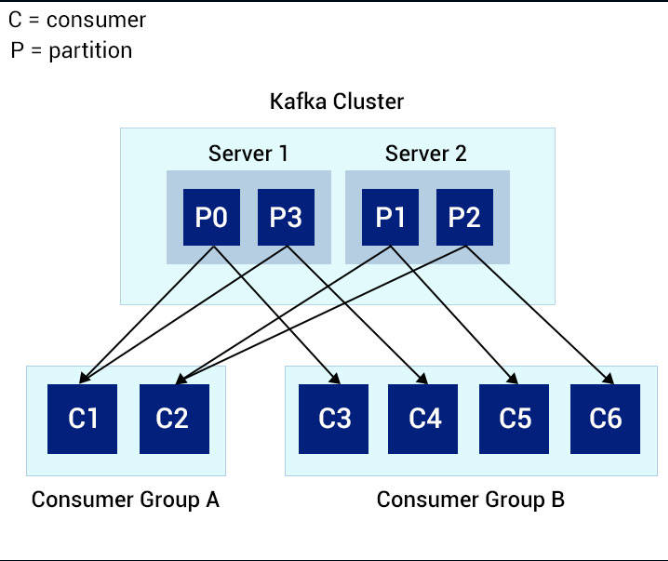
If all the consumer instances are within **the same consumer group**, then the records will **be load balanced** over the instances. (This is a Queuing model).

If all the consumer instances are within **different consumer groups**, then each record will be **broadcast** to all the consumer processes. (This is a Publish-Subscribe model).

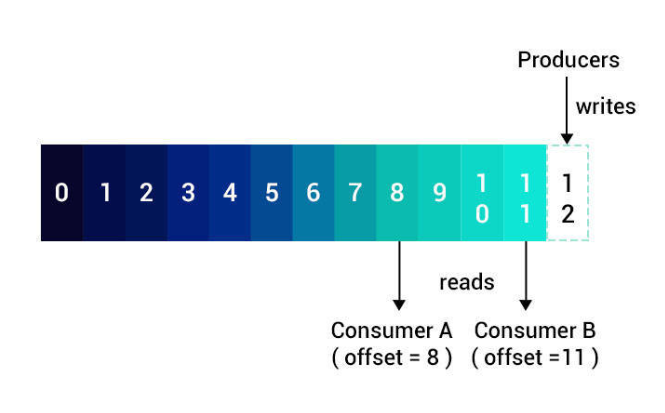
**In Kafka, the consumer group divides processing of messages among its consumer instances, similar to a queue. Again, Kafka broadcasts messages to all subscribing consumer groups, as with Publish-Subscribe.**

Thus, Kafka combines the strength of both these message models, enabling it to easily scale.

**Kafka also provides better ordering guarantees than a traditional messaging system using topic partitions.**



\* The consumer can read data from any position in the partition log - it can move back to an older offset to re-read older data or jump ahead to the latest record and start consuming from there.



**ECOSYSTEM IN KAFKA:**

**KAFKA CONNECT (KC**)

\* Integrates data between any system and Kafka ecosystem.

\* External client process (does not run on brokers)

\* Horizontally scalable

\* Fault tolerance.

**Data source -> KC -> Cluster -> KC -> Data sink**

**SCHEMA REGISTRY**

\* Schema evolves as business evolves(grows).

\* Server process external to Kafka brokers. (external component)

\* Maintain a DB of schemas persisted in internal Kafka topic.

\* the DB is cached in schema registry for better access.

\* HA deployment available.

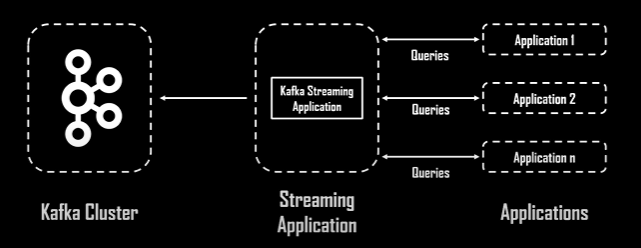
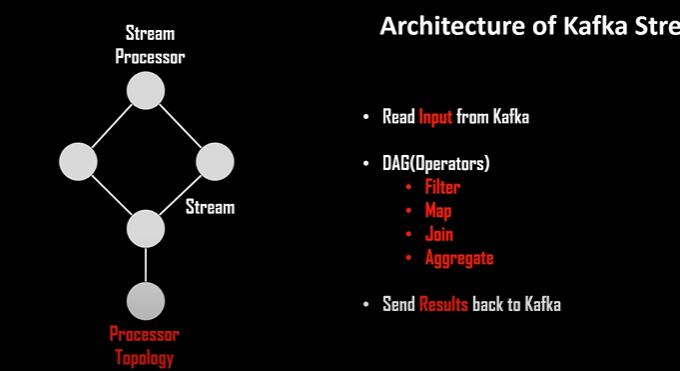
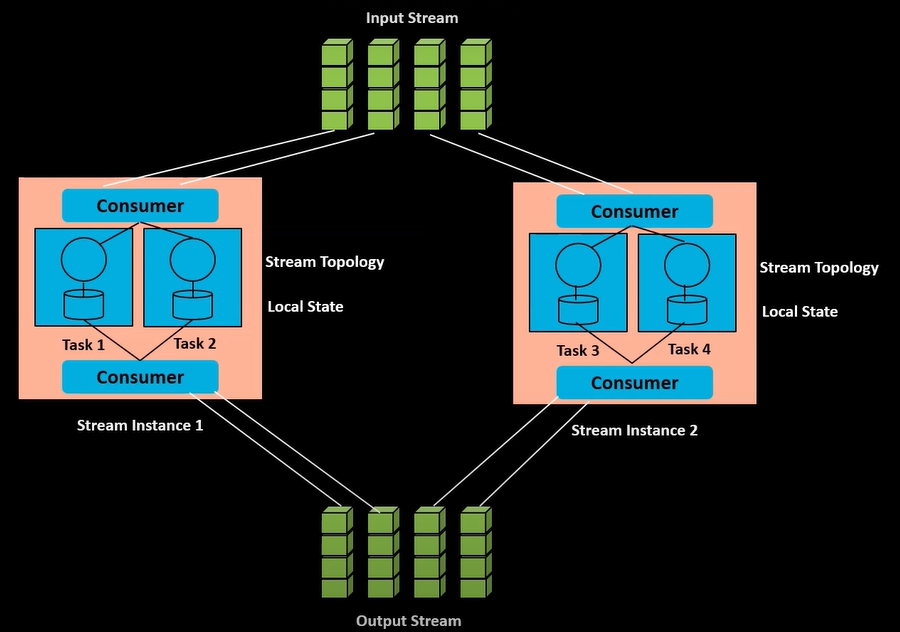
\* Producer prevents incompatible messages from being produced. (same applies for consumer)

\* Schema registry supports Avro, JSON, Protocol Buffer. (Avro schema is robust)

**KAFKA STREAMS**

\*Updated Kafka messaging system build on top of Kafka.

\*Unbounded and continuous flow of data packets in real-time. Data packets are generated in the form of key-value pairs. Automatically transferred from publisher, there is no need to request.

Each dot in above is **Stream processor** and line connecting them is **Stream.**

stream --> unbounded and continuous flow of data packets in realtime.

Data packets are generated in the form of key-value pairs

automatically transferred from publisher, there is no need to request.

Kafka stream archi :

Input stream

output stream

Stream instance --> consumers and local state and stream topology

Input stream & output stream can be one single Kafka cluster.

consumers --> provides input and receives output.

stream topology --> steps in which a process executed.

local state --> memory allocation

stores intermediate data or result provided by

stream topology.

flow :

read from input stream

processes with STREAM PROCESSORS (appy various operators map , filter , join , aggregator)

receive from output stream

features:

elastic , fault tolerance , highly viable , exactly once , java and scala support , integrated security

EXAMPLE WITH TOY FACTORY:

Stream: Think of streams as conveyor belts that carry toy parts (data) into the factory. These parts keep coming without stopping.

Stream Processor: This is like a worker on the assembly line. The worker takes the toy parts (data), puts them together, and makes complete toys (processed data).

Topology: Imagine a map of the entire factory floor, showing all the conveyor belts and workers. This map helps us see how toy parts move through the factory and where they get assembled.

Stateful Processing: Sometimes, workers need to remember how they put together certain toys (like instructions) to make sure all toys are assembled correctly. This memory is like a special notebook they keep, which helps them handle tricky parts.

Fault Tolerance: If a worker gets tired and stops, another worker can come in and keep making toys. This ensures that the toy-making process doesn't stop.

Example Workflow

Input Streams: The factory receives endless boxes of toy parts on conveyor belts. These parts are like pieces of data coming into Kafka Streams.

Processing: Workers on the assembly line take these parts, figure out what they need to do with them, and create toys. In Kafka Streams, this means taking data, doing something with it (like changing it or combining it with other data), and getting it ready for the next step.

Output Streams: Once the toys are made, they are sent out on another conveyor belt to the toy store. This is like sending the processed data to another place for use.

State Management: Workers keep a notebook to remember how to assemble toys in case they need to make similar ones later. In Kafka Streams, this notebook is like a state store that helps keep track of information needed for processing data.

Scaling and Fault Tolerance: More workers can be added to the line to make toys faster, and if one worker takes a break, another one can continue the work without starting over.

Benefits

Easy to Use: The workers have a simple set of instructions to follow, making toy assembly easy.

Integration: All parts of the factory work together smoothly.

Stateful Processing: Workers can handle complicated toy designs because they have their notebooks.

Fault Tolerance: The toy-making process keeps going, even if some workers need a break.

Two special stream processors:

Source Processor - Does not have any upstream processors. It produces an input stream to its topology by consuming records from one or more Kafka topics and forwards it to downstream processors.

Sink Processor - Does not have downstream processors. It sends any received records from its upstream processors to a specified Kafka topic.

Simple, Lightweight Client Library: Kafka Streams is designed to be easy to use and integrate into your existing applications. It doesn't require a separate processing cluster, making it lightweight compared to other stream processing frameworks. You can embed it directly in Java applications, allowing for seamless integration.

No External Dependencies on Systems Other Than Kafka: Kafka Streams operates entirely on Kafka, which means it doesn't rely on external systems like databases or Hadoop clusters for state management or processing. This makes deployment and management simpler, as you only need to maintain Kafka.

Supports Exactly-Once Processing Semantics: Kafka Streams offers exactly-once processing semantics, ensuring that each record is processed once and only once, even in the case of failures. This is crucial for maintaining data accuracy in applications where duplicate or missed records could cause problems.

Supports Event-Time Based Windowing Operations: Kafka Streams provides powerful support for windowed operations based on event time. This allows you to group and process records within specific time windows (e.g., tumbling, sliding, or session windows), making it easier to handle time-based aggregations and analytics.

