4/6/2021

Bipin Kumar

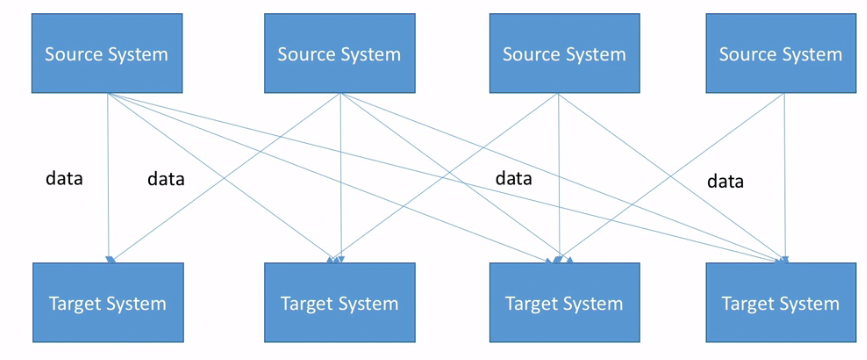
shael.dhn88@gmail.com

**KAFKA**

Kafka notes

When we start writing any application we generally have source system and target system as given in below diagram.

But after time system become complex and more complicated as shown in below diagram.



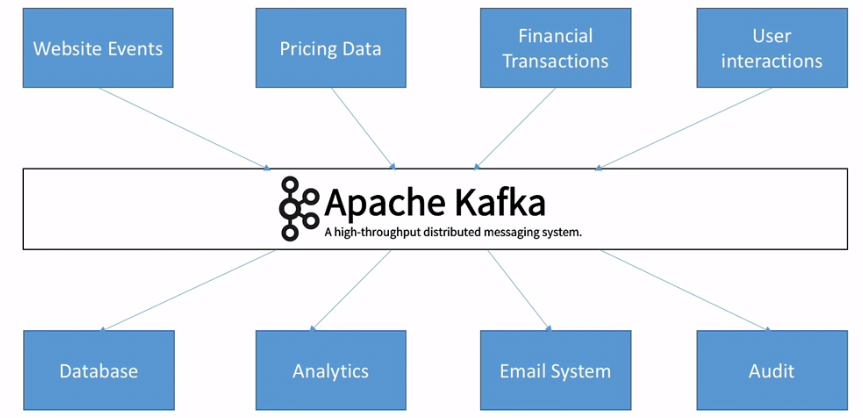
If we take above example and suppose if we have 4 source system and 4 target system then we need to write 16 integration code.

Each integration will come with some difficulties.

* Protocol: which protocol it gone use (HTTP, FTP, TCP, JDBC etc.)
* Data format: How the data is parsed (csv, json, xml etc.)
* Data schema and evolution- How the data id shaped and may change.

Each source system will have an increase load from the connections.

To resolve these issue Apache Kafka came in picture as shown in below diagram.



So Now all the source system send data to Kafka and target system will use these data as per there needs. So due to this we achieve decoupling of system.

Why Kafka?

* Created by linkedin, now open source project mainly maintained by confluent.
* Distributed, resilient architecture, fault tolerant.
* Horizontal scalability:
* Can scale to 100s of brokers
* Can scale to millions of messages per second.
* High performance (latency of less than 10ms) real time
* Used by the 2000+ firms, 35% of the fortune 500: e.g. Airbnb, linkedin, uber,Netflix etc

Use cases:

* Messaging system
* Activity tracking
* Gather metrics from many different locations
* Application logs gathering
* Stream processing (With the kafka streams API or Spark for example)
* De-coupling of system dependencies
* Integration with spark, Flink, storm, Hadoop and many other big data technologies.

e.g. Netflix uses kafka to apply recommendation in real time while you’re watching TV shows

Uber uses kafka to gather user, taxi and trip data in real time to compute and forecast demand and compute surge pricing in real time.

LinkedIn uses kafka to prevent spam, collect user interaction to make better connection recommendations in real time.

Kafka is only used as a transportation mechanism.

Kafka Theory:

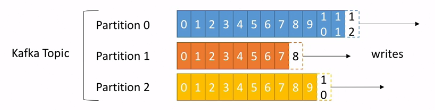
1. Topics, Partitions and offsets:

Topics: a particular stream of data

* Similar to a table in a database (without all the constraints)
* You can have as many topics as you want
* A topic is identified by its name

Topics are split in partitions

* Each partition is ordered.
* Each message within a partition gets an incremental id, called offset.



Topic example: truck\_gps

* Say we have a fleet of trucks; each truck reports its GPS position to Kafka.
* You can have a topic trucks\_gps that contains the position of all trucks.
* Each truck will send a message to kafka every 20 seconds, each message will contain the truck ID and the truck position (latitude and longitude)
* We choose to create that topic with 10 partitions (arbitrary number)

Offset only have a meaning for a specific partition. E.g. offset 3 in partition 0 doesn’t represent the same data as offset 3 in partition 1.

* Order is guaranteed only within a partition (not across partitions)
* Data is kept only for a limited time (default is one week)
* Once the data is written to a partition, it can’t be changed.
* Data is assigned randomly to a partition unless a key is provided.

**Broker:**

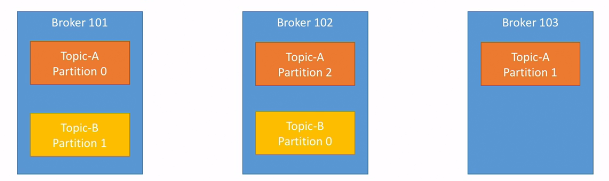
* A Kafka Cluster is composed of multiple brokers(servers)
* Each broker is identified with its ID (integer)
* Each broker contains certain topic partitions.
* After connecting to any broker (called a bootstrap broker), we will be connected to the entire cluster.
* A good number to get started is 3 brokers, but some big clusters have over 100 brokers.
* In these examples we choose to number brokers starting at 100 (arbitrary)



Example of Topic-A with 3 partitions:

Then Topic-A with partition 0 is in 101, partition 2 in 102 and partition 1 in 103.

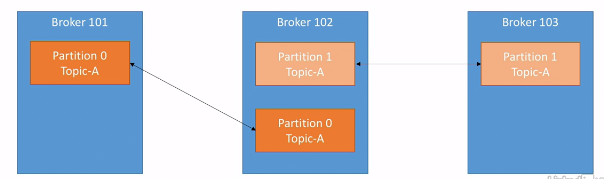
Now suppose we have Topic-B with 2 partitions then may be broker 101 will have partition 1, 102 will have partition 0 as shown in below diagram.



Note: Data is distributed and broker 103 doesn’t have any topic B data.

**Topic replication factor:**

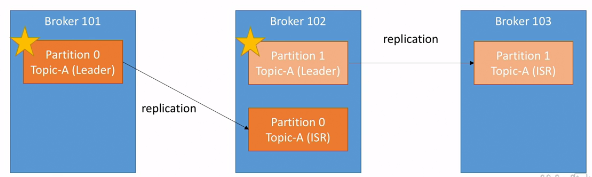
* Topic should have a replication factor > 1 (usually between 2 and 3)
* This way if a broker is down, another broker can serve the data.
* Example Topic-A with 2 partitions and replication factor of 2.



Suppose we lost broker 102. Result broker 101 and 103 still serve the data.

Concept of leader for a partition.

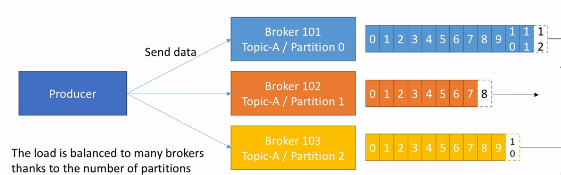
* At any time only one broker can be a leader for a given partition.
* Only that leader can receive and serve data for a partition.
* The other brokers will synchronize the data.
* Therefore, each partition has one leader and multiple ISR (in-sync replica)



Leader and ISR decide by zookeeper.

Producers:

* Producer write data to topics (which is made of partitions)
* Producer automatically know to which broker and partition to write to.
* In case of broker failures, Producers will automatically recover.



If we send data without key then data will be distributed between brokers in round robin fashion.

Producers can choose to receive acknowledgment of data writes:

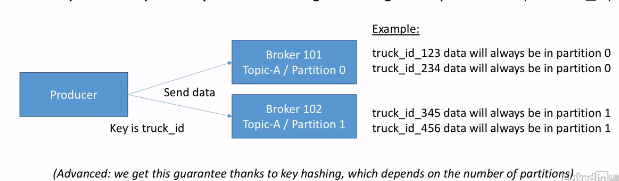
Acks=0: producer won’t wait for acknowledgment (possible data loss)

Acks=1: producer will wait for leader acknowledgment (limited data loss)

Acks=all: leader + replicas acknowledgment (no data loss)

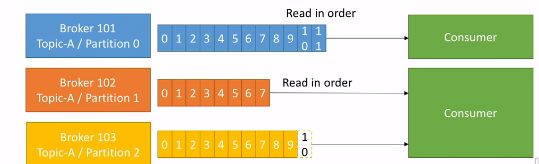
Producers: Message keys

* Producers can choose to send a key with the message (string, number etc.)
* If key=null, data is sent round robin.
* If a key is sent, then all message for that key will always go to the same partition.
* A key is basically sent if you need message ordering for a specific field (e.g.: truck\_id)



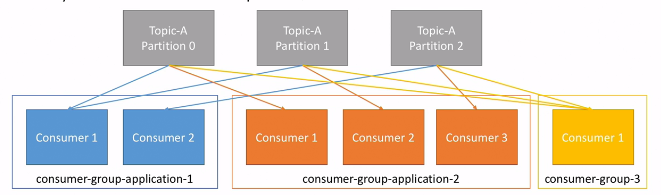
Consumers:

* Consumers read data from a topic (identified by name)
* Consumers know which broker to read from
* In case of broker failures, consumers know how to recover
* Data is read in order within each partition



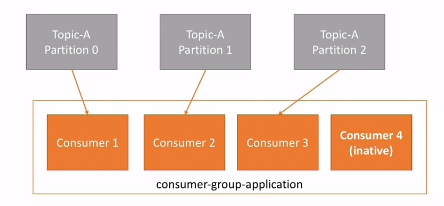
Consumer groups:

* Consumer read data in consumer groups.
* Each consumer within a group reads from exclusive partitions
* If you have more consumers than partitions, some consumers will be inactive.



Note: Consumers will automatically use a Group Coordinator and a consumer coordinator to assign a consumer to a partition.

If we have more consumers than partitions, some consumers will be inactive.



**Consumer Offsets:**

* Kafka stores the offsets at which a consumer group has been reading.
* The offsets committed live in a kafka topic named \_\_consumer\_offsets
* When a consumer in a group has processed data received from Kafka, it should be committing the offsets.
* If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets.



* Consumers choose when to commit offsets.
* There are 3 delivery semantics:
* At most once:

- offset are committed as soon as the message is received.

- if the processing goes wrong the message will be lost (it won’t be read again)

* At least once (Usually preferred):

- offsets are committed after the message is processed.

- If the processing goes wrong, the message will be read again.

- This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won’t impact your systems)

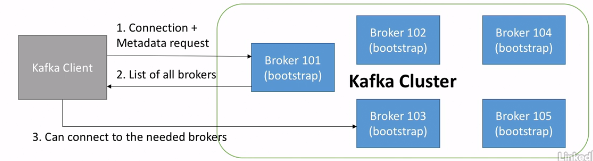
* Exactly once:

- Can be achieved for Kafka => Kafka workflows using Kafka streams API

- For Kafka => External system workflows, use an idempotent consumer.

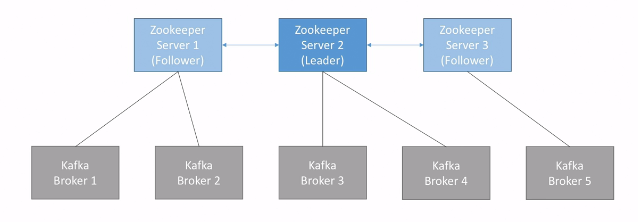
**Kafka Broker Discovery:**

* Every Kafka broker is also called a “Bootstrap server”
* That means that you only need to connect to one broker, and you will be connected to the entire cluster.
* Each broker knows about all brokers, topics and partitions (metadata).



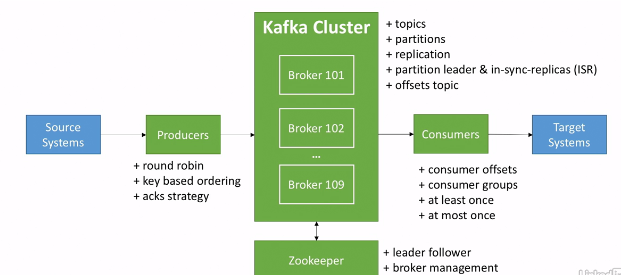
Zookeeper:

* Zookeeper manages brokers (keeps a list of them)
* Zookeeper helps in performing leader election for partitions
* Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics etc.
* Kafka can’t work without zookeeper
* Zookeeper by design operates with an odd number of servers (3, 5, 7 …)
* Zookeeper has a leader (handle writes) the rest of the servers are followers (handle reads)
* Zookeeper does not store consumer offsets with Kafka > v0.10



Kafka Guarantees:

* Messages are appended to a topic-partition in the order they are sent.
* Consumers read messages in the order stored in a topic- partition
* With a replication factor of N, Producers and consumer can tolerate up to N-1 brokers being down.
* This is why a replication factor of 3 is good idea. Allows for one broker to be taken down for maintenance, and for another broker to be taken down unexpectedly.
* As long as the number of partitions remains constant for a topic ( no new partitions), the same key will always go to the same partition.



**Kafka installation in windows:**

* Install java 8 and set its path in environment variable.
* Download binaries for kafka from its official website. <https://kafka.apache.org/downloads>
* Extract downloaded file in C:// drive.
* Set C:\kafka\_2.12-2.7.0\bin\windows in path variable.

In Kafka directory create folder called data and inside it create two more folder called zookeeper and Kafka. Now open config folder and open zookeeper.properties file and change dataDir to folder which you create for zookeeper. Similarly open server.properties file and change the log.dirs=C:/kafka\_2.12-2.7.0/data/kafka.

To check if path set correctly. Open terminal or cmd and type below commands.

Cd C:\kafka\_2.12-2.7.0

zookeeper-server-start.bat config/zookeeper.properties

Now open another terminal and go to kafka directory. And then run below command

kafka-server-start.bat config/server.properties

Kafka CLI:

To start using Kafka first start zookeeper then start Kafka server using below command.

* zookeeper-server-start config/zookeeper.properties
* kafka-server-start config/server.properties

1. Kafka-topics:

* When we type kafka-topics and enter it will show you all the argument it takes.
* kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --create you will get Missing required argument "[partitions]"
* kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --create --partitions 3 you will get Missing required argument "[replication-factor]"
* kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --create --partitions 3 --replication-factor 2 you will get Replication factor: 2 larger than available brokers: 1.
* kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --create --partitions 3 --replication-factor 1 you will get success message.

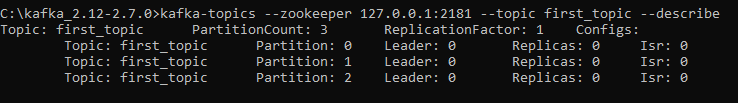
To list down all the topics:

kafka-topics --zookeeper 127.0.0.1:2181 –list



To describe topics:

kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --describe



Here leader is broker 0, replicas is also in broker 0 and isr is also in broker 0. When we have more broker then we can see different broker id there.

To delete topics:

kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic –delete

In windows there is bug in kafka due to which if you try to delete a topic it will automatically get crashed to overcome this go to data/kafka folder and manually delete topic.

1. Kafka console producer:

kafka-console-producer it will give you all the list of argument you will get. Here there are two argument is required.

--bootstrap-server

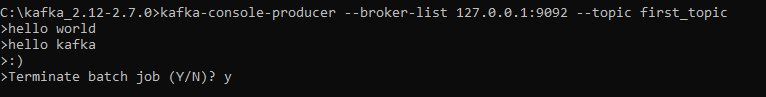
--broker-list (deprecated)

--topic

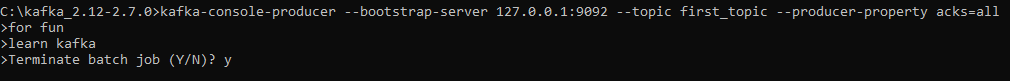
kafka-console-producer --broker-list 127.0.0.1:9092 --topic first\_topic

kafka-console-producer --bootstrap-server 127.0.0.1:9092 --topic first\_topic

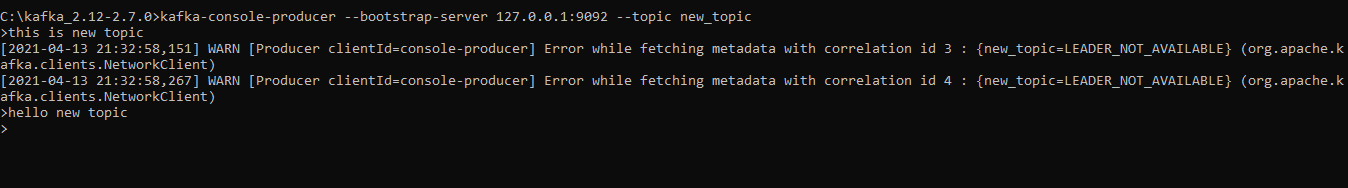
If everything is all right then you will get ‘>’



kafka-console-producer --bootstrap-server 127.0.0.1:9092 --topic first\_topic --producer-property acks=all



kafka-console-producer --bootstrap-server 127.0.0.1:9092 --topic first\_topic



Here new topic is not present that’s why it give warning but as we know Kafka is fault tolerance system it automatically created the topic with default value i.e. partition 1 and replication-factor 1.

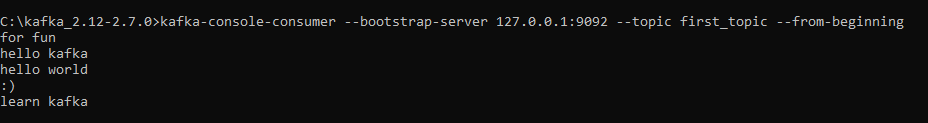
So, it is recommended that before producing message first create topics. We can change the default value by changing value in server.properties

1. Kafka console consumer:

kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first\_topic

when we entered above cmd we will not see any message it is because it only consume new messages. So, to see all the messages we need to run below command. It also read new messages.

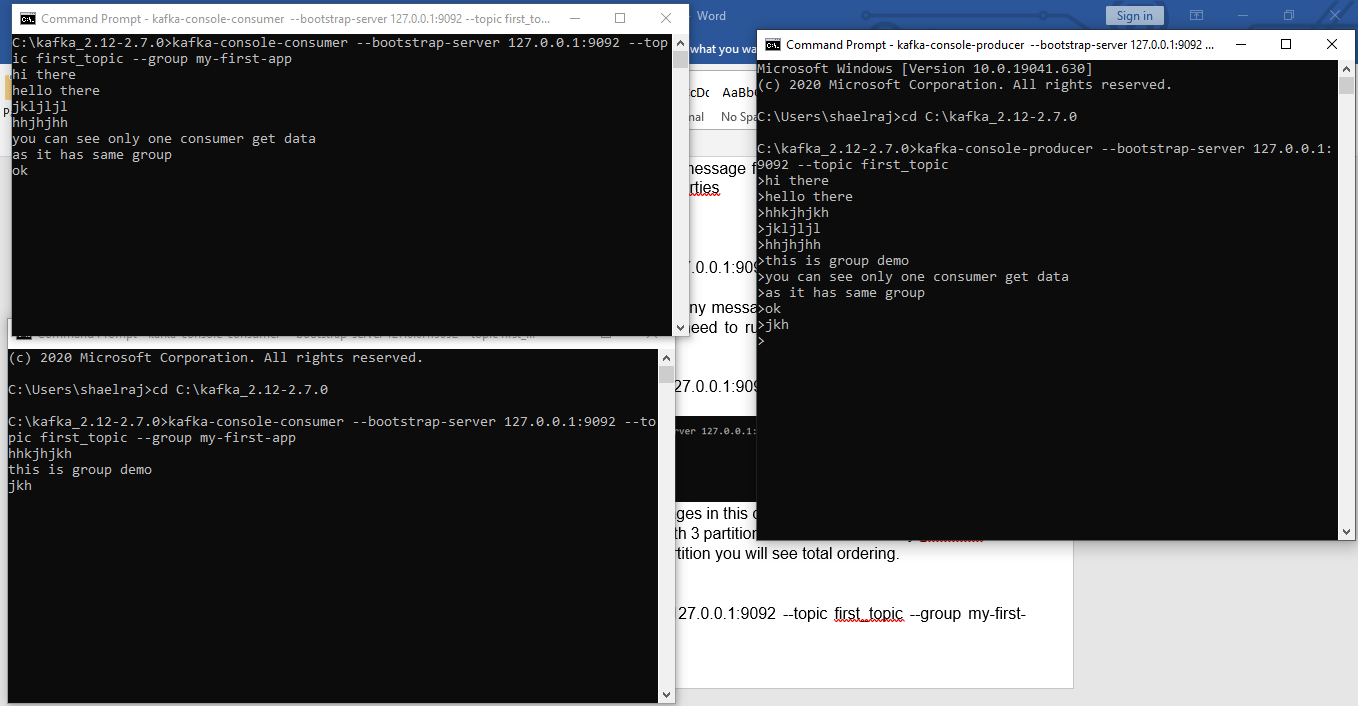
kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first\_topic --from-beginning



Notes: As we can see, the order of the messages in this consumer is not “total”, the order is per partition. Because “first\_topic” was created with 3 partitions and the order is only guranted at the partition level. If you try with a topic with 1 partition you will see total ordering.

Consumer-group:

kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first\_topic --group my-first-app



kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first\_topic --group my-second-app –from-beginning

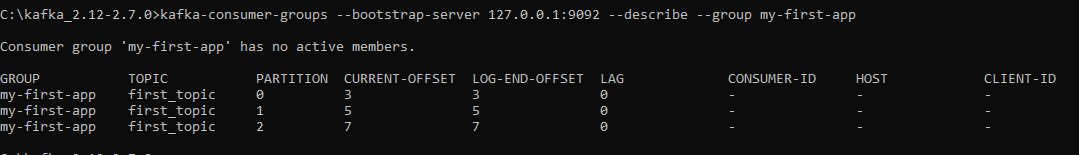
It will show you all the message from beginning.

Now stop the command and re-run the command, you will not see the messages because its offset is committed for the group and now it will only show you new messages.

1. kafka-consumer-groups:

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --list

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group my-first-app



Here lag is zero because we already consume new message.

You can see consumer-id and host when we have running kafka-console-consumer

1. Resetting offset:

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group my-first-app --reset-offset --to-earliest

it will give you some error related to --execute

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group my-first-app –reset-offset --to-earliest –execute

Again it will give you error because it doesn’t know for which topics it needs to reset offset.

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group my-first-app –reset-offset --to-earliest –execute --topic first\_topic

now if we run consumer again then we can see the all the messages from beginning.

kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group my-first-app –reset-offset –shift-by 2 –execute --topic first\_topic

it will shift the offset by 2.

Sometime we want to see all the data in Kafka UI for this we can install Kafka tool <https://www.kafkatool.com/>

Kafka Java Programming:

1. ProducerDemo:

Step 1: Create properties for help you can refer below link. You can also use ProducerConfig class to use pre-defined static properties instead of using hard coded string. <https://kafka.apache.org/documentation/#producerconfigs>

Step 2: Create Producer using KafkaProducer class and using ProducerRecord class create records which needs to send to kafka.

Step 3: Send data using send method. After this you will not able to see the record because it sends data asynch so to see data you need flush and close producer. Using flush and close method.

For callback: use below code:

producer.send(record, **new** Callback() {

**public** **void** onCompletion(RecordMetadata metadata, Exception e) {

// Execute every time when record sent successfully or exception thrown

**if** (e != **null**) {

***LOG***.error("Error occured : {}", e.getMessage());

}

***LOG***.info("Received Metadata \n Topic:{} \n Partition :{} \n Offset :{} \n TimeStamp :{}",

metadata.topic(), metadata.partition(), metadata.offset(), metadata.timestamp());

}

});

For key value pair we can use another constructor of ProducerRecord class

1. Consumer: for consumer you can follow below steps.

Step 1) create properties using ConsumerConfig class.

Step 2) create consumer using kafkaconsumer class. Pass the topic list to subscribe the topic to be consumed.

Step 3) Poll on consumer to get the consumerrecords. Using below code.

ConsumerRecords<String, String> records = consumer.poll(Duration.*ofMillis*(100));

**for** (ConsumerRecord<String, String> record : records) {

***LOG***.info("Key : {} Value :{}", record.key(), record.value());

***LOG***.info("Partition : {} Offsets :{}", record.partition(), record.offset());

}

We can also see how kafka distribute its load by running same class multiple time with same groups.

We can also use assign and seek functionality to read data from particular offset.

// create consumer

KafkaConsumer<String, String> consumer = **new** KafkaConsumer<String, String>(prop);

// assign

TopicPartition partitionToReadFrom = **new** TopicPartition(topic, 0);

consumer.assign(Arrays.*asList*(partitionToReadFrom));

**long** offsetToRead = 10L;

// seek consumer

consumer.seek(partitionToReadFrom, offsetToRead);

**int** noOfMsgToRead = 5;

**boolean** keepReading = **true**;

**int** count = 0;

// poll for new data

**while** (keepReading) {

ConsumerRecords<String, String> records = consumer.poll(Duration.*ofMillis*(100));

**for** (ConsumerRecord<String, String> record : records) {

count++;

***LOG***.info("Key : {} Value :{}", record.key(), record.value());

***LOG***.info("Partition : {} Offsets :{}", record.partition(), record.offset());

**if** (count >= noOfMsgToRead) {

keepReading = **false**;

**break**;

}

}

}

consumer.close();

***LOG***.info("Application existng !!!");

}

**Real world Example:**

Create TwitterProducer class and follow below steps.

Step 1) add maven dependency given in below link and use sample code given to create client to connect to twitter.

<https://github.com/twitter/hbc>

Step 2) follow the same steps to create kafka producer as given above and instead of hard code value pass message to producer.

Step 3) start zookeeper and kafka broker and also create new topic which you gone use then start kafka-console consumer. You can also try to change items value and tweets in your profile to see if it is picked or not.

Now we can see how to set other important properties. As prints in logs

[main] INFO org.apache.kafka.clients.producer.ProducerConfig - ProducerConfig values:

acks = 1

batch.size = 16384

bootstrap.servers = [127.0.0.1:9092]

buffer.memory = 33554432

client.dns.lookup = use\_all\_dns\_ips

client.id = producer-1

compression.type = none

connections.max.idle.ms = 540000

delivery.timeout.ms = 120000

enable.idempotence = false

interceptor.classes = []

internal.auto.downgrade.txn.commit = false

key.serializer = class org.apache.kafka.common.serialization.StringSerializer

linger.ms = 0

max.block.ms = 60000

max.in.flight.requests.per.connection = 5

max.request.size = 1048576

metadata.max.age.ms = 300000

metadata.max.idle.ms = 300000

metric.reporters = []

metrics.num.samples = 2

metrics.recording.level = INFO

metrics.sample.window.ms = 30000

partitioner.class = class org.apache.kafka.clients.producer.internals.DefaultPartitioner

receive.buffer.bytes = 32768

reconnect.backoff.max.ms = 1000

reconnect.backoff.ms = 50

request.timeout.ms = 30000

retries = 2147483647

retry.backoff.ms = 100

sasl.client.callback.handler.class = null

sasl.jaas.config = null

sasl.kerberos.kinit.cmd = /usr/bin/kinit

sasl.kerberos.min.time.before.relogin = 60000

sasl.kerberos.service.name = null

sasl.kerberos.ticket.renew.jitter = 0.05

sasl.kerberos.ticket.renew.window.factor = 0.8

sasl.login.callback.handler.class = null

sasl.login.class = null

sasl.login.refresh.buffer.seconds = 300

sasl.login.refresh.min.period.seconds = 60

sasl.login.refresh.window.factor = 0.8

sasl.login.refresh.window.jitter = 0.05

sasl.mechanism = GSSAPI

security.protocol = PLAINTEXT

security.providers = null

send.buffer.bytes = 131072

socket.connection.setup.timeout.max.ms = 127000

socket.connection.setup.timeout.ms = 10000

ssl.cipher.suites = null

ssl.enabled.protocols = [TLSv1.2]

ssl.endpoint.identification.algorithm = https

ssl.engine.factory.class = null

ssl.key.password = null

ssl.keymanager.algorithm = SunX509

ssl.keystore.certificate.chain = null

ssl.keystore.key = null

ssl.keystore.location = null

ssl.keystore.password = null

ssl.keystore.type = JKS

ssl.protocol = TLSv1.2

ssl.provider = null

ssl.secure.random.implementation = null

ssl.trustmanager.algorithm = PKIX

ssl.truststore.certificates = null

ssl.truststore.location = null

ssl.truststore.password = null

ssl.truststore.type = JKS

transaction.timeout.ms = 60000

transactional.id = null

value.serializer = class org.apache.kafka.common.serialization.StringSerializer

**Acks:**

acks = 0 (no acks)

* No response is requested
* If the broker goes offline or an exception happens, we won’t know and will lose data
* Useful for data where it’s okay to potentially lose messages like metrics collection, log collection.
* It is faster than other.

acks = 1 (leader acks)

* Leader response is requested, but replication is not a guarantee (happens in the background)
* If an ack is not received, the producer may retry.
* If the leader broker goes offline but replicas haven’t replicated the data yet, we have a data loss.

acks = all (replicas acks)

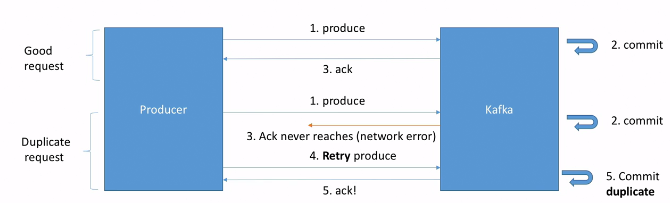
* Leader + replicas ack requested.
* It will send data to leader and then leader send data to replicas and collect all the acks and send to producer.
* Added latency and safety.
* No data loss if enough replicas.
* Acks=all must be used in conjunction with min.insync.replicas.
* Min.insync.replicas can be set at the broker or topic level(override).
* Min.insync.replicas=2 implies that at least 2 brokers that are ISR (including leader must respond that they have the data.
* That means if you use replication.factor=3, min.insync=2,acks=all, you can only tolerate 1 broker going down, otherwise the producer will receive and exception on send.

**Producer reties:**

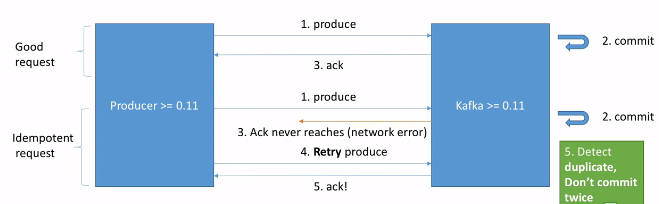
* In case of transient failures, developers are expected to handle exceptions,otherwise the data will be lost.
* Example of transient failue notenoughreplicasexception.
* There is a “retries” setting, defaults to 0
* You can increase to a high number, e.g. Integer.MAX\_VALUE
* In case of retries,by default there is a chance that messages will be sent out of order.
* If you rely on key-based ordering, that can be an issue.
* For this you can set the setting while controls how many produce request can be made in parallel: max.in.flight.request.per.connection. default is 5. Set it to 1 if you need to ensure ordering ( may impact throughput)
* In Kafka >=1.0.0, there’s a better solution!

**Idempotent Producer:**

* The producer can introduce duplicate messages in Kafka due to network errors.



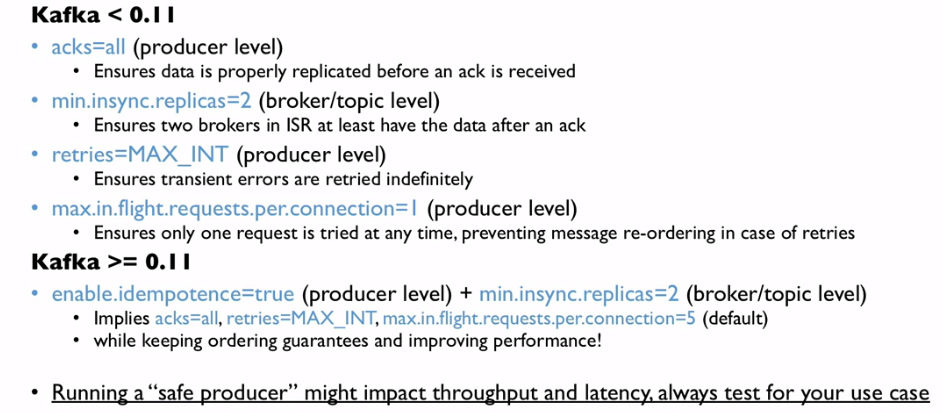
* In Kafka >=0.11, we can define a” idempotent producer” which won’t introduce duplicates on network error. It has produced id by using it identify the duplicate.



* Idempotent producers are great to guarantee a stable and safe pipeline.
* They come with retries = Integer.MAX\_VALUE
* max.in.flight.request= 1 (Kafka >=0.11 & 1.1)
* Max.in.flight.request= 5 (Kafka>= 1.1 - higher performance)
* It also come with acks=all, so that we can’t lose data.
* For using this we just need to set:

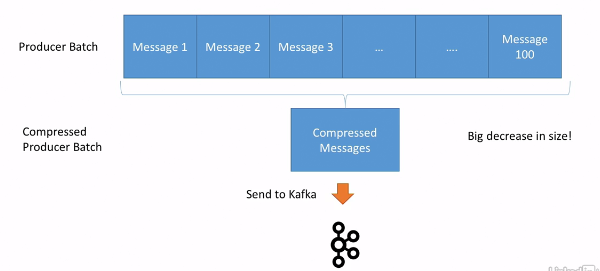
producerProps.put(“enable.idempotence”, true);

Summary:



**Producer Compression:**

* Producer usually send data that is text-based, for e.g. with JSON data.
* In this case, it is important to apply compression to the producer.
* Compression is enabled at the Produce level and doesn’t require any configuration change in the brokers or in the consumers.
* “compression.type” can be “none” (default), “gzip”, “Iz4”, “snappy”
* Compression is more effective the bigger the batch of message being sent to Kafka.
* Benchmarks here: [https://blog.cloudflare.com/squeezing-the -firehose/](https://blog.cloudflare.com/squeezing-the%20-firehose/)



* The compressed batch has the following advantage:
* Much smaller producer request size (Compression ratio up to 4x)
* Faster to transfer data over the network => less latency
* Better throughput
* Better disk utilization in Kafka (stored messages on disk are smaller)

Disadvantages:

* Producers must commit some CPU cycles to compression.
* Consumer must commit some CPU cycles to decompression.

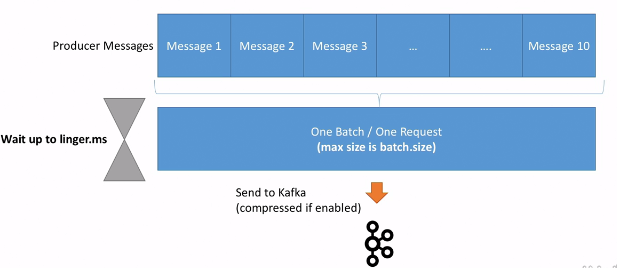
Overall:

* Consider testing snappy or Iz4 for optimal speed/ compression ratio.
* Find a compression algorithm that gives you the best performance for your specific data.
* Always use compression in production and especially if you have high throughput.
* Consider tweaking linger.ms and batch.size to have bigger batches, and therefore more compression and higher throughput.

**linger.ms and batch.size:**

By default, Kafka tries to send record as soon as possible.

* It will have up to 5 requests in flight, meaning up to 5 messages individually sent at the same time.
* After this if more messages have to be sent while others are in flight Kafka is smart and will start batching them while they wait to send them all at once.
* This smart batching allows Kafka to increase throughput while maintaining very low latency.
* Batches have higher compression ration so better efficiency.
* Linge.ms: Number of milliseconds a producer is willing to wait before sending a batch out. (default 0)
* By introducing some lag , we increase the chances of messages being sent together in a batch.
* So at the expense of introducing a small delay, we can increase throughput, compression and efficiency of our producer.
* If a batch is full before the end of the linger.ms period, it will be sent to kafka right away.



* Batch.size: Maximum number of bytes that will be included in a batch. The default is 16KB.
* Increasing a batch size to something like 32KB or 64KB can help increasing the compression, throughput and efficiency of requests.
* Any message that is bigger than the batch size will not be batched.
* A batch is allocated per partition so make sure that you don’t set it to a number that’s too high, otherwise you’ll run waste memory.
* We can monitor the average batch size metric using Kafka producer metrics

Producer Default Practitioner and how keys are hashed:

* By Default, keys are hashed using the “murmur2” algorithm.
* It is most likely preferred to not override the behavior of the partitioner but it I possible to do so using partitioner.class property.
* The formula is:

targetPartition = Utils.abs(Utils.murmur2(record.key())) % numParittions

Max.block.ms & buffer.memory:

* If the producer produces faster than the broker can take, the records will be buffered in memory.
* Buffer.memory=33554432(32MB) the size of the send buffer
* That buffer will fill up over time and fill back down when the throughput to the broker increases.
* If that buffer is full then the .send() method will start to block.
* Max.block.ms=60000 the time the .send() will block until throwing an exception. Exception are basically thrown when
* The producer has filled up its buffer.
* The broker is not accepting any new data
* 60 second has elapsed
* If you hit an exception hit that usually means your brokers are down or overloaded as they can’t respond to requests.