

At the heart of Kafka, we have kafka cluster. Kafka cluster generally consist of multiple brokers. In order to manage multiple brokers, we need a zookeeper. Basically, zookeeper keeps track of the health of the brokers and manage the cluster for you.

Broker is what kafka clients will interact with. Having just a cluster doesn’t add any value.

The very first client is Kafka producers. Kafka producer is the way to write and produce new data into kafka. Clients uses the producer API to write data into the kafka. After the data written into kafka, what needs to be done ? we need to consume the data from kafka. So kafka consumers comes in handy to consume the messages. They use the consumer API to consume the message from the kafka cluster.

Kafka consumer and producers are basic client API’s through which we can interact with kafka. There are 2 advanced client api’s that comes with kafka.

1. Kafka Connect :- It has 2 different types of connectors ( source connector & sink connector ). Source connector is used to pull the data from an external data source such as database, file system or an elastic search into the kafka topic and opposite of the same is done using the sink connector.

With Kafka connect, you can perform the data movement in and out of Kafka without writing a single line of code.

1. Stream API :- which is used to basically to take the data from Kafka and perform simple to complex transformation on it and put it back to Kafka.

To summarize, you have 4 client API’s such as the Producer API, Consumer API, Connect API and Streams API, using which you can interact with Apache Kafka.

Understand Kafka Components and its internal :-

1. Kafka Topics & Partitions :-
2. Kafka Topics :-

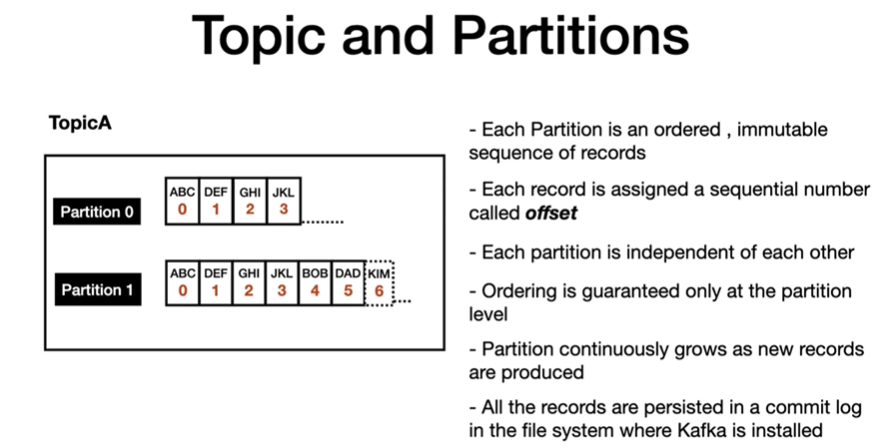


Kafka Topic is an entity in Kafka and it has a name. A quick analogy is to think of topic as a table in a database. Topics in general live inside the Kafka broker.

In this ex, we have Topic A inside the broker. Kafka clients uses the topic name to produce and consume messages. The behaviour of Kafka consumer is to poll continuously for new messages. So here, consumer is polling the broker using the topic name. Producer in general produces a message into the topic if something outside invokes a producer. But even here, the producer uses the topic name to produce the message. So when the message sent from the producer, it reaches the Kafka topic first, once a poll notices this message, then it is consumed by the consumer and the consumer does some processing on that record, thus retrieved.

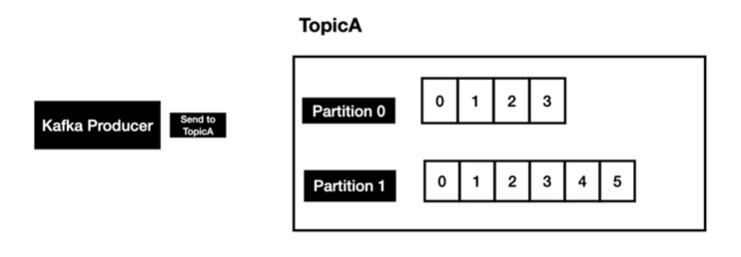
Note : even though the record is read by the consumer, the message still resides inside the Kafka as per the defined retention time.

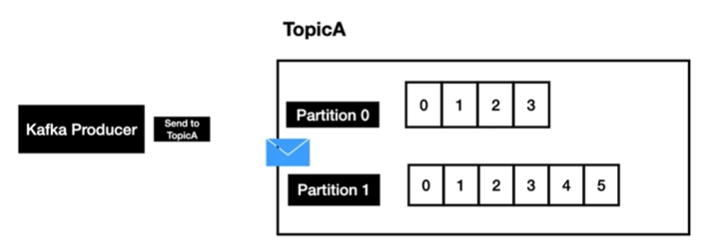
Partitions :- is where actually the message is located inside the topic. Each topic in general can have one or more partitions. It is pretty common practice to have more than one partition. I have seen scenarios where topic had 100 partitions or more. The partition has a significant effect on scalable message consumption.

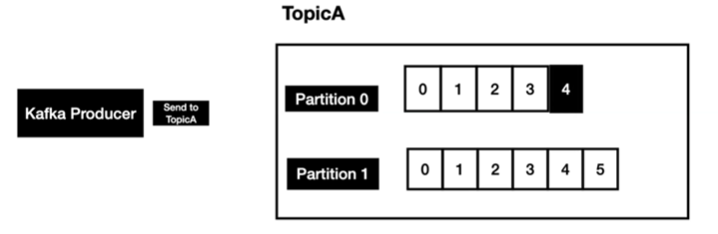


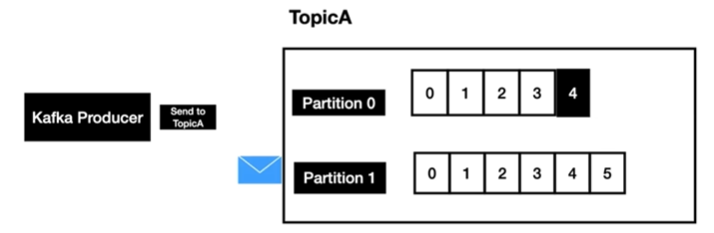
So if you have a use case where you would like to publish and read the records in a certain order, then you have to make sure to publish the record to the same partition.

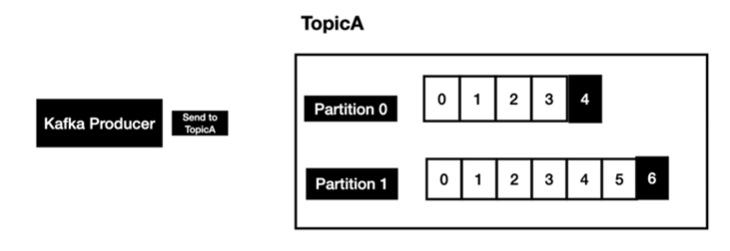
All records are persisted in a physical log file where the Kafka is installed. It is similar to the commit log file that you find with DB transactions, but this one is the distributed log file.







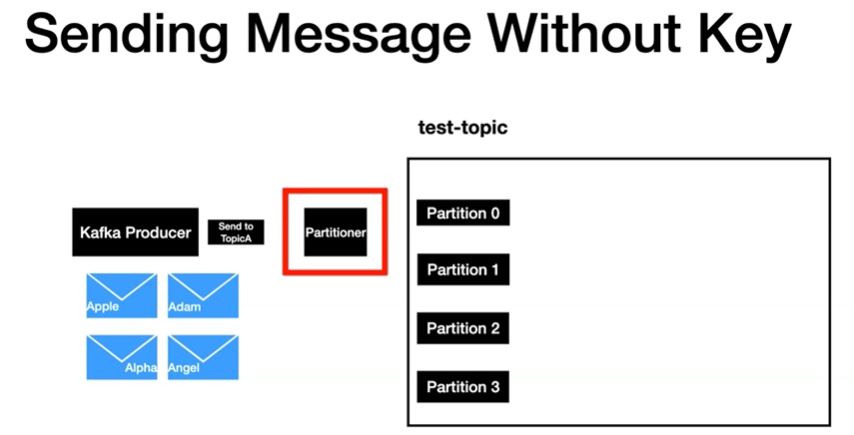




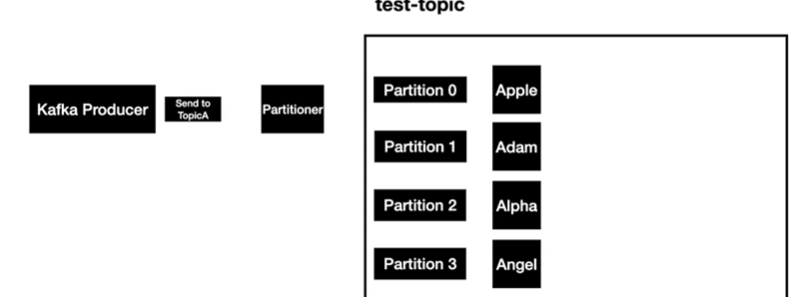
And it continuous to do the same as new records are produced into the Kafka topics. Producer has complete control of which partition the message is going to go. We will cover different strategy when we code the Kafka producer.



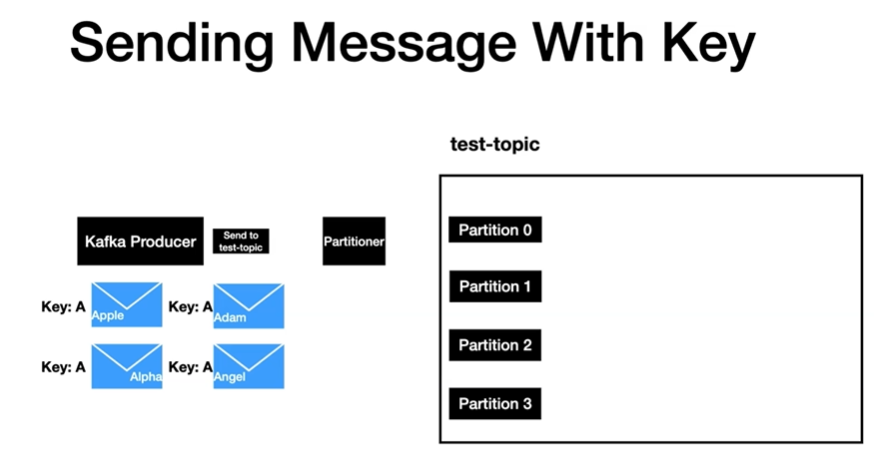
We are going to spin up a zookeeper instance. Zookeeper comes with Kafka distribution that we download. After that, we are going to spin up Kafka broker, once a broker is up , it register itself with the Zookeeper. After that zookeeper manages and monitors the health of the Kafka broker. Zookeeper plays a vital role when you have multiple brokers.



Partitioner first checks whether a key is present as part of message or not. In this ex: we are not sending any key , so in this case partitioner will use the round robin approach to send the message across all partitions.

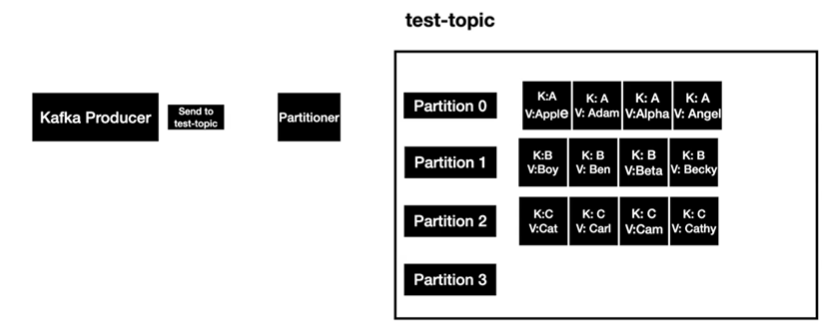


In this approach, there is not guarantee that the consumer will be able to read the messages in the same order. Because consumer polls the messages from all the partitions at the same time.

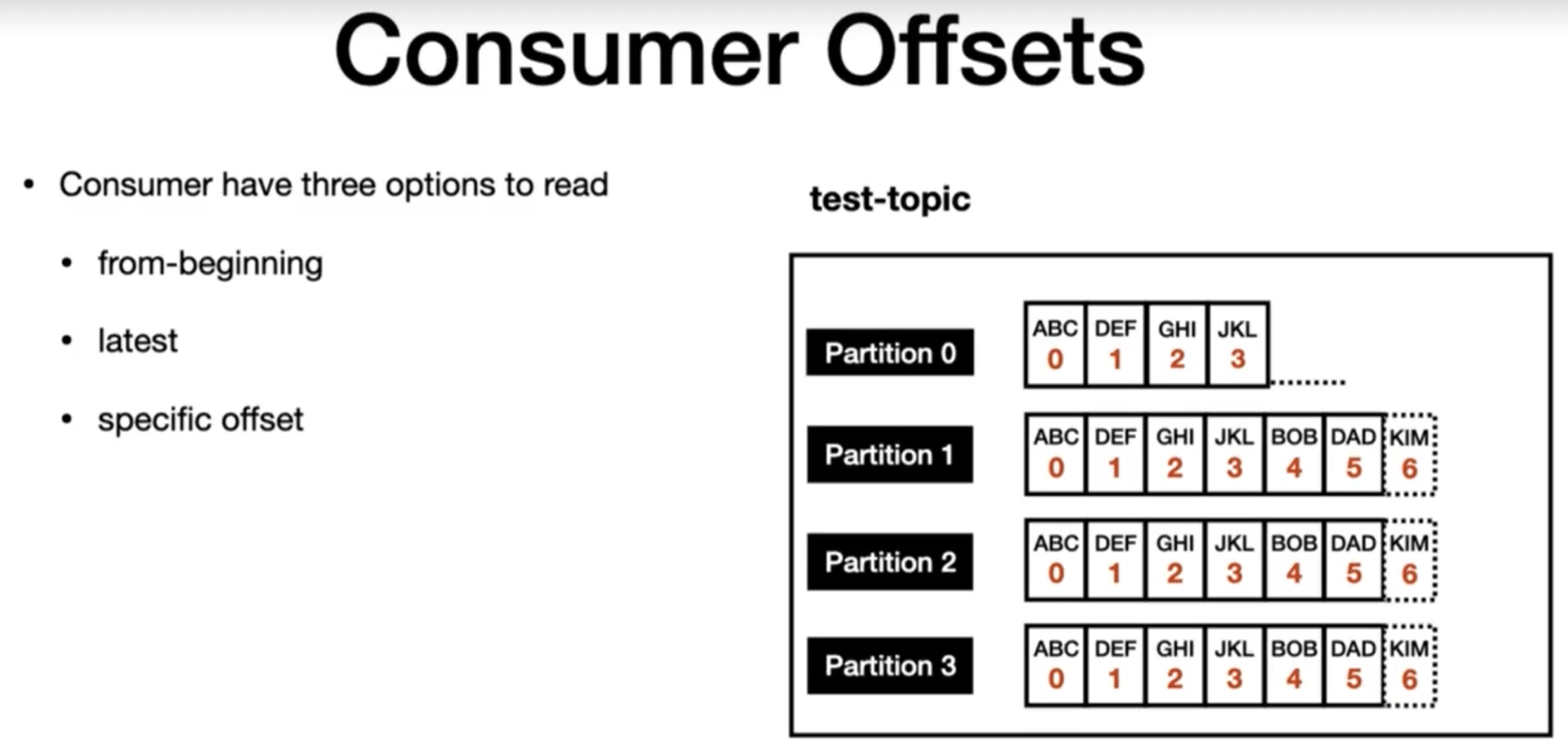


Here all the messages has key A. The key can be of any type. In this ex, we are using string. When a message is sent, Kafka partitioner, which is part of Kafka producer is going to applies some hashing technique to determine partition value.

So here, we have the key ‘A’, and its going to apply some partitioning technique meaning the hashing technique, it is going to resolve it to partition 0. If the same key is sent again in another message, it will resolve it to same partition 0.



But in reality, Key A can go to the one of the available partitions of the topic based on the hashing technique. But one thing to keep in our mind that same key always resolve to the same partition.



Any message that produced into the topic will have a unique ID called offset. Consumers have 3 options when it comes to read messages from the topic.

1. They can read the messages from the beginning using the “from-beginning” option.
2. Latest – meaning read only the messages that’s going to come after the consumers spun up.
3. Specific offset – meaning read the messages in the topic by passing a specific offset value from the consumer. This option can only be done programmatically.



For any Kafka consumer, it is required for the consumer to provide the group id. Now the consumer in general polls and retrieves multiple record at the same time. As it process each message, it moves the consumer read offset one by one.

Lets say for some reason, the consumer is crashed, while the consumer was down, the producer of the topic, produce some more messages. Now the consumer is brought up after some time, How does it know that it needs to read from offset 4.

The consumer offset in general are stored in an internal topic called \_consumer\_offsets.

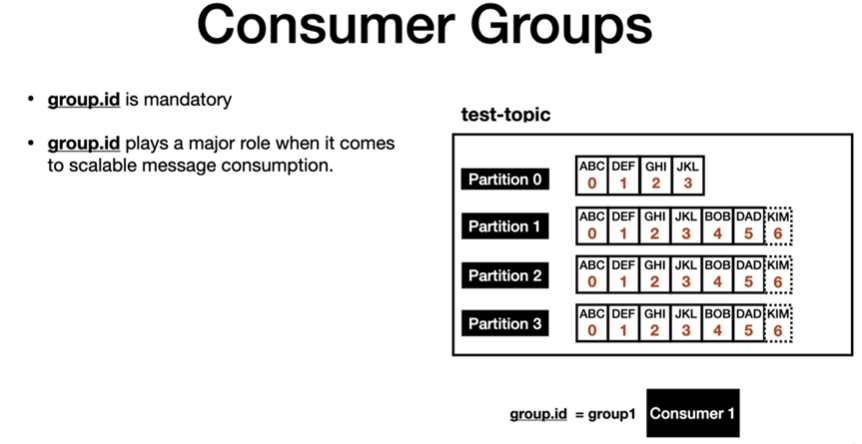


Now the consumer knows where to start reading the messages in the topic using the information that’s available in the consumer offset topic with the group id.

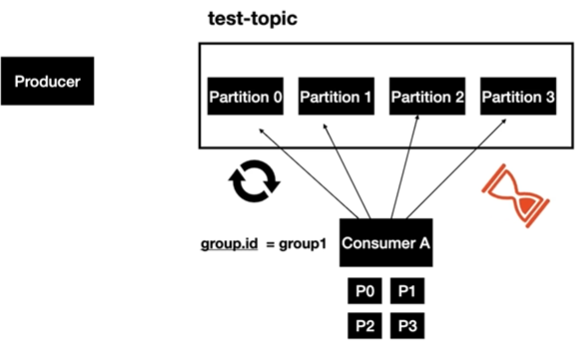
In a nutshell, the consumer offsets behave like a bookmark for the consumer to go and check from which point in the topic it needs to read the messages from.



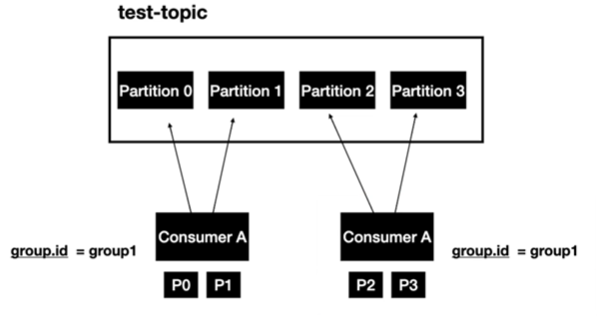
So this is the topic which was auto-created by the broker which takes care of maintaining the consumer offsets for you.



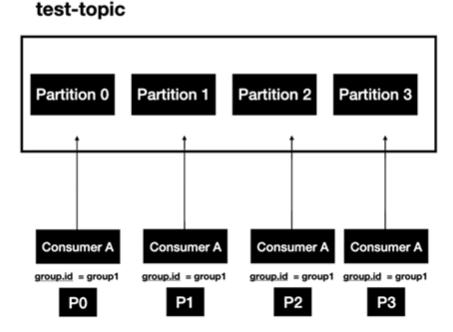
For eg :- We have one topic “test-topic” and it has 4 partitions. Now, we have consumer A with the group id “group1”. We have one single consumer polling all the 4 partitions in the topic and processing them. The Poll is always single threaded so in this case, a single thread is going to poll from all the partitions.



Lets say producer of the topic is going to produce messages at a faster rate than the consumer processing rate, Then in this case it will introduce a lag in the consumer and you might end up not processing the events in real time. This is where consumer group comes in handy.

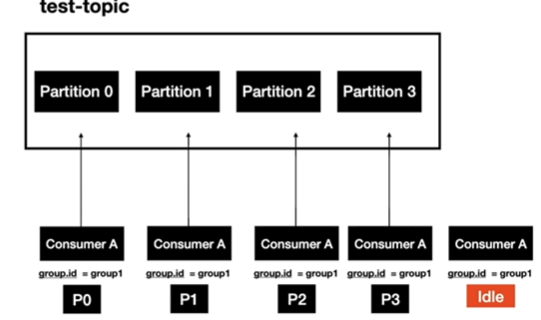


Now lets say we spin up another instance of consumer A but make sure you are using the same group id. Now the partitions are split between the 2 instances of the consumer. Partition 0 and partition 1 is taken care by the first instance and partition 2 & 3 are taken care by the 2nd instance. So it means, we scaled our message consumption. This will help process the record little faster than it was before.



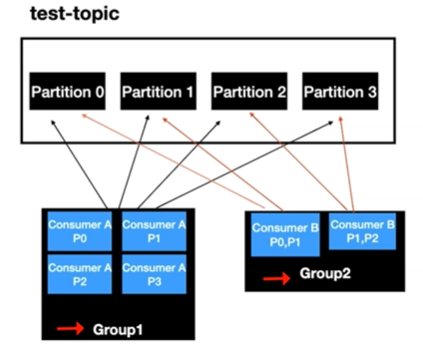
Lets make it much better by spinning up 2 more instances. So we have 4 instances of consumer A application but group id is same across all the different instances that we have. At any given point, you are going to process 4 records parallelly.

So the consumer groups are fundamentally basis of scalable message consumption.



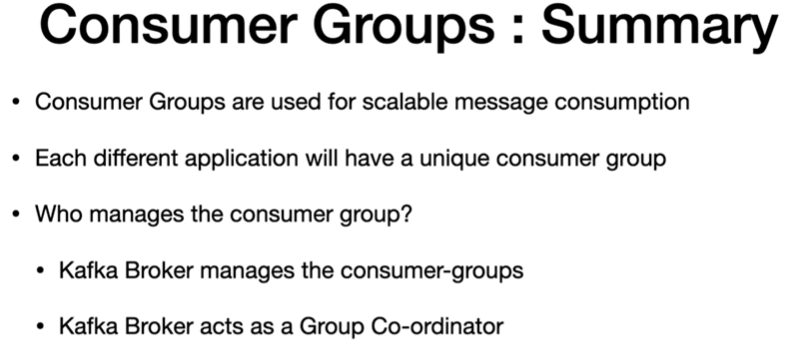
Now let’s say you have 5 consumer instances but only 4 partitions are available for a given topic. In that case, one of the instances will be Idle. As we have mentioned before, the consumer poll is single threaded. So, in this case, this will lead to an efficient use of resources.

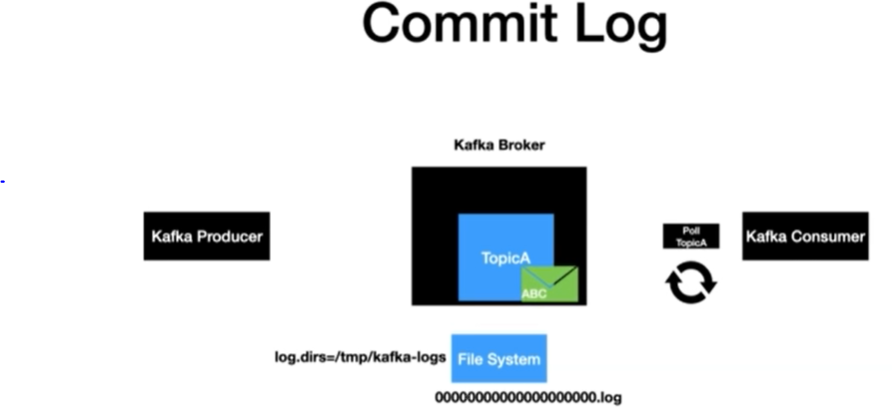
Lets highlight the second common usage of consumer groups.



Let’s say you have 2 different applications for the same Kafka topic. This is a pretty common scenario in an enterprise. Each consumer app will have their own processing logic. Each app can have different number of instances based on the requirement.

In this ex, we have 4 instances of the consumer A application with a group id 1 and 2 instances of the consumer B app with the group id 2. But the teams have to make sure they are not using the same group Id. That’s the reason why you see the group id is unique between the application. So fact is that it is up to the team to decide on how many number of instances they want.





When producer send a message it first reaches the topic and then the very next thing that happens is that the record gets written to a file system in the machine. So the file system is where Kafka broker is installed. In this ex, its our local machine.

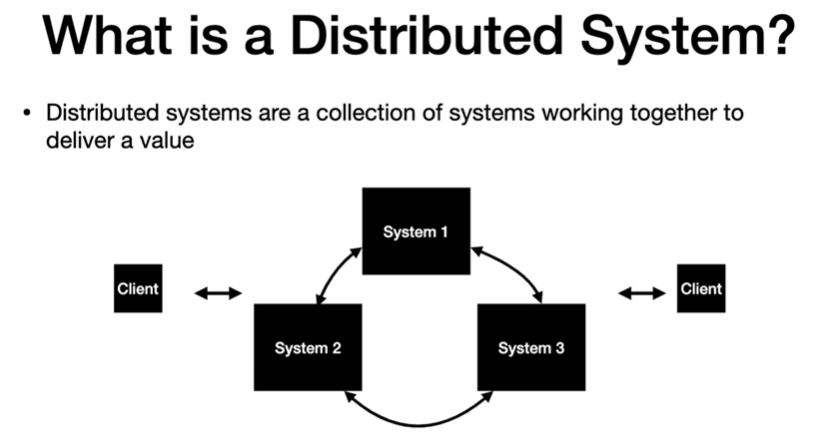
The record is always written in to the file system as bytes, the file system where that file needs to written is configured using the “log.dirs” property. The property is available in the server.properties file. It creates a file with the extension of “.log”. As you can see, we have numerous zeros followed by that .log. Each partition will have its own log actually meaning if we have 4 partitions then you will have 4 log files in the file system. That’s why these log files are otherwise called partition commit log.

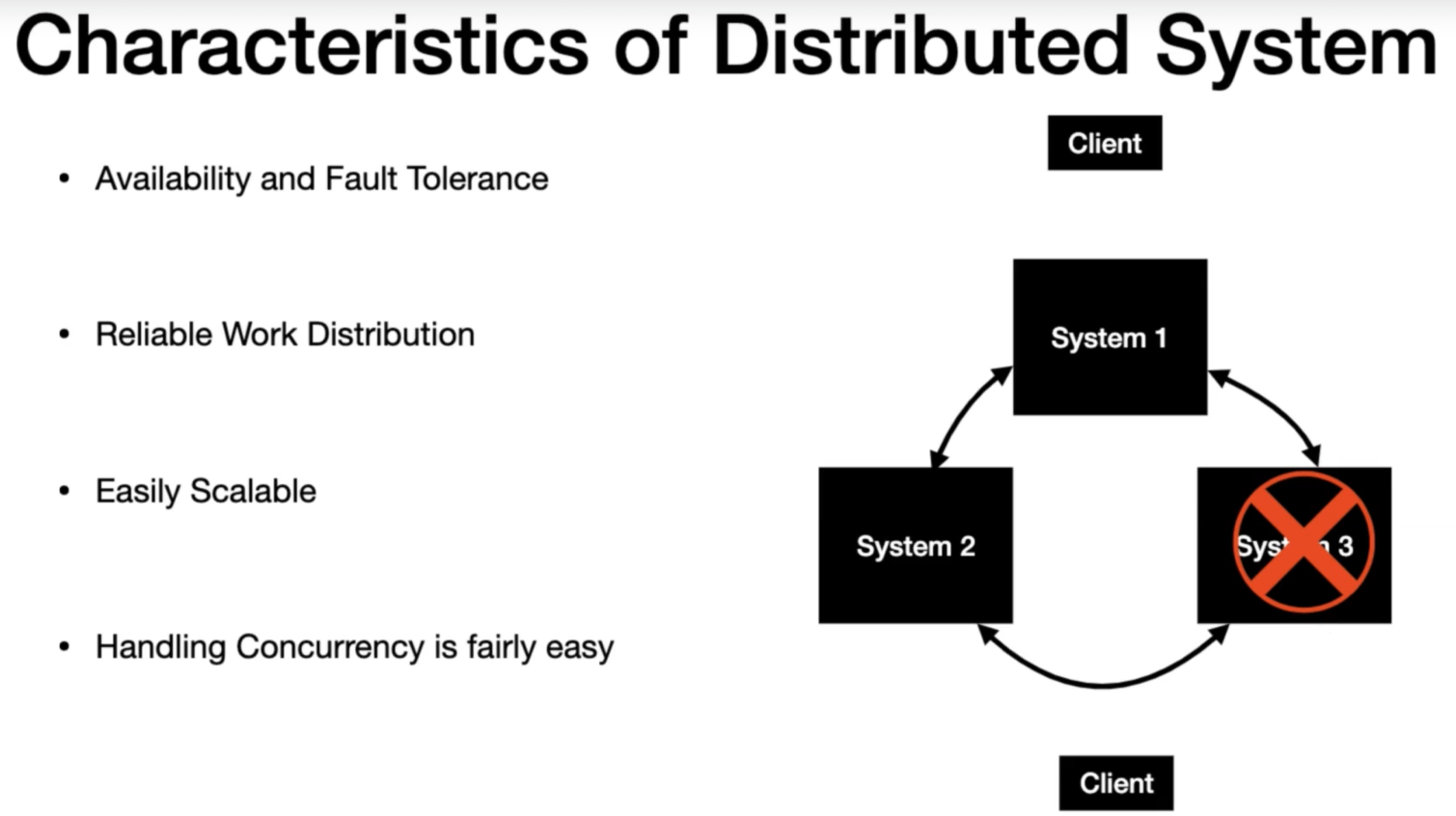
After the messages are written into the log file, that’s when the records that got produced are committed. So, when the consumer who is continuously pulling for new records can only see the records that are committed to the file system. As new records are produced to the topic, then records get appended to the log file and the process continues.

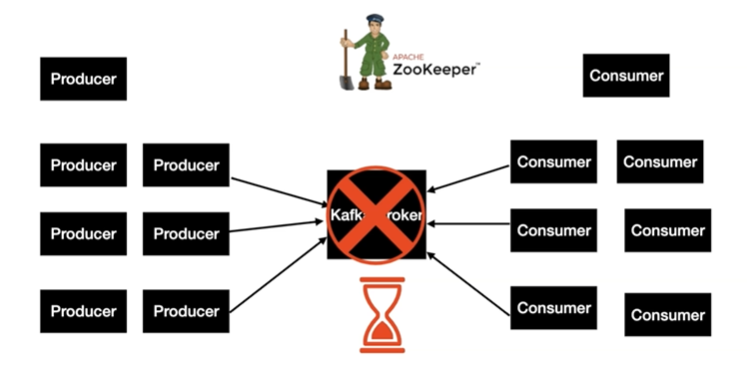
So, this is all about commit log.



Kafka as Distribution Streaming System -> Kafka is a distributed streaming platform.

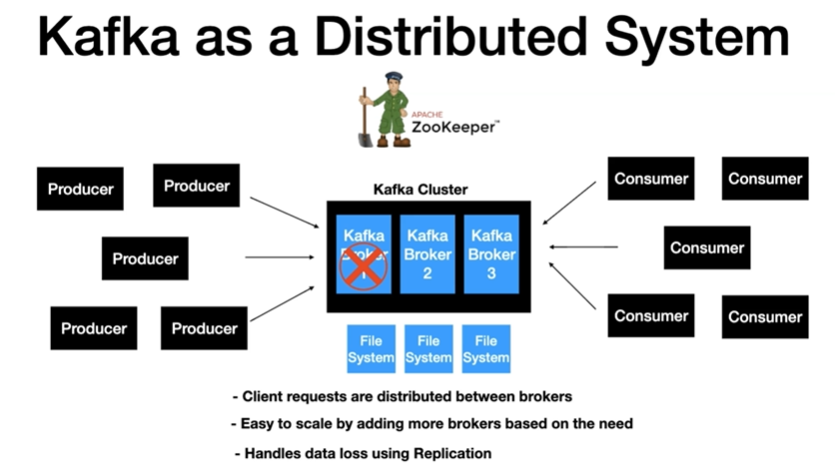






In any enterprise, it is pretty common to have bunch of producers and consumers. With one broker in place, lets talk through some of the key behaviour.

All the producer & consumer requests will go to the same broker. There is a very big possibility the current setup will get overwhelmed with bunch of requests pretty faster and it might crash the system. This leads to single point of failure.



In this example we have 3 brokers and cluster will be managed by zookeeper . All the brokers send heartbeats to the zookeeper at regular intervals to ensure that the state of the Kafka broker is healthy and active to serve client request.

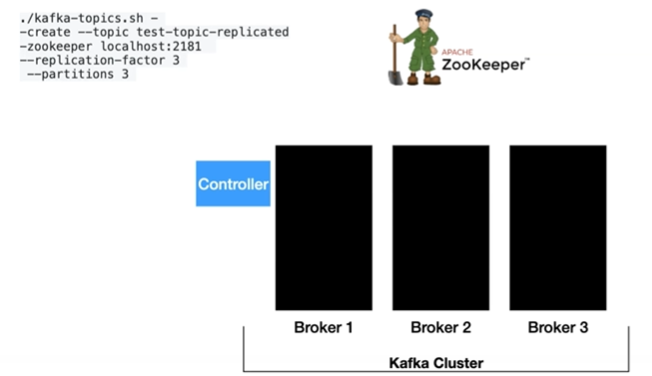
With 3 brokers in place, the client requests are evenly distributed between them and it handles the load very well.

If one of the broker goes down then the cluster manager which is the zookeeper here gets notified then all the client requests will be routed to the other available brokers. Clients won’t have any clue that an issue is going on.

Setting up Kafka in Local with 3 Kafka Brokers ->

How Kafka Cluster distributes the Client Requests between the brokers? – Leader/Follower

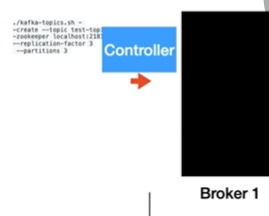
Lets first learn about how Kafka topic distributed across the available brokers. At first we have zookeeper and a Kafka cluster. In this ex, we have a cluster with 3 brokers. Out of the 3 brokers, one of the available broker will behave as a Controller. Normally, this is first broker to join the cluster. Think of this as one additional role for the broker. Now we have environment completely setup.



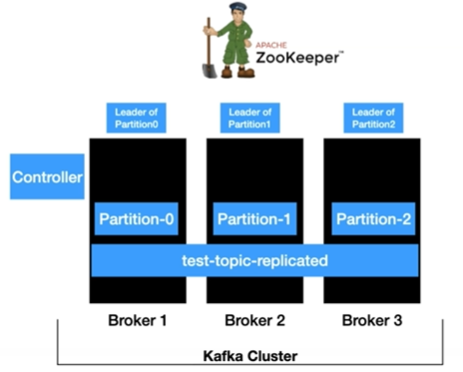
Now its time to create the topic, create topic command is issued to the zookeeper.



Zookeeper takes care of redirecting this request to the controller. The role of this controller is to distribute the ownership of the partition to the available broker.



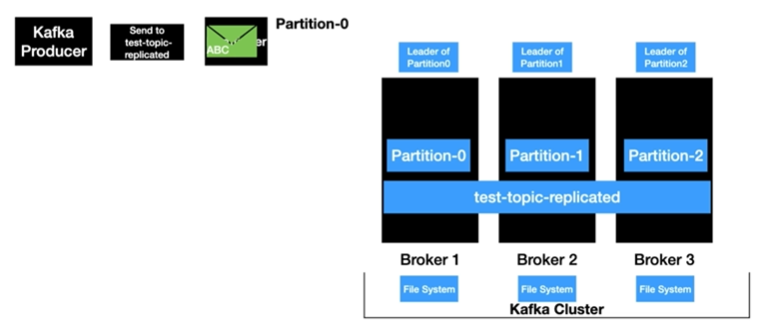
So in this ex, we have partition 0 sitting in broker 1, partition 1 sitting in broker 2 and partition 2 sitting in broker 3 . In distributed system, This concept of distributing partitions to the broker is called a leader assignment.



So in nutshell, the topic is distributed across the Kafka cluster.

Now its time to learn about how does the cluster handles the connection from the clients.

So producer has a layer called partitioner which takes care of determining which partition the message is going to go. So the producer sends the first message, it goes to the partitioner before the messages send to the Kafka topic.



The partitioner determines this message should go the partition-0. In this case, the leader of partition-0 is Broker 1, so the message will be sent to Broker 1. The client will always invoke the leader of the partition. After that message is persisted into the file system of Broker 1.

ABC - > Partition 0

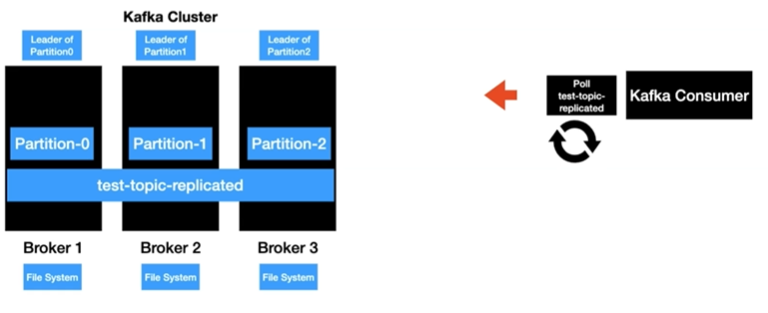
The process is repeated for the following messages.

DEF -> Partition 1

GHI -> Partition 2

As you can see, the client requests from the producer end are distributed between the broker based on the partition which indirectly means that the load is distributed between the broker.

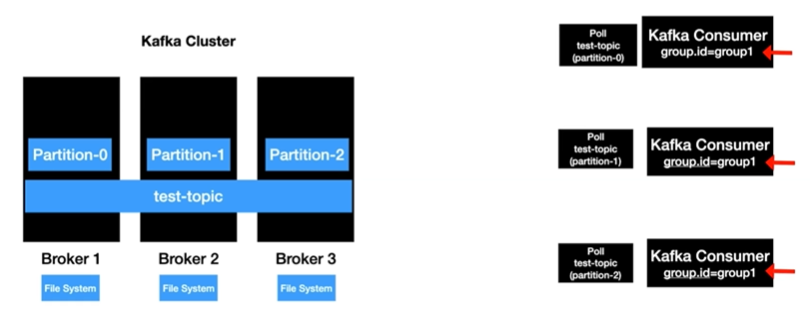
Now it’s time to look at how the client requests are distributed from the Kafka consumer end. We have consumer ready to poll this test-topic-replicated topic. When the poll loop is executed, the request goes to all the partitions and retrieve the records from them.



In here, each broker owns the partition. In this case, the poll request goes to all the brokers and retrieves the record from them and the retrieve records are handed over to the Kafka consumer and the consumer process the record successfully and the same flow repeats.

So, in nutshell, even from the consumer end the request to retrieve the data are distributed between the brokers. Basically, the client call will only go to the partition leader of the topic and retrieve the data.

Now let’s take a look at the Kafka consumer flow with consumer groups ->

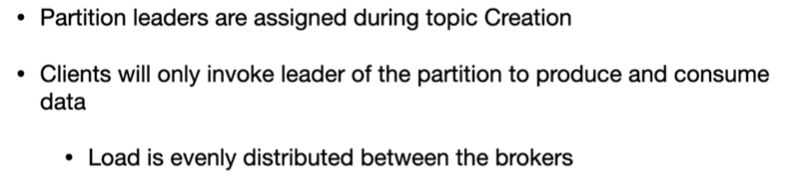


Because it is common practice to run multiple instances of the consumer and process the records from the Kafka topic.

So in here we have 3 instances of the consumer with the same group ID. If you can’t recall from the concepts of consumer groups if there are one or more instances of the consumer with the same group ID, then partitions are distributed for scalable message consumption. So, in here, each consumer instance has one partition assigned. When the poll gets executed, each instance is going to poll the data from the partition that they are interested in and poll call goes to the leader of the partition of the topic and retrieves the data.

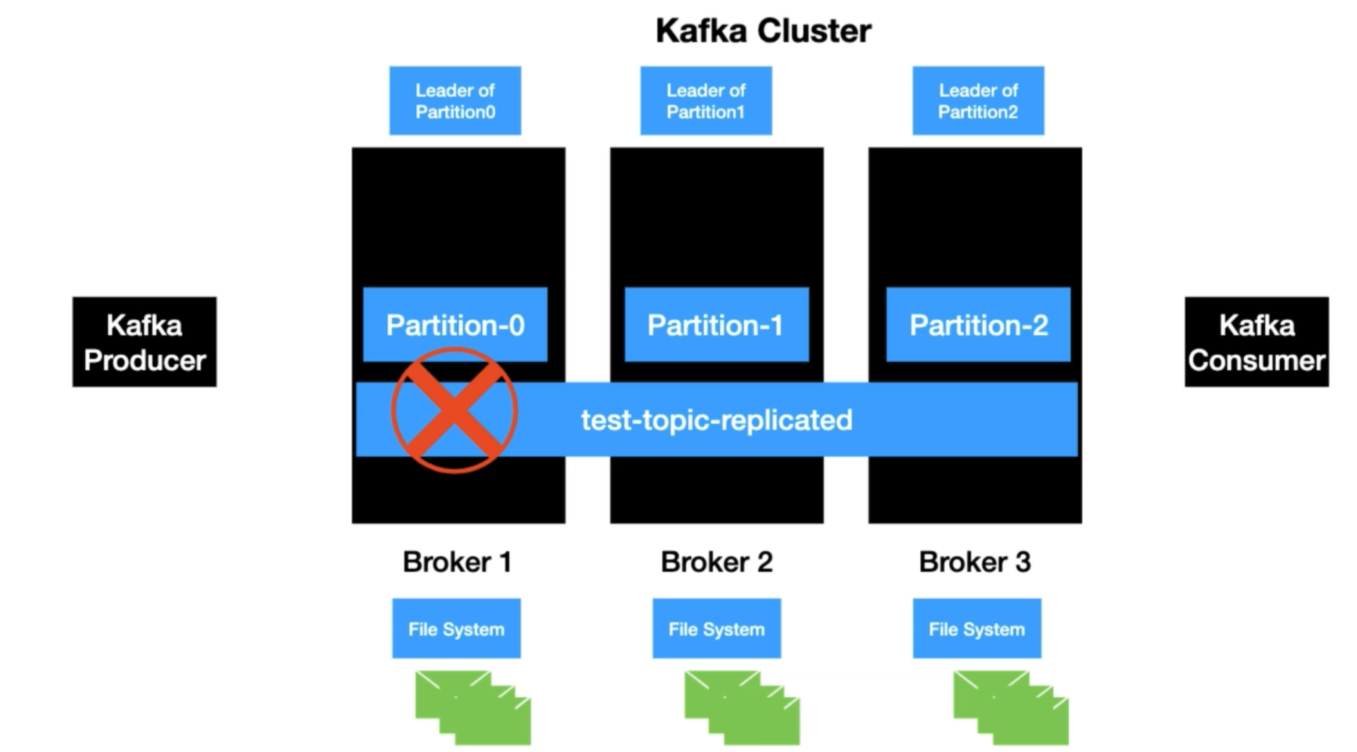
So in here, partition-0 call goes to broker 1, partition-1 call goes to broker 2 and so on and the process repeats.

To summarize ->



How Kafka handles Data Loss ? – Replication and In-Sync-Replica (ISR)

Lets learn how Kafka handles data loss in the event of the broker failure.



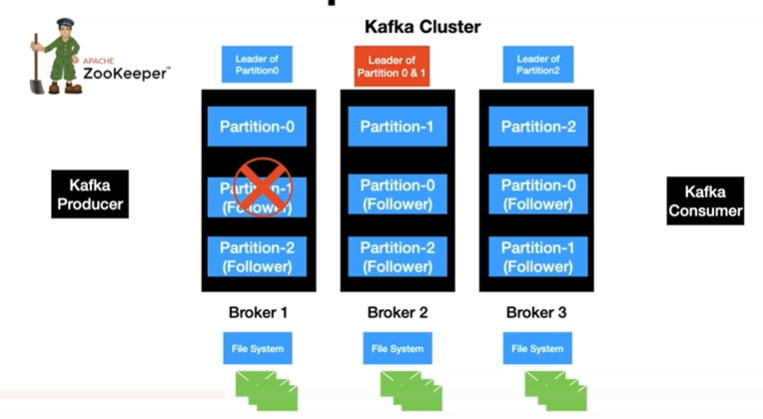
In here, we have Kafka cluster and a representation of how the topic is distributed across the cluster and we have some records present in the file system.

The clients which are the producers and consumers always talk to the leader to retrieve the data from a partition.

Lets say broker 1 goes down for some reasons. This is the broker which is the leader of partition-0, all the data which is returned to partition-0 is residing in the file system of this broker-1. Once it goes down there is no way for the clients to access the data. This is a big problem.

Kafka handles this issue using replication.





In here, We have Kafka producer on the left which produces the message to partition-0 and it goes to the leader which is the broker-1. The message is persisted into the filesystem. Now the broker 1 is the Leader replica.

Since the replication factor is 3, we need 2 more copies of the same message.

So the replication factor = number of copies of the same message.

So the next step is same message is copied to broker-2 and it gets written into the file system.

So Broker-2 is the follower of Partition-0 which is also known as the follower replica.

And the same step is repeated for broker-3.

Now we have 3 copies of same data available in all the brokers.

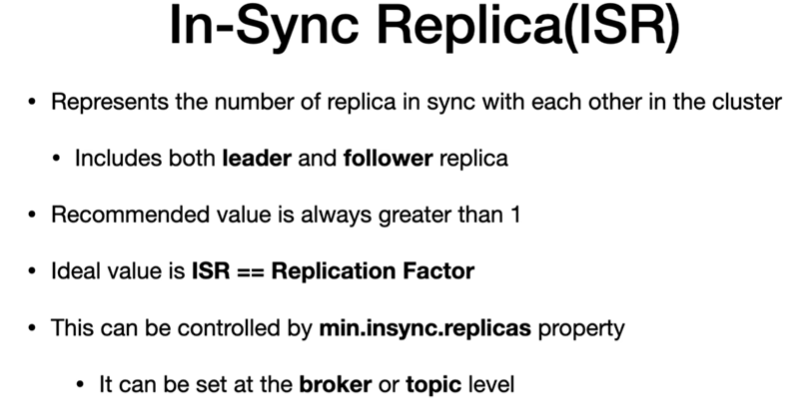
In Kafka terminology, this concept is known as Replication and the replica of the leader is called the leader replica and the other 2 replicas are called follower replicas.

Now its time to revisit the flow with the failure of broker-1.

So when broker-1 is down, still the data of the partition is available in broker 2 & 3. Zookeeper gets notified about the failure and it assign the new leader (Broker-2) to the controller. Now the Broker 2 is the leader of partition 0 & 1. This leader assignment is taken care by controller node which is the part of the cluster.

So now the clients request for producing and consuming the data for partition 0 will go the broker hereafter.

This is how Kafka handles data loss.



Fault Tolerance & Robustness in Kafka