**Apache Kafka**

**Usman Arshad (BSEF19M038)**

**Kafka:**

**“Kafka is a distributed Message Streaming Platform that uses publish and subscribe model to stream records.”**

**Centralize vs Decentralize Distributed System:**

In centralized Distributed System, Whole Data is stored and maintained at only one system/location.

**Defects of Centralized Database Management:** No fault tolerance, bottlenecks, vulnerable to thefts, lack of privacy

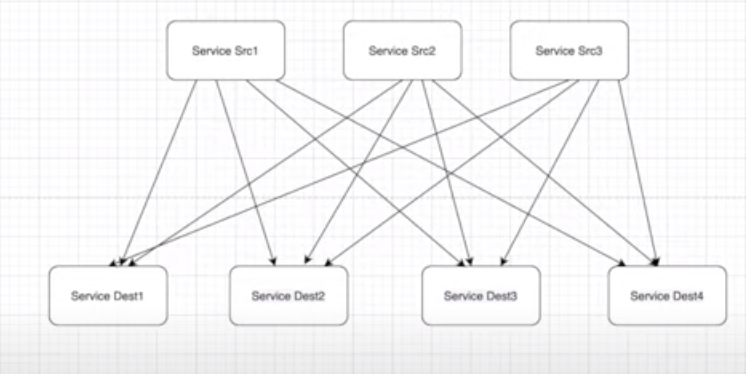
In decentralized distributed system, Whole data is stored at multiple positions either parts of data at multiple locations or copies of the whole data at multiple location.

**Advantages of De-centralized Distributed System:** Fault Tolerance, Scalability, Security

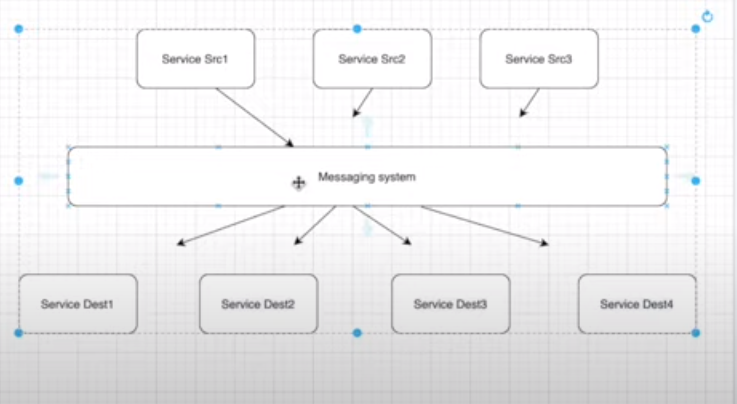
**Disadvantages of De-centralized Distributed System**: More Space Required, Redundancy of Data

**Message Streaming Platform:**

Let’s say we have some Services that are producing some data and there are some destinations that needs that data. We can see that we need many connections in order to achieve this communication.



In order to reduce number of connections, we use messaging system in between services providers and services users. Now the services will send the data to messaging system and the user of that data will fetch this data from the messaging system



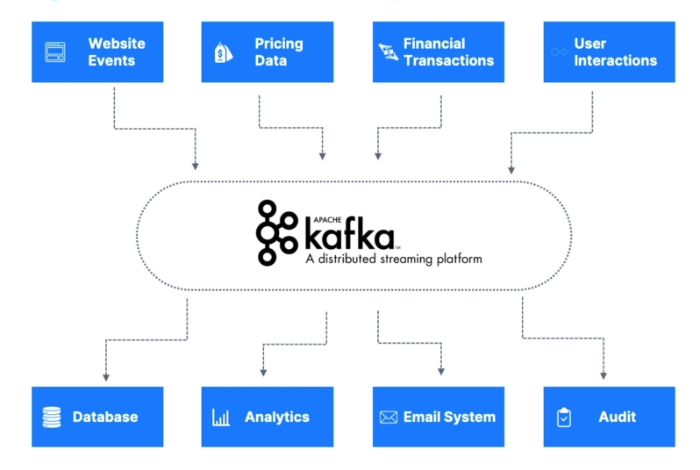
**Question**: How Messaging system is different from Database?

**Answer**: Messaging System not only stores data but also notifies about the data arrival to the receivers that are interested in that data. Moreover, Kafka is scalable, fault tolerant and reliable. It has higher performance, low latency etc. It ensures no data loss.

**Publish-Subscribe Messaging System:**

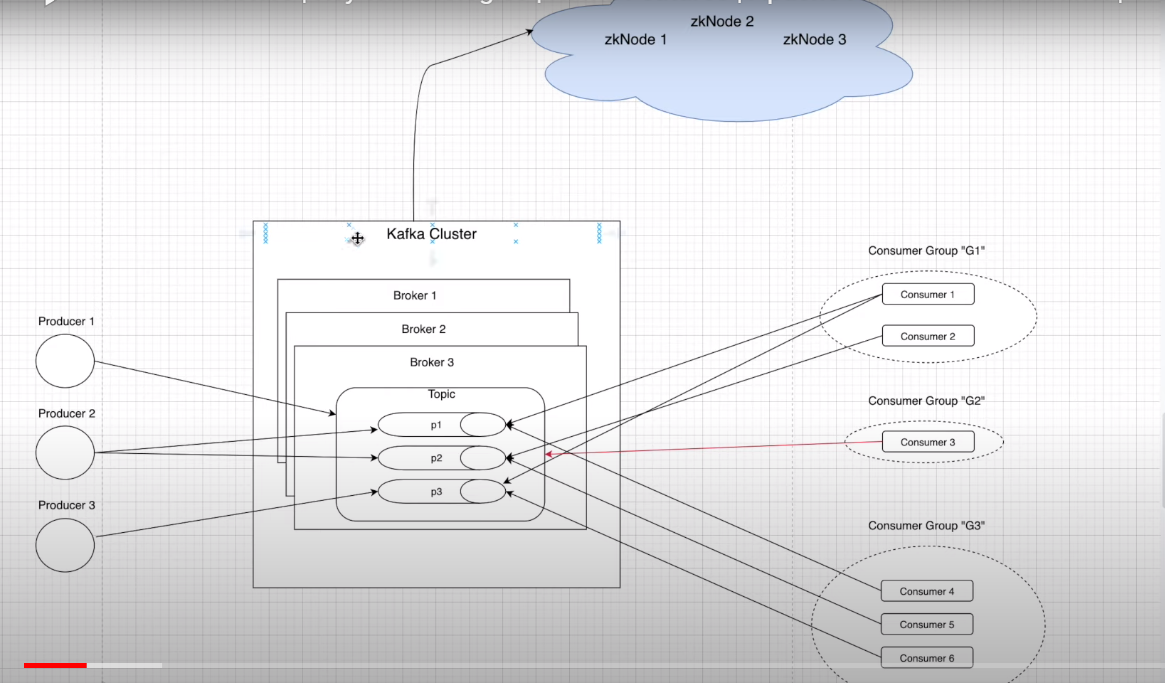
One Message can be consumed by multiple subscribers that is opposite to point-to-point messaging system in which one message can be consumed by only one receiver.

* Messages are persisted as topics
* There is some time dependency that is customizable in kafka (Default 1 week)

**Why Apache Kafka:**

* Distributed, Resilient, Fault Tolerant
* High Performance (latency of less than 10 MS) - real time
* Horizontal Scalability
  + Can scale to 100’s of brokers
  + Can Scale to millions of messages per second

**Kafka Architecture:**

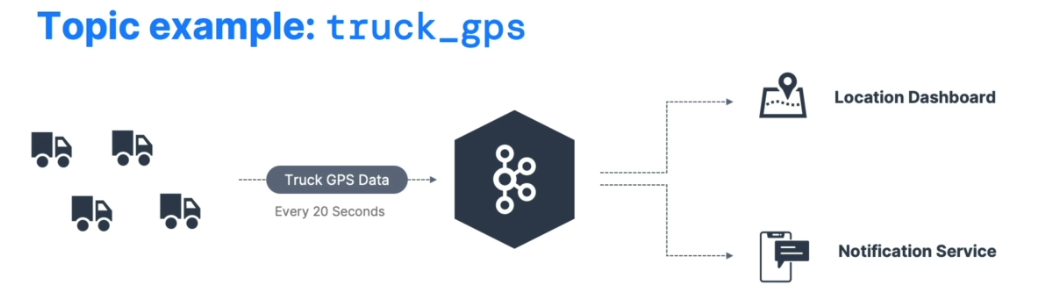


**Kafka Topics:**

Kafka topics are the particular stream of data belonging to some particular category.

Topic is like a table in the database. There can be multiple topics in a cluster. A topic is identified by its name so it has to be unique. Topics support any kind of message format. You can’t query from topic like in database. To store data in topic we use kafka producers and to read messages of topic we’ll use kafka consumers.

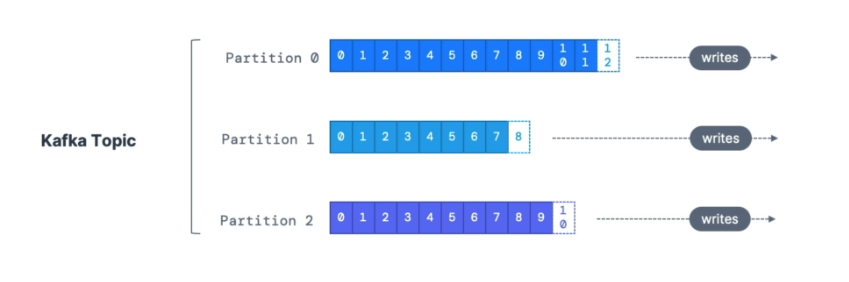
With the help of topics, producers come to know what category of data they have and to which topic they have to store it. Similarly, Consumers know that they are interested in which topic. for example, producers will store data in a topic name Sports and All the applications that are interested in that particular data will fetch this data from Sports topic.



Here is the example of topic named truck\_gps that contains the position of all the trucks. Let’s say we have a fleet of trucks; Each Truck Report its GPS position to kafka every 20 seconds. Each message will contain the truck id and position (longitude and latitude). Then there are multiple consumers that are consuming data from this topic.

**Partitions:**

Topics are split in partitions. You can have as many partitions per topic as you want. Messages within each partition are ordered as seen in fig:

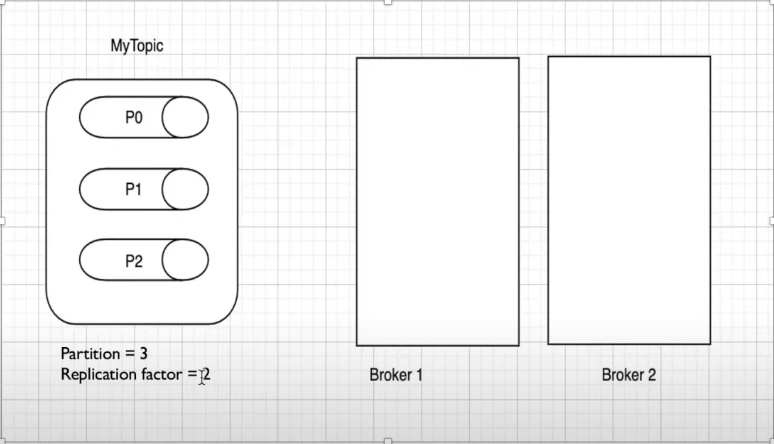


Each message within a partition gets an incremental id called **offset.** Kafka topics are immutable means that one message is written in partition we can’t change it.

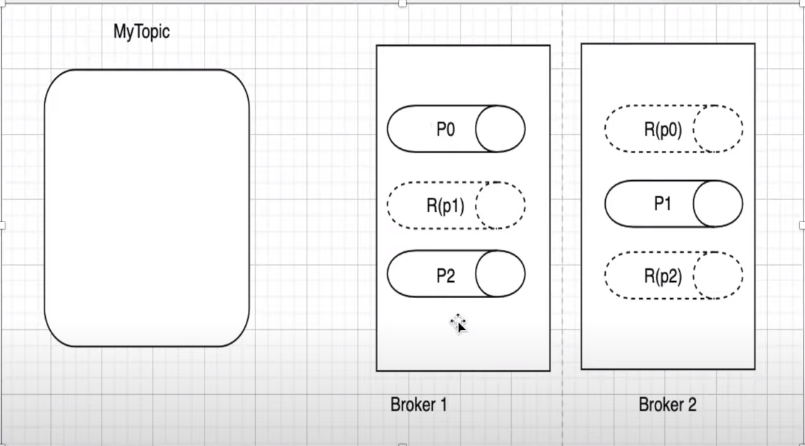
Some basic points about offset:

* Offset only have the meaning for the specific partition which means Offset 3 in partition 1 does not represent the same data as Offset 3 in partition 2.
* Offset are not reused even if the previous messages have been deleted.
* Data is assigned randomly to a partition unless a key is provided.

**Replications** are backup of partitions. Producers can’t produce data to a replication. Similarly, Consumers can’t consume data from replica of the partition. The sole purpose of replica is fault tolerance. Topics should have replication factor >1 normally b/w 2 and 3.



As the replication factor is 2, So it means that each partition will have one replica. The figure of how the replica will be created is discussed in the following diagram:

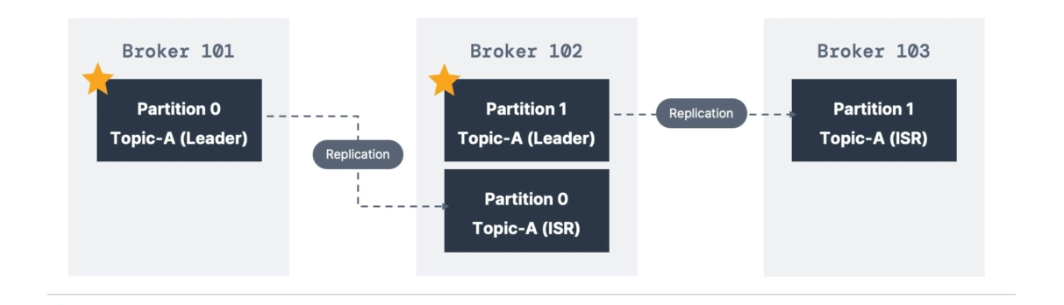


First of all, our partitions are distributed in a round robin fashion and the replica of each partition is present in the alternative Broker. Now if one broker is down, we will not lose any data and can continue to stream records by making replica’s the main partition.

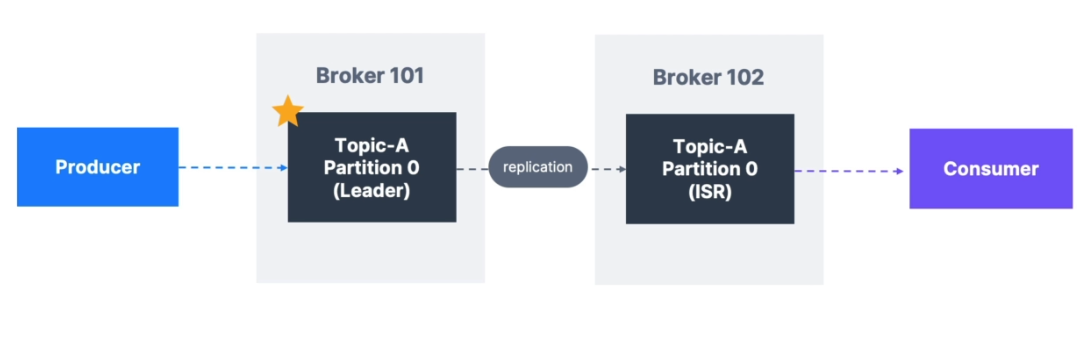
**As a rule, for a replication factor of N, you can lose up-to N-1 brokers without any data loss.**

**Concept of leader of Partition:**

As kafka producer will only write to the leader broker for the partition and similarly, Kafka Consumer will only read from the leader broker for the partition. If the leader broker goes down then the replica broker for that partition can become leader for that partition.



But in latest versions, you can read/write to the closest replica broker for that partition in order to reduce latency as we can see in figure:



**Producers:**

Producers are the applications that write/publish data to the topics within a cluster using the producing API’S. Producers know in advance which partition to write to.

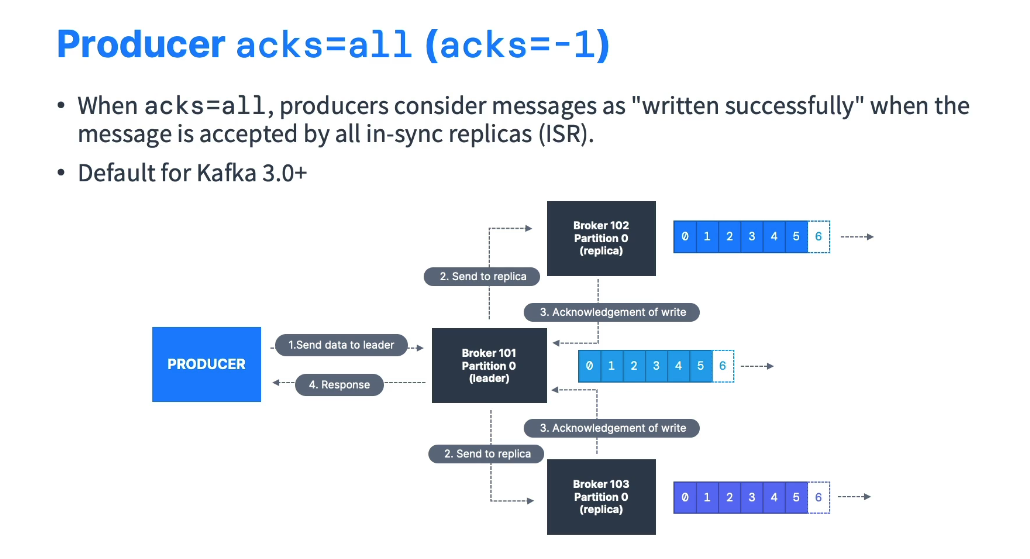
Producers can send to either topic level or partition level. For topic level, round robin fashion will be used meaning that first message will go to partition 1 then next message will go to partition 2 and so on. Sending message to the partition level means that we’ll have a key in our message that will tell to which specific partition we want to write data.

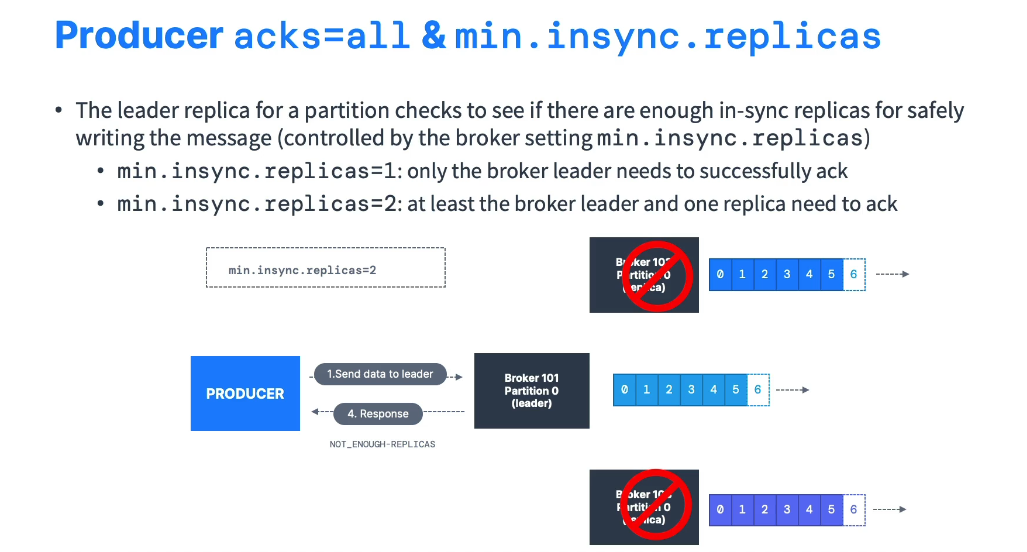


It means if key is not null then all messages for that key will always go to the same partition using hashing. A key is used if you need message ordering for a specific field. for example, in the truck\_gps example as we discussed above, we’ll send ID of that truck with the messages so that the position of the specific truck will go to the specific partition.

Producers can choose to receiver the data acknowledgements of data write:

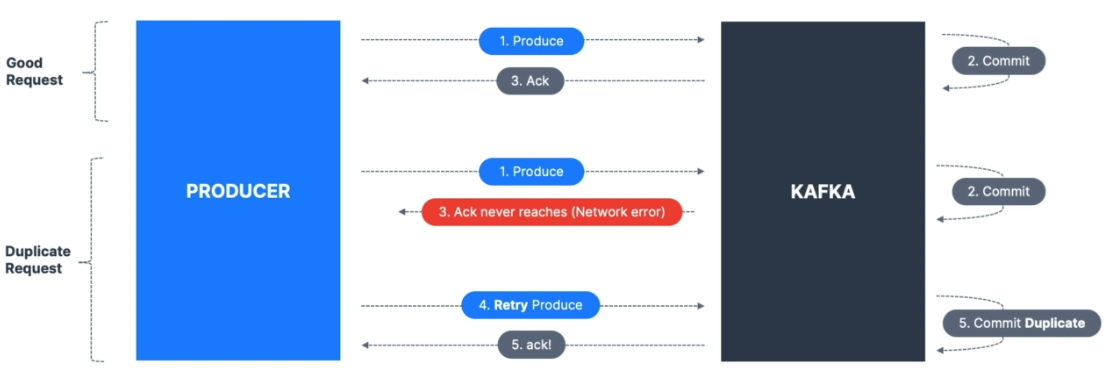
* Acks=0 means that no acknowledgement will be sent (Possible Data Loss)
* Acks=1 means that producers will wait for leader’s acknowledgment (Limited Data Loss)
* Acks=all means that producers will wait for leader as well as replica’s acknowledgement (No data loss)

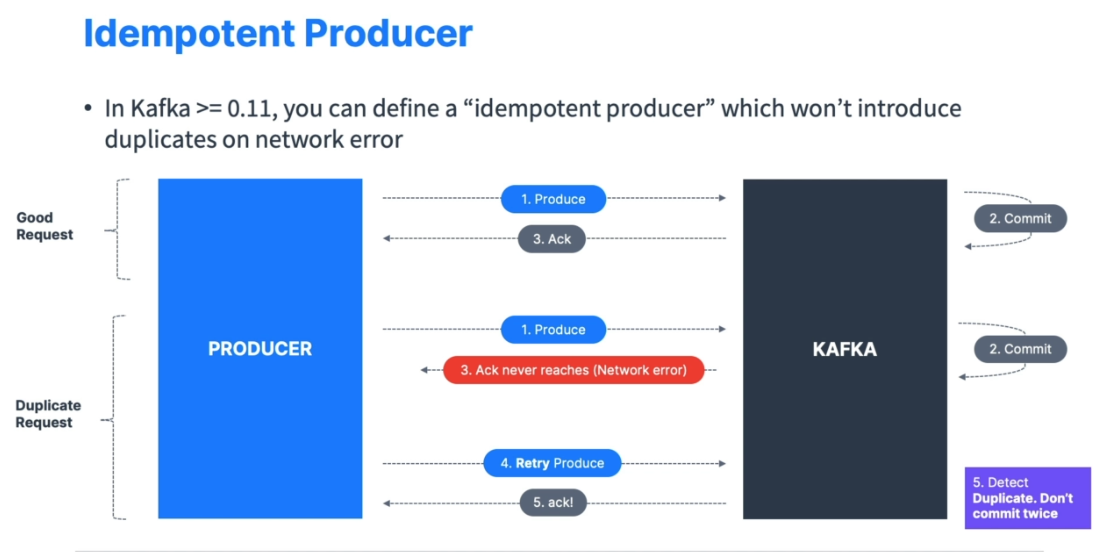


For more security we use min.insync.replicas=2. In summary when acks=all with a replication. Factor=N and min.insync.replicas=M then we can tolerate N-M brokers to get down. 

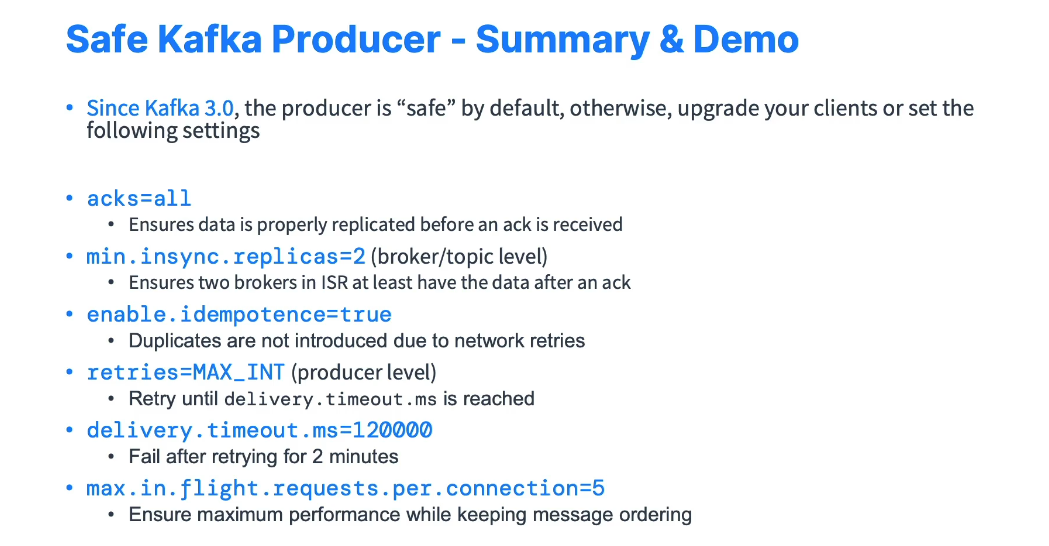
**Idempotent Producer:**

The producer can introduce duplicate messages in kafka due to network errors. To solve this issue we can use idempotent producer. These producers ensure stable and safe pipeline. Default for kafka>=3.0. If not manually set using producerprops.put(“enable.idempotence”,true)

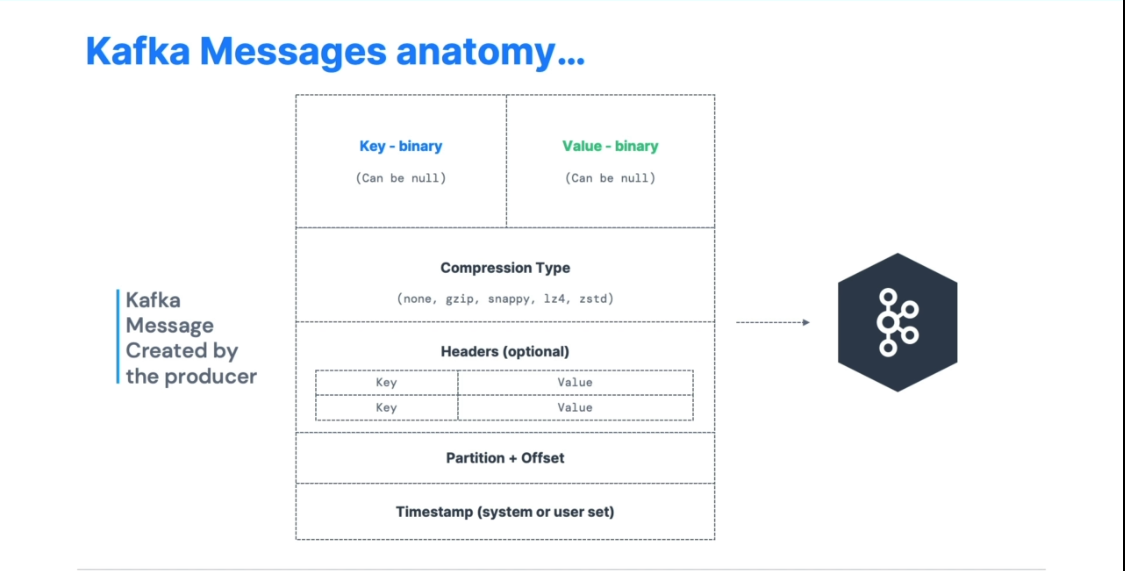




**What is the best and safe properties for a Producer:**



**Kafka Message Anatomy:**

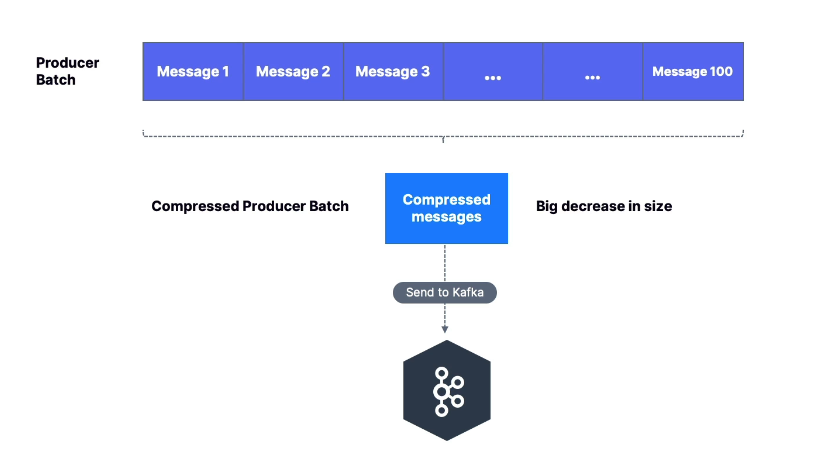


Kafka only accepts bytes as an input from the producers and also outputs bytes to the consumer, so we need to serialize/deserialize the key and the value to convert it into bytes in order to complete some tasks.



**Message Compression**:

We must apply compression before sending the messages. It has many advantages we’ll see these advantages in a bit. Compression.type can be none(default), gzip, lz4, snappy, zstd. Compression is effective when the batch size is very huge. This message compression can occur at Producer level, Topic/Broker level(Default Compression.type=producer). If the compression type of broker/topic contradicts with that of producer then the message will be first decompressed and then compressed according to the type of broker/topic.



**Advantages of Message Compression**:

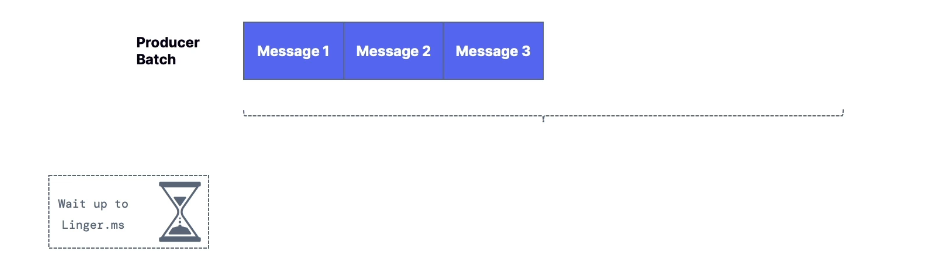
* Less Latency
* Better throughput
* Efficient disk utilization in kafka

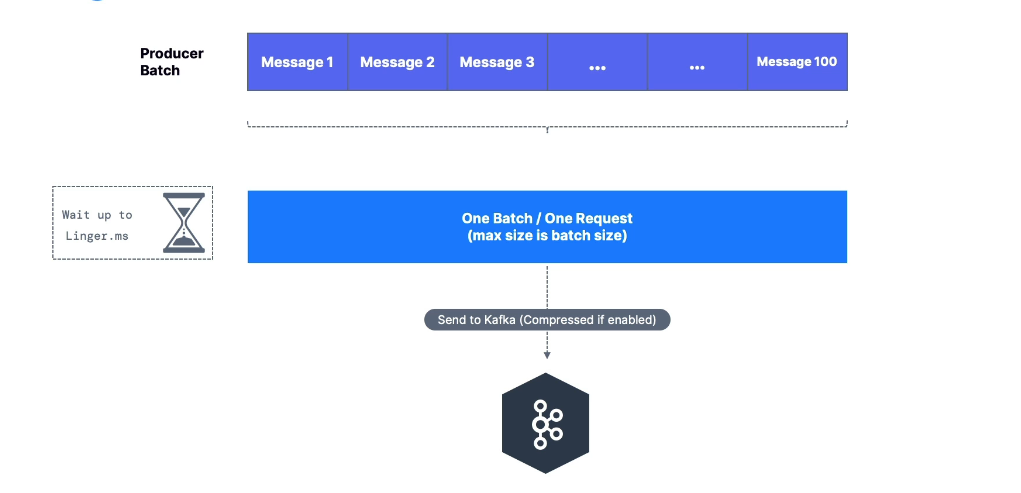
**Linger.ms and Batch Size:**

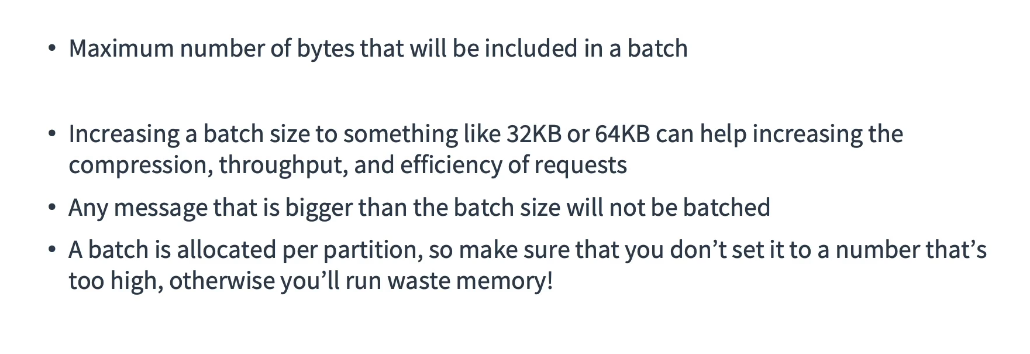
max.in.flight.requests.per.connection=5 means that up to 5 messages batches being in flight can be send at most. Two settings influence the batch mechanism:

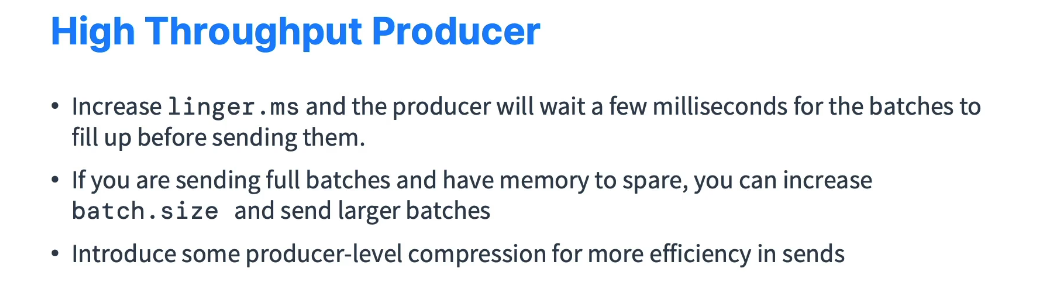
**1)Linger.ms** is how long to wait until we send a batch. Adding a small number for example 5ms help add more messages in the batch at the expense of latency.

**2)Batch.size**: if a batch is filled before linger.ms, increase the batch size.



After waiting up to linger.ms there are 100 messages in a batch that can be out batch size. Then we’ll send it to kafka. 





**Consumers:**

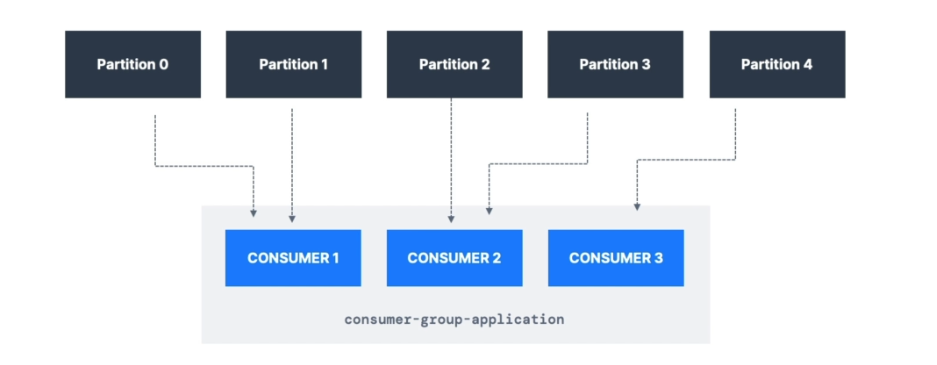
Consumers are the application that read/consumes data from the topic within a cluster using some Consuming API’S.

Kafka is a pull model means that producers does not push data to the consumers rather consumers request for the data they are interested in and then pull it.

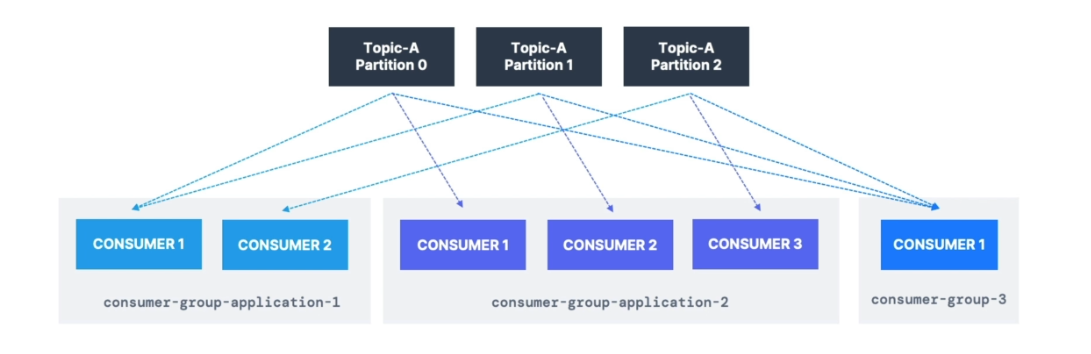
Data is read in order from low offset to high offset within a partition.

Consumers can consume data either on a topic level means from all partition of the topic or the partition level means that from specific partitions of the topic.

**Consumer Groups** is a group of related consumers that perform a task. Consumers are always associated with exactly one consumer group.



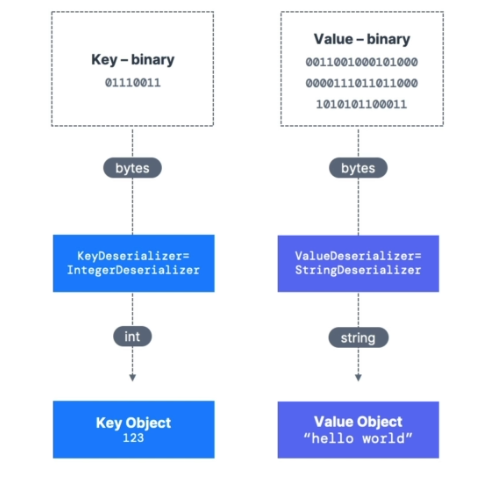
If we have more consumers than no of partition, then it means that some consumers will be inactive. To create distinct consumer groups, we’ll use consumer property group.id. Moreover, we can have multiple consumer group consuming data from the same topic as shown in figure:

**Consumer Offset** Kafka stores the data at which a consumer has been reading. The offsets committed are in kafka topic named \_\_consumer\_offsets. Consumer will commit the offset at which it done processing/reading. So, it will help Broker to know until which offset, we done reading the messages. So, if a consumer dies it will be able to read back from where it left off thanks to the committed consumer offsets.



The above diagram shows that we have offset from 4258 to 4269. As the consumer reads from the offset it commits that offset that helps in fault tolerance.

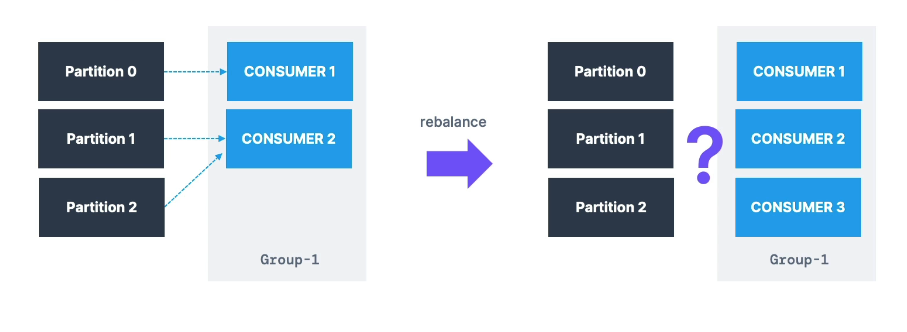
We need to deserialize the data from bytes into some specific datatype that is required by our consumer application



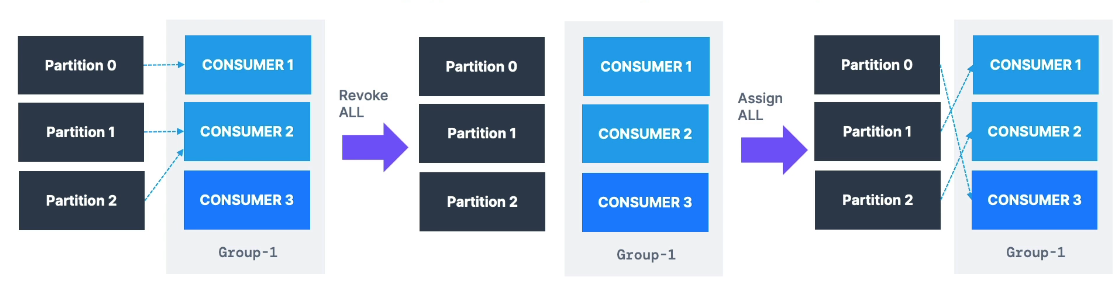
**Consumer Group and Rebalancing:**

**(How to use partition.assignment.strategy=RangeAssignor/CooperativeStickyAssignor)**

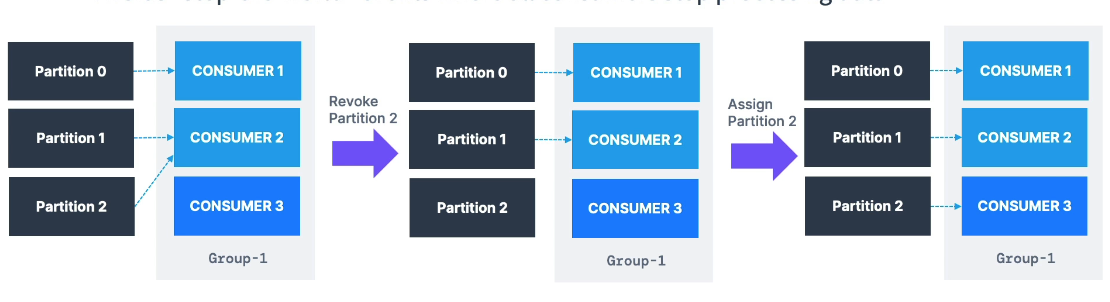
Moving Partition b/w consumers is called rebalancing. Reassignment happens when a customer leaves or joins a group. It can also happen when new partitions are added by administrators.

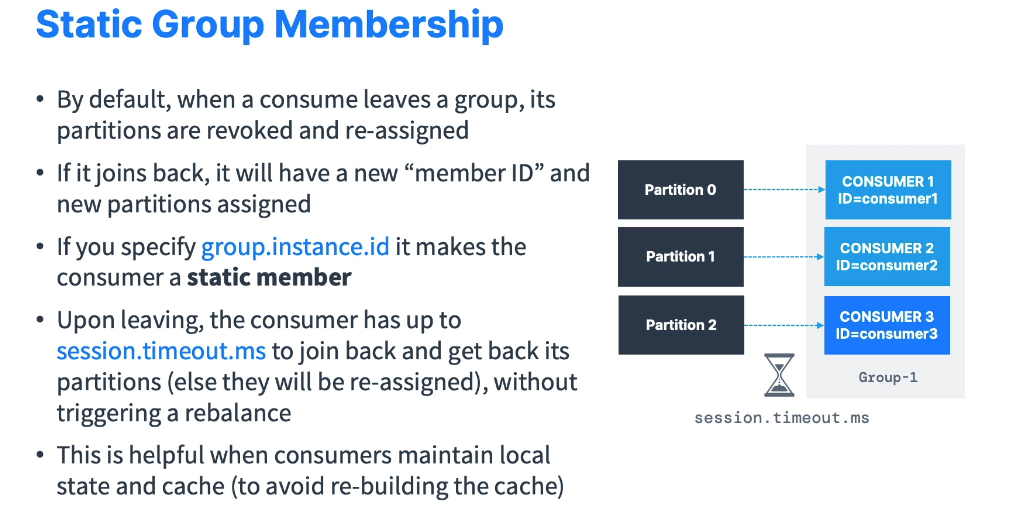


1)**Eager Rebalance**: All members stop, give up their membership of partitions. They rejoin the consumer group and get a new partition assignment. **Disadvantages** of this type of rebalancing is that all consumer stops working until reassigned and there is no certainty that consumer will get the same partition as it used to**.** This is the default behavior.



2)**Cooperative Rebalance**: Reassigning a small subset of partitions from one consumer to another. Other consumers that don’t have reassigned partitions can still process uninterrupted. Can go through several iterations to find a stable assignment hence “incremental”. In given example new consumer Consumer3 added and cooperative rebalance take place.





**Brokers:**

Brokers are simple software processes who maintain and manage the published messages. A kafka cluster is made up of many brokers(servers). Each broker contains multiple topics and partitions. After connecting to any broker (also known as bootstrap broker), we will be connected to the entire cluster.

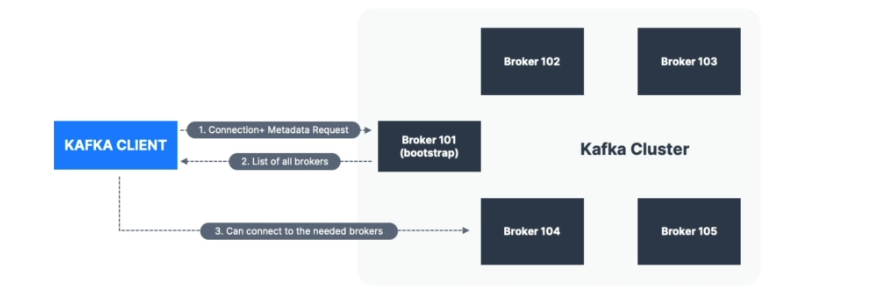
Each Broker is identified by its id. Brokers also manage the consumer offsets so they are responsible for the delivery of messages to the right customer.

Example of Topic A with 3 partition and Topic B with 2 partitions:



We can see that our data is distributed and more broker we add more spread of data will be generated within a cluster. This is a concept of horizontal Scalability.

Every Kafka server is a bootstrap server and we only need to connect to only one broker. The connection with rest of the brokers will be done automatically as each broker knows about all the brokers, topics and partitions.



**Zookeeper:**

Zookeeper is the software used to monitor kafka cluster and coordinate with each broker. It keeps all the metadata in the form of key value pair.

Metadata includes:

* Configuration Information
* Health Status of each broker

Zookeeper helps in electing the leader of the partition. It sends notification to kafka in case of any change (new topic created, broker dies). Zookeeper by design have odd number of zookeepers for example u can either have 1 zookeeper or 3 and so on.

As we know, kafka is transitioning to no zookeeper, so should we use zookeeper? Answer is yes we have to use zookeeper to make it ready for production but don’t use zookeeper for the configuration of clients only use it for the broker’s configuration.

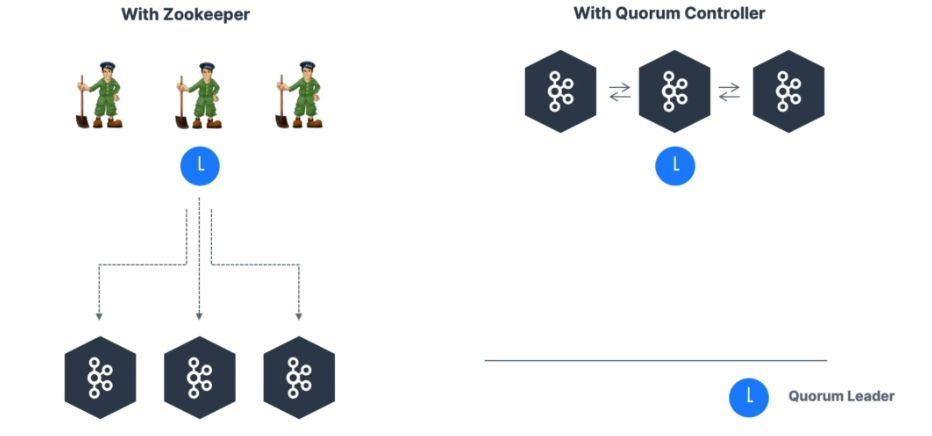
**Kafka kraft:**

We can use kraft protocol to use kafka without the help of zookeeper. But why we are trying to get rid of zookeeper dependency on kafka?

There are following reasons:

* Zookeeper shows scaling issues when number of partitions exceeds 100,000.
* Zookeeper is less secure than kafka
* It is hard to monitor and maintain the zookeeper together with the kafka.

If we get rid of zookeeper we can have greater performance, faster controller shutdown and recovery time. Single security model for the whole system. Single process to start kafka.



Above figure shows the difference of using kafka with zookeeper and without zookeeper. In left side 3 zookeeper nodes are trying to manage 3 brokers and one of the zookeeper nodes is the leader zookeeper. However, in the righthand side where we are using kraft protocol, we have 3 brokers and one of them is selected as the quorum leader that is managing the other brokers.

**Commands**

**Start Kafka with Zookeeper:**

~/kafka\_2.13-3.3.1/bin/zookeeper-server-start.sh ~/kafka\_2.13-3.3.1/config/zookeeper.properties **(Starting Zookeeper)**

~/kafka\_2.13-3.3.1/bin/kafka-server-start.sh ~/kafka\_2.13-3.3.1/config/server.properties **(Starting Kafka Server)**

**Starting Kafka Without Zookeeper (KRAFT Mode):**

~/kafka\_2.13-3.3.1/bin/kafka-storage.sh random-uuid **(Generating Random Id)**

~/kafka\_2.13-3.3.1/bin/kafka-storage.sh format -t <uuid> -c ~/kafka\_2.13-3.3.1/config/kraft/server.properties **(Formatting)**

~/kafka\_2.13-3.3.1/bin/kafka-server-start.sh ~/kafka\_2.13-3.3.1/config/kraft/server.properties **(Starting Kafka Server)**

**Topics:**

kafka-topics.sh --bootstrap-server localhost:9092 --list (**list all topics**)

kafka-topics.sh --bootstrap-server localhost:9092 --create --topic first\_topic **(Create a new Topic with default configuration (1 partition and 1 replication factor))**

kafka-topics.sh --bootstrap-server localhost:9092 --create --topic second\_topic --partitions 3 --replication-factor 1 **(Create a new customized Topic U can’t have replication factor>1 if u have only 1 broker)**

kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic <topic name> **(to describe particular topic if u want all topics, then simply write up to --describe)**

kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic <topic name> **(to delete a specific topic)**

**Producers:**

kafka-console-producer.sh --bootstrap-server localhost:9092 --topic <topic name> **(produce data on a specific topic)**

kafka-console-producer.sh --bootstrap-server localhost:9092 --topic second\_topic --producer-property acks=all **(set Properties of producer)**

**=>If we produce data on the topic that doesn’t exist then kafka will create that topic on its own with default topic properties**

kafka-console-producer.sh --bootstrap-server localhost:9092 --topic second\_topic --property parse.key=true --property key.separator=: **(Produce Data with Keys)**

**Consumers:**

kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic check\_topic1 --from-beginning **(Read the whole topic from the beginning)**

kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic check\_topic1 **(read from tail i.e., only new messages that u will produce will be appear)**

kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic <topic name> --group <group name> **(Make a consumer that relates to some specific consumer group)**

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group <group name>--reset-offsets --to-earliest --execute --topic <topic name> **(Reset the offset so that we can read from the beginning)**

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list **(List all the consumer groups)**

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group <group name> --describe **(Describe a particular consumer group)**

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group <group name>--reset-offsets --shift-by 2 --execute --topic <topic name> **(Shift the Consumer offset by some number either positive or negative)**