Before going into the kafka let’s see what is message queue.

**Introduction to Message Queue:**

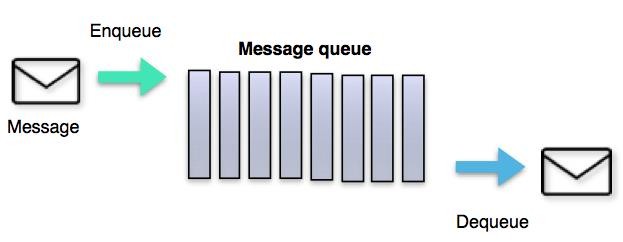
Message queuing allows applications to communicate by sending messages to each other.

The message queues provides a temporary message storage when the destination program is busy or not connected

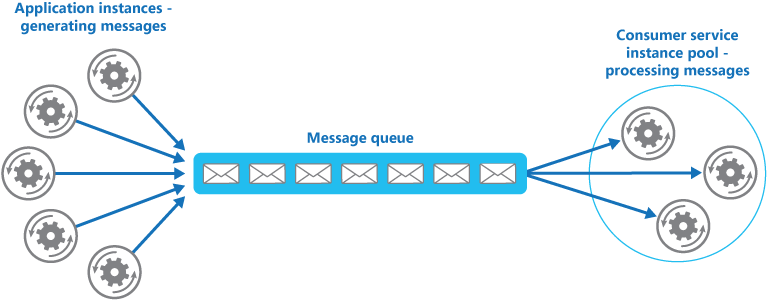
In other words,

* When a sender application is sending data continuously to a receiver program, the receiver may become busy at some time.
* Because the receiver program may do some computation with the received data in parallel.
* In that case, if the receiver is not able to accept some data, then that will lost by the receiver and the receiver can’t ask for that data to the sender again.
* To avoid this kind of data losing scenario, an intermediation was introduced called as MQ.
* This will only receiver the message continuously from the receiver and will retain it based on the set up and the receiver can fetch the info from the MQ

A queue is a line of things waiting to be handled - in sequential order starting at the beginning of the line. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed.



INCEPTEZ TECHNOLOGIES

**Need for distributed messaging queue:**

Distributed MQ offers,

* Distributed systems provides a very easy way to scale out. **(scalable)**
* It offers high throughput for both producer and consumer. **(though put)**
* It persist messages on disk and thus can be used for batched consumption such as ETL, in addition to real time applications. **(fault tolerant)**

**Kafka Introduction: (It is a distributed messaging queue)**

* Kafka is a leading general purpose publish-subscribe distributed messaging system, which offers strong durability, scalability and fault- tolerance support.
* Designed for processing of real time activity stream data such as logs, metrics collections.
* Written in Scala
* An apache project initially developed at LinkedIn at the year 2007.
* It is not specifically designed for Hadoop rather Hadoop ecosystem is just be one of its possible consumers.

In other words

**Kafka is a distributed, replicated commit log MQ service.**

* **Distributed** because Kafka is deployed as a cluster of nodes, for both fault tolerance and scale
* **Replicated** because messages are usually replicated across multiple nodes (servers).
* **Commit Log** because messages are stored in partitioned, append only logs which are called Topics. This concept of a log is the principal killer feature of Kafka.

**Why we need kafka:**

Before that lets see what is producer and consumer..

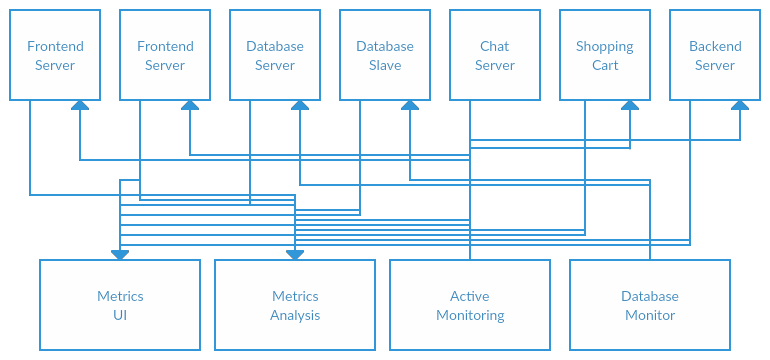
* **Producer** 🡪 the application which sends data
* **Consumer** 🡪 the application which receives the data sent by the producer.

Consider if we don’t use Kafka and if the producers and consumers interact to each other directly without a mediator,

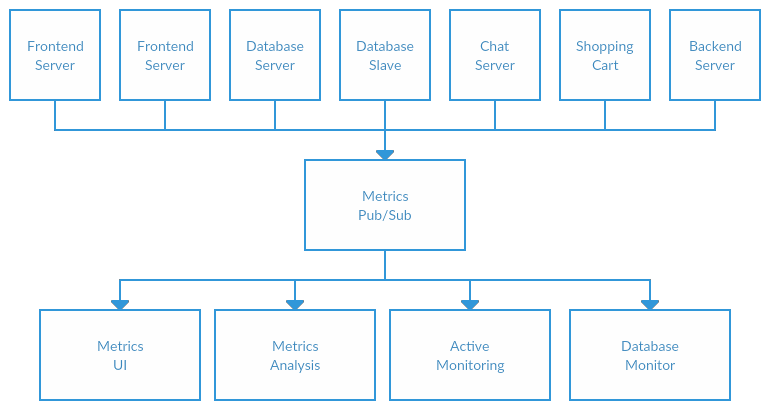
* then the network traffic will be complex.
* If the receiver is busy in a task and missed to receive a part of data, then its lost. In case of streaming data even producer cant send the lost data again to receiver.

To avoid that we introduce a mediator while transmitting data from a producer and consumer.

Direct Connection:



Publish/Subscribe system (such as kafka)



Here we have a mediator such as kafka which can

* receive messages from multiple producer at the same time (parallelism)
* can retain that message for particular configurable period of time (durable,fault tolerance)
* can be consumed by multiple consumer and multiple times..

Some of the key point to be noted are:

* Kafka is Highly scalable very easy to add large number of consumers without affecting performance and without down time. That's because Kafka does not track which messages in the topic have been consumed by consumers. It simply keeps all messages in the topic within a configurable period.
* Kafka handle spike of the events more efficiently. This is where Kakfa truly shines because it acts as a "shock absorber" between the producers and consumers. Kafka can handle events at 100k+ per second rate coming from producers. Because Kafka consumers pull data from the topic, different consumers can consume the messages at different pace.
* Kafka also supports different consumption model. You can have one consumer processing the messages at real-time and another consumer processing the messages in batch mode.
* Message durability is high in Kafka – Persists the messages/events for the specified time period
* Integration support for Kafka with other frameworks are high So mostly likely it needs to integrate with other event processing frameworks such as Apache Storm, Spark, Nifi, Flume etc to complete the job.

**Applications of Kafka**

**Messaging**

* Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications. In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

**Website Activity Tracking**

* The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type.

**Metrics**

* Kafka is often used for operation monitoring data pipelines. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

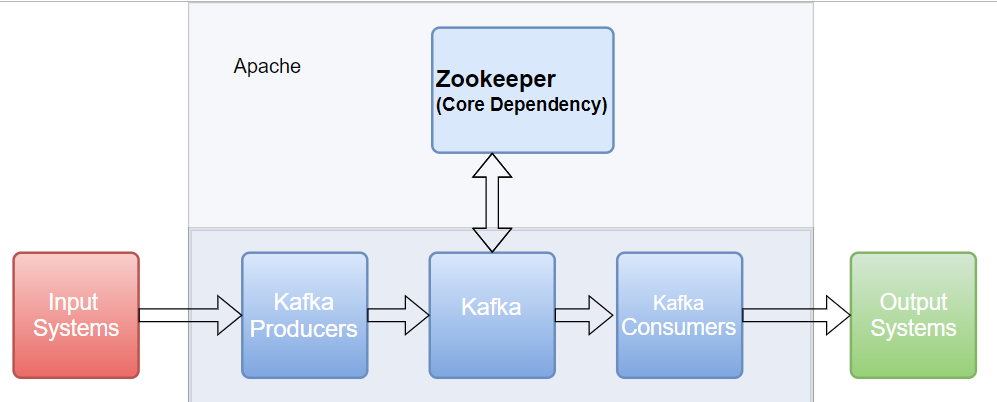
**Log Aggregation**

* Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

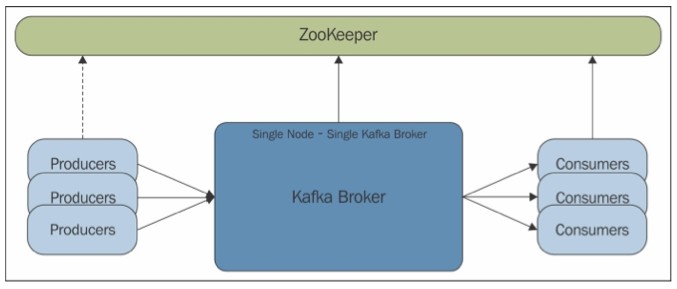
**Stream Processing**

* Many users end up doing stage-wise processing of data where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption. For example a processing flow for article recommendation might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage might attempt to match this content to users. This creates a graph of real-time data flow out of the individual topics. The Spark/Storm framework is one popular way for implementing some of these transformations.

**Kafka Architecture:**

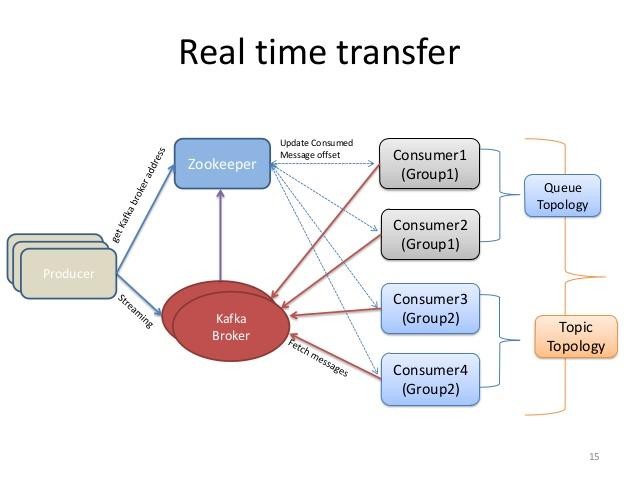
****

**Zookeeper**:

****

The below are the components which belongs to the Kafka architecture:

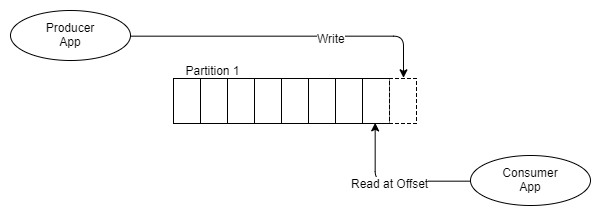
* Messages
  + The unit of data within Kafka with optional byte header, eg. Record in a table.
* Batches
  + Collection of messages eg. RecordSet
* Schema
  + Additional structure be imposed on the message eg. JSON, XML, Avro.
* Producer
  + It is the source of data which sends data to kafka.
  + Note: Kafka is not capable of fetching and sending data.
  + Some one needs to send data to kafka and similarly need to get data from kafka.
  + Maintains messagekey to deliver Old messages to the particular partitions.
* Consumer
  + It’s the consumer of the message sent by the producer
  + Subscribe to one or more topics and read messages in order.
  + Keep track of offset of last consumed messages.
* Broker
  + As we know kafka is distributed.
  + So Kafka may be running in many data noted.
  + The daemon which is running on each machine is called broker.
  + Each broker has a unique ID.
  + It is a single server running on one data node. Likewise many brokers would be running in a cluster.
  + The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk.
  + The broker responds to the consumer with the message based on the partitions.
* Kafka Cluster
  + It’s a Group of Brokers
  + One broker will act as controller and assigns leader for partition.
* Topic
  + It’s the category of message. For eg; a table or a folder
  + Its spread across all the brokers/nodes.
* Partition
  + Horizontal division of topics.
  + Replicated and scaled across multiple broker nodes.
* Replicas
  + Partitions in the Topics are replicated for high availability
  + ISR maintains – Fire and Forget, Sync and Async.

**Let’s see what’s the purpose of Zoo keeper in Kafka architecture:**

* Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs.
* Also to manage service discovery for Kafka Brokers that form the cluster.
* Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc.
* Zookeeper provides an in-sync view of Kafka Cluster configuration.

**Now let’s dive deeper to have some better understanding.**

Understanding the log (Topic) and its partitions are the key to understanding Kafka. So how is a partitioned log different from a set of queues? Let's visualise it.



* When even a new message is pushed into kafka, it appends the message to the existing queue which is corresponding to the topic.
* The message stays put whether it is consumed once or a thousand times.
* It is removed according to the data retention policy (often a window time period).
* So now how the consumer consumes the data is, the consumer keeps track of the pointer to the last message it consumed. That’s pointer is called **offset.**

|  |
| --- |
| Concept:  Producer:   * it’s the application which sends message to kafka. * Kafka accepts the data as array of bytes. * We producer sends a text file, each message will be sent as a message and kafka accepts each line in the file as array of bytes. * In case of producing data from a table, each row will be sent as message and in turn each message will be accepted as array of bytes.   Consumer:   * An application which receives data from kafka. * Any consumer can subscribe for a particular topic which is in kafka and can receive the data.   Broker   * It’s a single kafka broker.. many brokers will be running in a cluster. Each in different data node. * Each broker will be having a broker ID.   Topic   * It’s a specific kind of data which is sent by producer. * Its used to have a unique name for a specific data stream. * Consumers must subscribe for a topic and need to get data from it.   Partition   * Since the data produced by the producer may be very large, it cannot fit in a single broker. * So it is split into different partitions and are spread across the cluster of brokers. * Its replicated at the partition level * The number of partitions created for a topic is decided by the producer at the time of creation of that topic. * Kafka broker gets the instruction from producer and creates that many number of partitons.   Offset   * These are the sequence numbers assigned for each partition when the partitions arrive kafka. * Once the sequence numbers are assigned to the partitions, then don’t change. Its fixed to that partition. Immutable. * Kafka sets offset number based on the sequence of arrival of parittions. * First partition get the sequence numbers as zero. Second gets one. And so on. * So for a consumer to locate a message, it should know three things. * Topic name * Partition number * Offset   Consumer group   * It’s a group of consumers to share a work. * If couple of consumers who belongs to the same group reached kafka for a particular same topic, then the partitions of that message will be shared to both the consumers. * For eg: if there are 4 partitons in a topic, and two consumers who belong to same group reaches kafka for that topic, then out of 4 partitions 2 will be consumed by consumer1 and other two will be consumed by consumer2. * If two consumers who belong to different group reach kafka for that particular topic, then all 4 partitions will be consumed by both. It will not be shared. |

**From a different reference:**

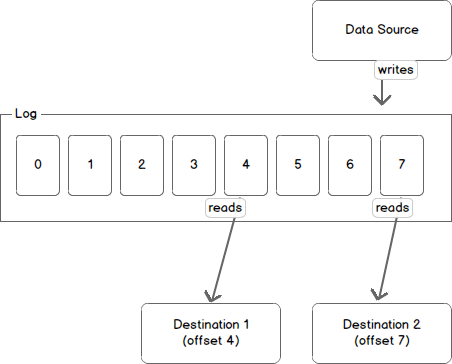
**Anatomy of kafka topic.**

* Kafka topics are divided into a number of partitions.
* Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers — each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel.
* Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message **processing throughput**.
* Each message within a partition has an identifier called its **offset**.
* The offset ordering of messages are an immutable sequence.
* Kafka maintains this message ordering depending on the arrival order.
* Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit.
* Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message’s topic, partition, and offset within the partition.
  + Like (“topic”,”partition#”,”offset”)



**Log anatomy**

* Another way to view a partition is as a log.
* A data source writes messages to the log and one or more consumers reads from the log at the point in time they choose.
* In the diagram below a data source is writing to the log and consumers A and B are reading from the log at different offsets.

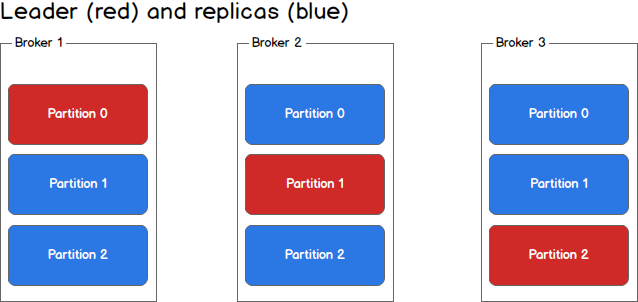


**Data log**

* Kafka retains messages for a configurable period of time and it is up to the consumers to adjust their behaviour accordingly.
* For instance, if Kafka is configured to keep messages for a day and a consumer is down for a period of longer than a day, the consumer will lose messages.
* However, if the consumer is down for an hour it can begin to read messages again starting from its last known offset.
* From Kafka’s point of view, it keeps no state on what the consumers are reading from a topic.
* Only the consumer has the stats of what he consumed and where is need to resume/start.

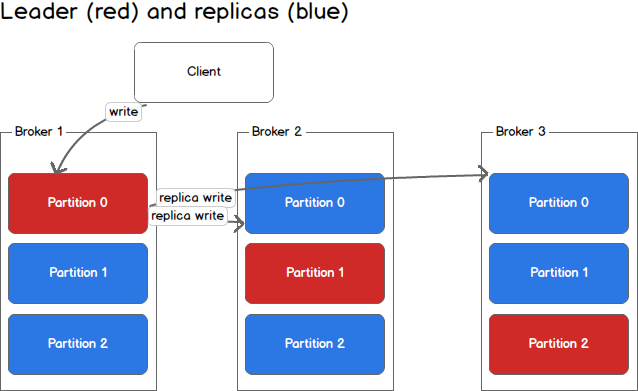
**Partitions and Brokers**

* Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic.
* All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data.
* If a leader fails, a replica takes over as the new leader.



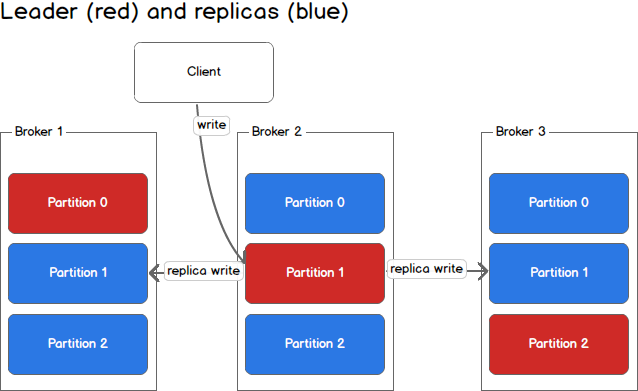
**Producers**

* Producers write to a single leader, this provides a means of load balancing production so that each write can be serviced by a separate broker and machine.
* In the first image, the producer is writing to partition 0 of the topic and partition 0 replicates that write to the available replicas.



**Producer writing to partition.**

* In the second image, the producer is writing to partition 1 of the topic and partition 1 replicates that write to the available replicas.



* Since each machine is responsible for each write, throughput of the system as a whole is increased.

**Consumers and Consumer Groups**

* Consumers read from any single partition, allowing you to scale throughput of message consumption in a similar fashion to message production.
* Consumers can also be organized into consumer groups for a given topic — each consumer within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic.
* If you have more consumers than partitions then some consumers will be idle because they have no partitions to read from. If you have more partitions than consumers then consumers will receive messages from multiple partitions. If you have equal numbers of consumers and partitions, each consumer reads messages in order from exactly one partition.
* The following picture describes the situation with multiple partitions of a single topic. Server 1 holds partitions 0 and 3 and server 2 holds partitions 1 and 2. We have two consumer groups, A and B. A is made up of two consumers and B is made up of four consumers. Consumer Group A has two consumers of four partitions — each consumer reads from two partitions. Consumer Group B, on the other hand, has the same number of consumers as partitions and each consumer reads from exactly one partition.
* 

**Consistency and Availability**

Kafka makes the following guarantees about data consistency and availability:

1. Messages sent to a topic partition will be appended to the commit log in the order they are sent
2. A single consumer instance will see messages in the order they appear in the log
3. A message is ‘committed’ when all in sync replicas have applied it to their log
4. Any committed message will not be lost, as long as at least one in sync replica is alive.

**Writing message into kafka**

While producer produces message into kafka the following happens.

1. First step of producer is, it serialize the data using the given key
2. Then its given to the partitioner. The partitioner will use the given message key to determine the appropriate partition.
3. Kafka will hash # the key to get the particular partition number. So if we give the same key it will be stored in the same partition.
4. If we don’t give particular key, then kafka will define its own key and will evenly distribute the message across available partitions using ROUND ROBIN ALGORITHM.
5. Actually here the message will not the written into the disk as streaming data.
6. It will be stored in the in-memory buffer and will be stored in the disk as batches.
7. Since kafka stored in-memory, the consumer can consume the data immediately without any disk i/o hit and will be very fast.

**Callback and Acknowledgements**

We have different approaches to implement a Kafka Producer. We also have some acknowledgment and call back methods to handle responses from brokers.

We will see the concepts here.

There are three approaches to send a message to Kafka

1. Fire-and-forget Producer
2. Synchronous Producer
3. Asynchronous Producer
4. In Flight Messages

* **Fire-and-forget:**
* Here producer send message to kafka continuously and it don’t care of any acknowledgement.
* If the message sent to the kafka broker is not stored there, then it will not be intimated to producer. In this case there are chances to loose data.
* To mention this types of ackw in code, mention this acks=0
* The record will be immediately added to the socket buffer and considered sent.
* No guarantee can be made that the server has received the record in this case, and the **retries** configuration will not take effect (as the client won't generally know of any failures).
* The offset given back for each record will always be set to -1.

**A-Synchronous:**

* This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers.
* In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.
* In code the mode is set to acks=1

**Synchronous:**

* Here the producer sends data to kafka producer and waits for acknowledgement from all the replications created for the particular partition.
* Once the acknowledgement is received, it send the next message.
* This process impacts the data throughput because, there is a delay in sending messages to kafka producer looking for the acknowledgement for reach message before sending the next message.
* This means the leader will wait for the full set of in-sync replicas to acknowledge the record.
* This guarantees that the record will not be lost as long as at least one in-sync replica remains alive.
* This is the strongest available guarantee. This is equivalent to the acks=-1 setting.
* In code the mode is set to acks=all

**Few other properties to be known.**

**bootstrap.servers:**

* A list of host/port pairs to use for establishing the initial connection to the Kafka cluster.
* The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers.
* This list should be in the form host1:port1,host2:port2,....
* Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

**buffer.memory:**

* The total bytes of memory the producer can use to buffer records waiting to be sent to the server.
* If records are sent faster than they can be delivered to the server the producer will block for max.block.msafter which it will throw an exception.
* This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering.
* Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.

**compression.type:**

* The compression type for all data generated by the producer.
* The default is none (i.e. no compression). Valid values are none, gzip, snappy, lz4, or zstd.
* Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression).

**Retries:**

* Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error.
* Note that this retry is no different than if the client resent the record upon receiving the error.
* Allowing retries without setting max.in.flight.requests.per.connection to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first.
* Note additionall that produce requests will be failed before the number of retries has been exhausted if the timeout configured by delivery.timeout.ms expires first before successful acknowledgement.
* Users should generally prefer to leave this config unset and instead use delivery.timeout.ms to control retry behavior.

**Example for PRODUCER Program:**

|  |
| --- |
| package com.inceptez.kafka  object kafkaproducer extends App {    import java.util.Properties  import org.apache.kafka.clients.producer.\_  /\*  **The acks=0 -> means the Producer does not wait for any ack from Kafka broker at all. (fire-and-forget)**  **The acks=1 is leader acknowledgment. (A-sync)**  **The acks=all or acks=-1 (Synch) is all acknowledgment which means the leader gets write confirmation from**  **the full set of ISRs before sending an ack back to the producer.**  \*/    val props = new Properties()  props.put("bootstrap.servers", "192.168.138.145:9092,192.168.138.145:9093")  **//first property is the ip addr of broker and its port no.**  **//multiple brokers/ports can be mentioned by comma separated.**  props.put("acks","0") **// 🡪 here the ack mode is its fire and forget.**  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")  **/\*here the kind of serialization has to be mentioned. In this example**  **we use string serializer. While consuming this message we need to used the same serializer. \*/**  val producer = new KafkaProducer[String, String](props)  **/\*while creating instance for the class KafkaProducer, we need to pass parameters which has the properties which we specified using PUT().**  **Here we pass the key value pair input which is defined in the map variable props \*/**  val record = new ProducerRecord("tk3", "100", "hello")  producer.send(record)  **/\* the created object “producer” has the config info..**  **Now the above created object “record” has the below info.**  **This consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value. \*/**    println(s"End")  producer.close()  }  **/\* here we hard core some value to the ProducerRecord class (“hello”) and we have producing it to the topic “tk3” which is asked to write in the broker specified in the IP address/port no. \*/** |

**Example for CONSUMER Program:**

|  |
| --- |
| package com.inceptez.kafka  import java.util  import org.apache.kafka.clients.consumer.KafkaConsumer  import scala.collection.JavaConverters.\_  object kafkaconsumer extends App {  import java.util.Properties  val TOPIC="tk3"  val props = new Properties()  props.put("bootstrap.servers", "localhost:9092")  props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")  props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")  **/\***  **Auto.offset.reset is a property to specify whether you want to consume the records**  **from the beginning (earliest) or the last committed offset (latest).**  **\*/**  props.put("auto.offset.reset", "latest")  props.put("group.id", "g123")  val consumer = new KafkaConsumer[String, String](props)  **/\*similarly to producer program here we use the class KafkaConsumer() along with the properties \*/**  consumer.subscribe(util.Collections.singletonList(TOPIC))  **//here we subscribe to a topic**  while(true){  val records=consumer.poll(100)  for (record<-records.asScala){  println(record)  **// we get 100 records from consumer() and loop to process it.**  }  }  } |