Kafka Producer

Producer Overview

There are many reasons an application might need to write messages to Kafka:

- recording user activities for auditing or analysis,
- recording metrics,
- storing log messages,
- recording information from smart appliances,
- communicating asynchronously with other applications,
- buffering information before writing to a database, and much more.

Those diverse use cases also imply diverse requirements:

- is every message critical, or can we tolerate loss of messages?
- Are we OK with accidentally duplicating messages?
- Are there any strict latency or throughput requirements we need to support?

❑ **credit card transaction processing**, it is critical to **never lose** a single message nor duplicate any messages.

Latency should be low but latencies up to 500ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

❑ A different use case might be to **store click information from a website**. In that case, **some message loss** or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience.

Kafka Producer

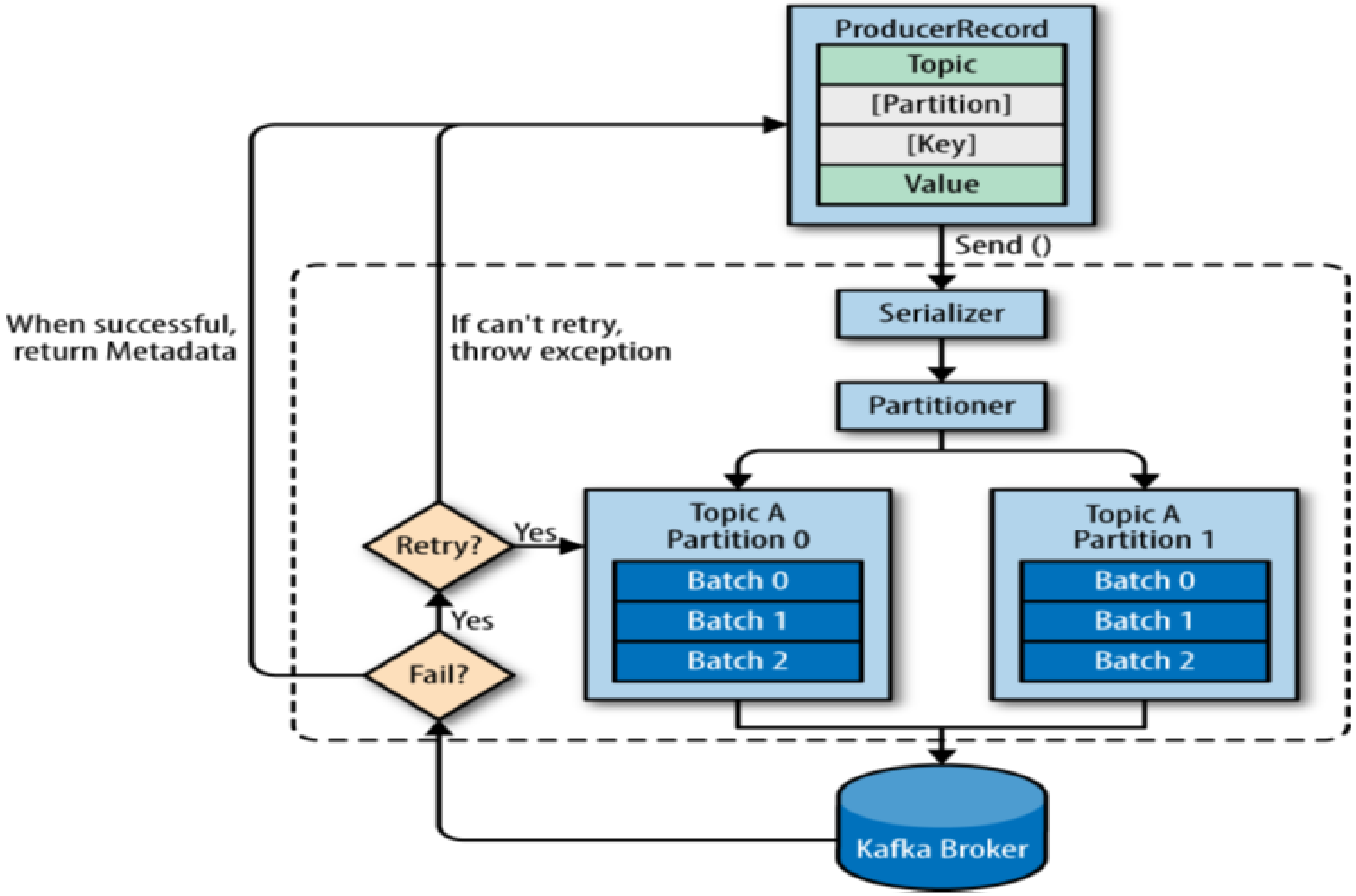
❑ Kafka client that publishes records to Kafka cluster

❑ The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances

❑ Producer has pool of buffer that holds to-be-sent records
background I/O threads turning records into request bytes and transmit requests to Kafka

❑ Close producer so producer will not leak resources

High-level overview of Kafka producer components



❑ We start producing messages to Kafka by creating a ProducerRecord, which must include the topic we want to send the record to and a value.

❑ Optionally, we can also specify a key and/or a partition.

❑ Once we send the ProducerRecord, the first thing the producer will do is serialize the key and value objects to ByteArrays so they can be sent over the network.

- ❑ Next, the data is sent to a partitioner. If we specified a partition in the ProducerRecord, the partitioner doesn’t do anything and simply returns the partition.

- ❑ If we didn’t, the partitioner will choose a partition for us, usually based on the ProducerRecord key[hashCode(key) % noOfPartitions]

- ❑ Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition.

- ❑ A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

❑ When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a RecordMetadata object with the topic, partition, and the offset of the record within the partition.

❑ If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

Kafka Producer Send, Acks and Buffers

❑ send() method is asynchronous
adds the record to output buffer and return right away
buffer used to batch records for efficiency IO
and compression

❑ acks config controls Producer record durability.
”all" setting ensures full commit of record, and is most
durable and least fast setting.

❑ Producer can retry failed requests

❑ Producer has buffers of unsent records per topic partition
(sized at batch.size)

Producer Acks

❑ Producer Config property acks (default all)

❑ Write Acknowledgment received count required from partition leader before write request deemed complete

❑ Controls Producer sent records durability

❑ Can be all (-1), none (0), or leader (1)

Acks 0 (NONE)

❑ Producer does not wait for any ack from broker at all

❑ Records added to the socket buffer are considered sent

❑ No guarantees of durability - maybe

❑ Record Offset returned is set to -1 (unknown)

❑ Record loss if leader is down

❑ Use Case: maybe log aggregation

Acks 1 (LEADER)

❑ Partition leader wrote record to its local log but responds without followers confirmed writes

❑ If leader fails right after sending ack, record could be lost
Followers might have not replicated the record

❑ Record loss is rare but possible

❑ Use Case: log aggregation

Acks -1 (ALL)

❑ Leader gets write confirmation from full set of ISRs before sending ack to producer

❑ Guarantees record not be lost as long as one ISR remains alive

❑ Strongest available guarantee

❑ Even stronger with broker setting min.insync.replicas (specifies the minimum number of ISRs that must acknowledge a write)

❑ Most Use Cases will use this and set a min.insync.replicas > 1

Kafka Producer: Buffering and batching

- ❑ Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight max.in.flight.requests.per.connection)

- ❑ The default value is 5

- ❑ The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).

❑ To reduce requests count, set linger.ms > 0

  -wait up to linger.ms before sending or until batch fills up whichever comes first

  -Under heavy load linger.ms not met, under light producer load used to increase broker IO throughput and increase compression

❑ buffer.memory controls total memory available to producer for buffering

  If records sent faster than they can be transmitted to Kafka then this buffer gets exceeded then additional send calls block. If period blocks (max.block.ms) after then Producer throws a TimeoutException

Batching by Size

❑Producer config property: batch.size
Default 16KB

❑Producer batch records
fewer requests for multiple records sent to same partition
Improves IO throughput and performance on both producer and server

❑If record is larger than the batch size, it will not be batched

❑Producer sends requests containing multiple batches
batch per partition

❑Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

Producer Buffer Memory Size

- ❑Producer config property: buffer.memory
  default 32MB

- ❑Total memory (bytes) producer can use to buffer records to be sent to broker

- ❑Producer blocks up to max.block.ms if buffer.memory is exceeded, if it is sending faster than the broker can receive, exception is thrown

Note :

batch.size   : By default, holds upto 16K per partition

buffer.memory : By default, holds upto 32MB for all partition buffers

Batching by Time and Size

❑ Producer config property: linger.ms
Default 0

❑ Producer groups together any records that arrive before they can be sent into a batch
good if records arrive faster than they can be sent out

❑ Producer can reduce requests count even under moderate load using linger.ms

❑linger.ms adds delay to wait for more records to build up so larger batches are sent
good brokers throughput at cost of producer latency

❑ If producer gets records whos size is batch.size or more for a broker’s leader partitions, then it is sent right away

❑ If Producers gets less than batch.size but linger.ms interval has passed, then records for that partition are sent

❑Increase to improve throughput of Brokers and reduce broker load (common improvement)

Compressing Batches

- ❑ Producer config property: compression.type
  Default 0
- ❑ Producer compresses request data
- ❑ By default producer does not compress
- ❑ Can be set to none, gzip, snappy, or lz4
- ❑ Compression is by batch
  improves with larger batch sizes
- ❑ End to end compression possible if Broker config "compression.type" set to producer. Compressed data from producer sent to log and consumer by broker

For high-throughput producers, tune buffer sizes

particularly buffer.memory and batch.size (which is counted in bytes). Because batch.size is a per-partition setting, producer performance and memory usage can be correlated with the number of partitions in the topic.

The values here depend on several factors: producer data rate (both the size and number of messages), the number of partitions you are producing to, and the amount of memory you have available.

Keep in mind that larger buffers are not always better because if the producer stalls for some reason (say, one leader is slower to respond with acknowledgments), having more data buffered on-heap could result in more garbage collection.

Custom Serializers

- ❑ Serialization is the process of converting an object into a stream of bytes.
- ❑ Deserialization is the process of converting bytes into desired data type.
- ❑ Kafka provides built in serializers and deserializers for a few data types(String, Long, Double, Integer, Bytes etc.,)
- ❑ If our requirement is not fit in with built in serializers, then we need to write custom one.
- ❑ Just need to be able to convert to/fro a byte[]
- ❑ Serializers work for keys and values
- ❑ value.serializer and key.serializer

How DefaultPartitioner works?

The below explains the default partitioning strategy:

❑ If a partition is specified in the record, use it - this can be used when we already know the partition.

❑ If no partition is specified but a key is present, choose a partition based on a hash of the key -

It can be used when we want to distribute the data based on a key. The following formula is used to determine the partition: hashCode(key) % noOfPartitions

❑ If no partition or key is present, choose a partition in a round-robin fashion - If we are not bothered which partition our data is going to, this strategy can be used.

Default strategies work well to start with. The problem comes when we want the data for the same customers to go to the same partition and we have used a composite key to partition data.

For example, and we have used a key which contains a customer id and a date. Now although the customer id is fixed, the date can change and we will end up with a different hash code.

Which means the data for the same customer will go to a different partition.

Producer Partitioning

❑ Producer config property: partitioner.class
org.apache.kafka.clients.producer.internals.DefaultPartitioner

❑ Partitioner class implements Partitioner interface

Producer Interception

❑ Producer config property: interceptor.classes
empty (we can pass an comma delimited list)

❑ interceptors implementing ProducerInterceptor interface

❑ intercept records producer sent to broker and after acks

❑ Interceptor API will allow mutate the records to support the ability to add metadata to a message for auditing/end-to-end monitoring.

KafkaProducer send() Method

❑ Two forms of send with callback and with no callback both return Future

Asynchronously sends a record to a topic

Callback gets invoked when send has been acknowledged.

❑ send is asynchronous and return right away as soon as record has added to send buffer

❑ Sending many records at once without blocking for response from Kafka broker

❑ Result of send is a RecordMetadata

record partition, record offset, record timestamp

❑ Callbacks for records sent to same partition are executed in order

KafkaProducer send() Exceptions

InterruptException - If the thread is interrupted while blocked

SerializationException - If key or value are not valid objects given configured serializers

TimeoutException - If time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc.

KafkaException - If Kafka error occurs not in public API.

KafkaProducer flush() method

- ❑ flush() method sends all buffered records now (even if linger.ms > 0)
  blocks until requests complete

- ❑ Useful when consuming from some input system and pushing data into Kafka

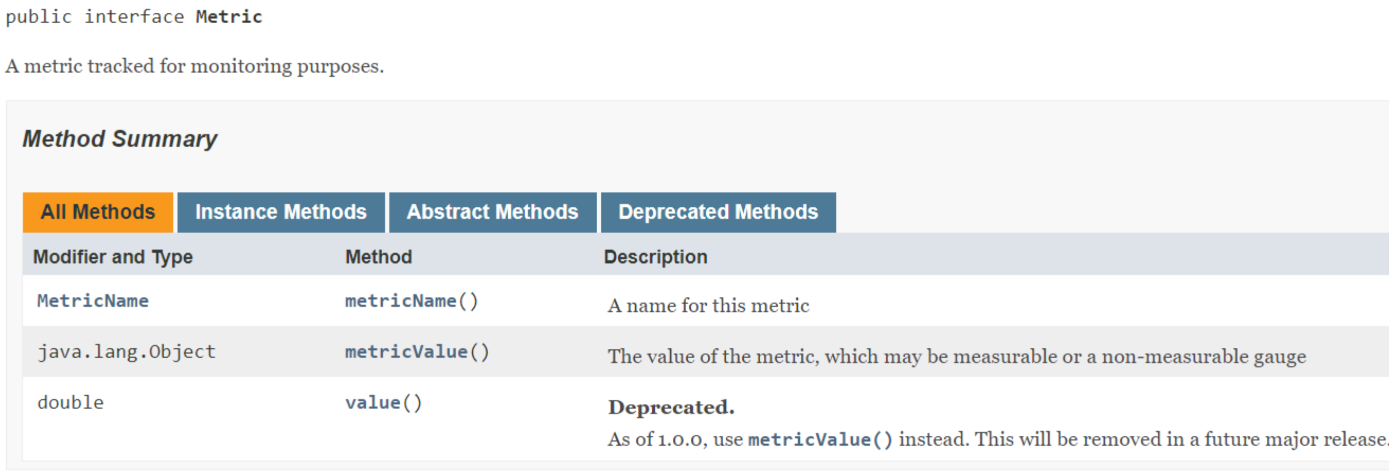
- ❑ flush() ensures all previously sent messages have been sent
  we could mark progress as such at completion of flush

KafkaProducer metrics() method

The metrics() method is used to get a map of metrics:

public Map<MetricName,? extends Metric> metrics().

This method is used to get a full set of producer metrics.

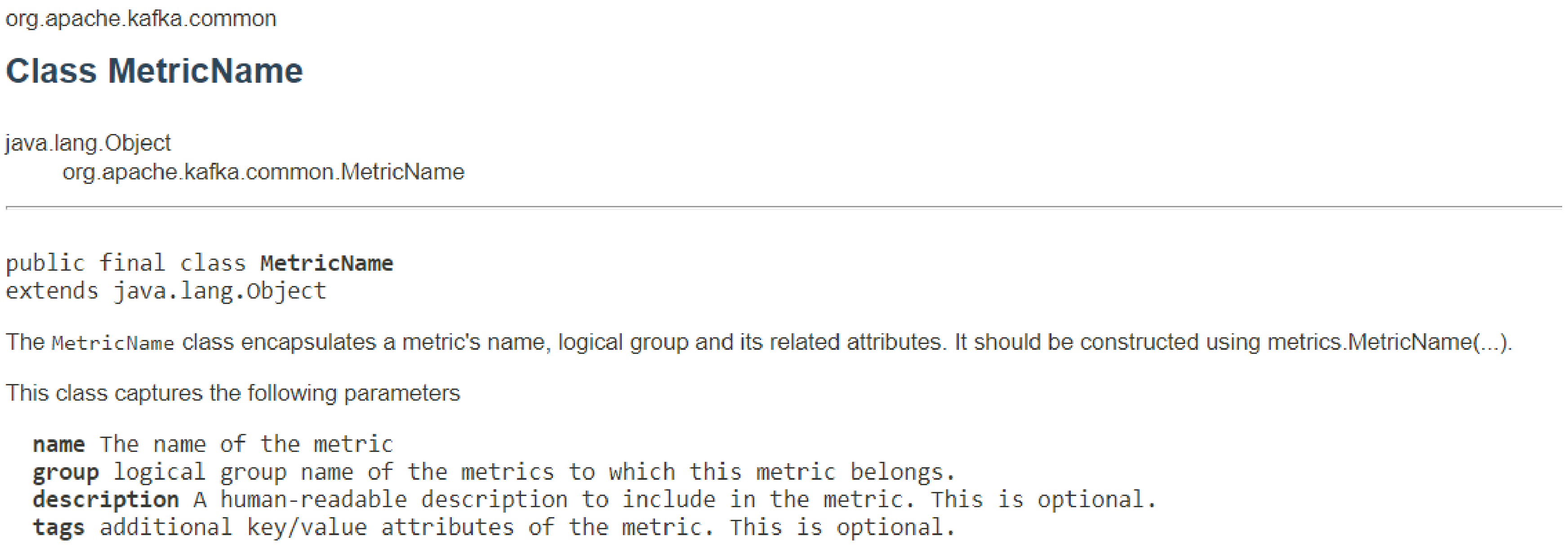


```
public interface Metric
```

A metric tracked for monitoring purposes.

Method Summary

Modifier and Type	Method	Description
`MetricName`	`metricName()`	A name for this metric
`java.lang.Object`	`metricValue()`	The value of the metric, which may be measurable or a non-measurable gauge
`double`	`value()`	**Deprecated.** As of 1.0.0, use `metricValue()` instead. This will be removed in a future major release.



org.apache.kafka.common

Class MetricName

java.lang.Object
    org.apache.kafka.common.MetricName

```
public final class MetricName
extends java.lang.Object
```

The `MetricName` class encapsulates a metric's name, logical group and its related attributes. It should be constructed using metrics.MetricName(...).

This class captures the following parameters

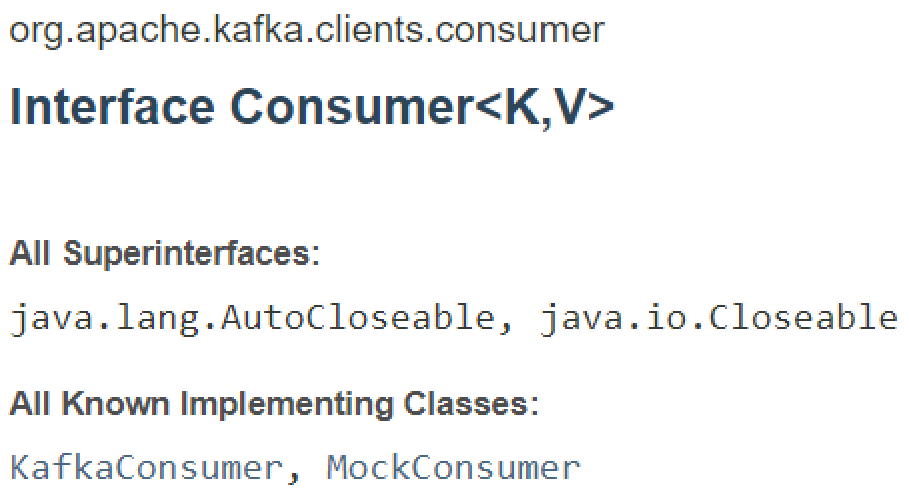
```
  name The name of the metric
  group logical group name of the metrics to which this metric belongs.
  description A human-readable description to include in the metric. This is optional.
  tags additional key/value attributes of the metric. This is optional.
```

Metrix via JMX

Kafka Consumer

KafkaConsumer

- ❑ A client that consumes records from a Kafka cluster.
- ❑ This client transparently handles the failure of Kafka brokers, and transparently adapts as topic partitions it fetches migrate within the cluster.
- ❑ This client also interacts with the broker to allow groups of consumers to load balance consumption using consumer groups.
- ❑ Consumer maintains TCP connections to Kafka brokers in cluster
- ❑ Use close() method to not leak resources
- ❑ Consumer is NOT thread-safe



org.apache.kafka.clients.consumer

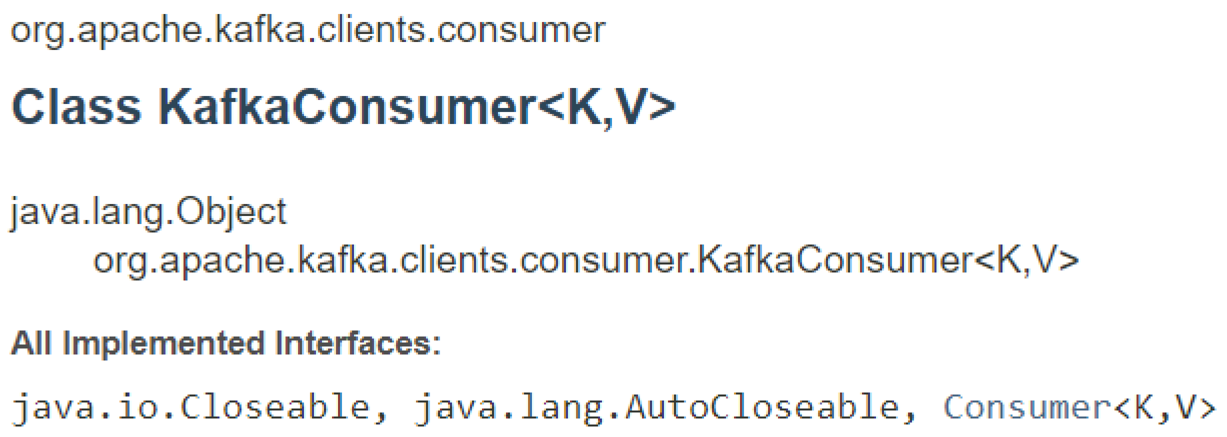
Interface Consumer<K,V>

All Superinterfaces:

`java.lang.AutoCloseable, java.io.Closeable`

All Known Implementing Classes:

`KafkaConsumer, MockConsumer`



org.apache.kafka.clients.consumer

Class KafkaConsumer<K,V>

java.lang.Object
    org.apache.kafka.clients.consumer.KafkaConsumer<K,V>

All Implemented Interfaces:

`java.io.Closeable, java.lang.AutoCloseable, Consumer<K,V>`

KafkaConsumer: Offsets and Consumer Position

❑ Kafka maintains a numerical offset for each record in a partition.

❑ This offset acts as a unique identifier of a record within that partition, and also denotes the position of the consumer in the partition.

❑ The position of the consumer gives the offset of the next record that will be given out.

- It automatically advances every time the consumer receives messages in a call to poll(long).

- Consumer *committed position* is last offset that has been stored to broker. If consumer fails, it picks up at last committed position

- Consumer can auto commit offsets (enable.auto.commit) periodically (auto.commit.interval.ms) or do commit explicitly using commitSync() and commitAsync()

KafkaConsumer: Consumer Groups and Topic Subscriptions

❑ Consumers organized into consumer groups (Consumer instances with same group.id).

❑ Pool of consumers divide work of consuming and processing records
Processes or threads running on same box or distributed for scalability/fault tolerance

❑ Kafka shares topic partitions among all consumers in a consumer group.

❑ Each partition is assigned to exactly one consumer in consumer group
Example topic has six partitions, and a consumer group has two consumer processes, each process gets consume three partitions

❑ Failover and Group rebalancing:
>> if a consumer fails, Kafka reassigned partitions from failed consumer to other consumers in same consumer group
>> if new consumer joins, Kafka moves partitions from existing consumers to new one

Consumer Groups and Topic Subscriptions

- ❑ Consumer group form single logical subscriber made up of multiple consumers

- ❑ Kafka is a multi-subscriber system, Kafka supports N number of consumer groups for a given topic without duplicating data

- ❑ To get something like a MOM queue all consumers would be in single consumer group : Load balancing like a MOM queue

- ❑ To get something like MOM pub-sub each process would have its own consumer group

KafkaConsumer: Partition Reassignment

- ❑ Consumer partition reassignment in a consumer group happens automatically

- ❑ Consumers are notified via ConsumerRebalanceListener, this will triggers consumers to finish necessary clean up.

- ❑ Consumer can use API to assign specific partitions using assign(Collection). This will disables dynamic partition assignment and consumer group coordination.

- ❑ Dead client may see CommitFailedException thrown from a call to commitSync(). Only active members of consumer group can commit offsets.

Controlling Consumers Position

- ❑ We can control consumer position. Moving consumer forward or backwards - consumer can re-consume older records or skip to most recent records

- ❑ Use consumer.seek(TopicPartition, long) to specify new position.
  consumer.seekToBeginning(Collection) and
  consumer.seekToEnd(Collection)

❑ Use Case

>> Time-sensitive record processing: Skip to most recent records

>> Bug Fix: Reset position before bug fix and replay log from there

>> Restore State for Restart or Recovery: Consumer initialize position on start-up to whatever is contained in local store and replay missed parts (cache warm-up or replacement in case of failure assumes Kafka retains sufficient history or you are using log compaction)

Storing Offsets : Managing Offsets

❑ For the consumer to manage its own offset we just need to do the following:

>> Set enable.auto.commit=false

>> Use offset provided with each ConsumerRecord to save your position (partition/offset)

>> On restart restore consumer position using kafkaConsumer.seek(TopicPartition, long).

❑ Usage like this simplest when the partition assignment is also done manually using assign() instead of subscribe()

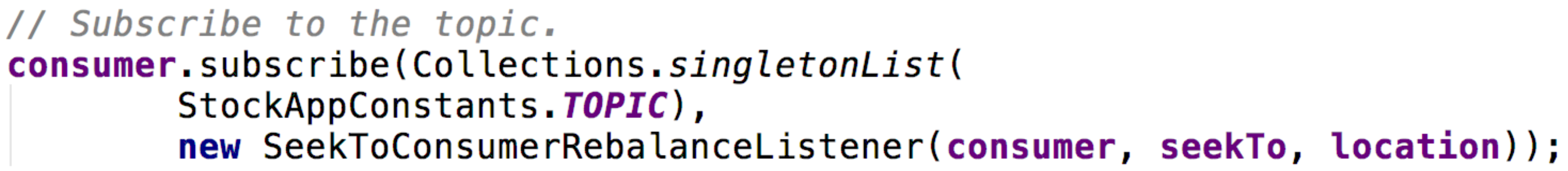
Storing Offsets : Managing Offsets

❑ If using automatic partition assignment, we must handle cases where partition assignments change

>> Pass ConsumerRebalanceListener instance in call to kafkaConsumer.subscribe(Collection,ConsumerRebalanceListener)

>> When partitions taken from consumer, commit its offset for partitions by Implementing ConsumerRebalanceListener.onPartitionsRevoked(Collection)

>> When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ConsumerRebalanceListener.onPartitionsAssigned(Collection)



```
// Subscribe to the topic.
consumer.subscribe(Collections.singletonList(
        StockAppConstants.TOPIC),
        new SeekToConsumerRebalanceListener(consumer, seekTo, location));
```

KafkaConsumer: Consumer Alive Detection

❑ Consumers join consumer group after subscribe and then poll() is called

❑ Automatically, consumer sends periodic heartbeats to Kafka brokers server

❑ If consumer crashes or unable to send heartbeats for a duration of session.timeout.ms, then consumer is deemed dead and its partitions are reassigned

KafkaConsumer: Manual Partition Assignment

- Instead of subscribing to the topic using subscribe, you can call assign(Collection) with the full topic partition list

```
String topic = “log-replication“;
TopicPartition part0 = new TopicPartition(topic, 0);
TopicPartition part1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(part0, part1));
```

- Manual partition assignment negates use of group coordination, and auto consumer fail over - Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion)

- We have to use assign() or subscribe() but not both

KafkaConsumer: Consumer Alive if Polling

❑ Calling poll() marks consumer as alive

>> If consumer continues to call poll(), then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every max.poll.interval.ms interval)

>> Not calling poll(), even if consumer is sending heartbeats, consumer is still considered dead

❑ Processing of records from poll has to be faster than max.poll.interval.ms interval

❑ max.poll.records is used to limit total records returned from a poll call - easier to predict max time to process records on each poll interval

Message Delivery Semantics

❑ **At most once**

Messages may be lost but are never redelivered

❑ **At least once**

Messages are never lost but may be redelivered

❑ **Exactly once**

this is what people actually want, each message is delivered once and only once

At-most-once Kafka Consumer (Zero or more deliveries)
At-most-once consumer is the default behavior of a KAFKA consumer.

To configure this type of consumer,

❑ Set ‘enable.auto.commit’ to true

❑ Set ‘auto.commit.interval.ms’ to a lower timeframe.

❑ And do not make call to consumer.commitSync(); from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

At-Least-Once Kafka Consumer (One or more message deliveries, duplicate possible)

To configure this type of consumer:

❑ Set ‘enable.auto.commit’ to true

❑ Set ‘auto.commit.interval.ms’ to a higher number.

❑ Consumer should then take control of the message offset commits to Kafka by making the following call consumer.commitSync()

Exactly-Once

Kafka ensures exactly-once delivery between producer and consumer applications through the newly introduced Transactional API.

```
producer.initTransactions();
try {
  producer.beginTransaction();
  producer.send(record1);
  producer.send(record2);
  producer.commitTransaction();
} catch(ProducerFencedException e) {
  producer.close();
} catch(KafkaException e) {
  producer.abortTransaction();
}
```

The code snippet above describes how we can use the new Producer APIs to send messages atomically to a set of topic partitions. It is worth noting that a Kafka topic partition might have some messages that are part of a transaction while others that are not.

So on the Consumer side, we have two options for reading transactional messages, expressed through the “isolation.level” consumer config:

read_committed: In addition to reading messages that are not part of a transaction, also be able to read ones that are, after the transaction is committed.

read_uncommitted: Read all messages in offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.

KafkaConsumer: Consumption Flow Control

❑ We can control consumption of topics using by using consumer.pause(Collection) and consumer.resume(Collection)

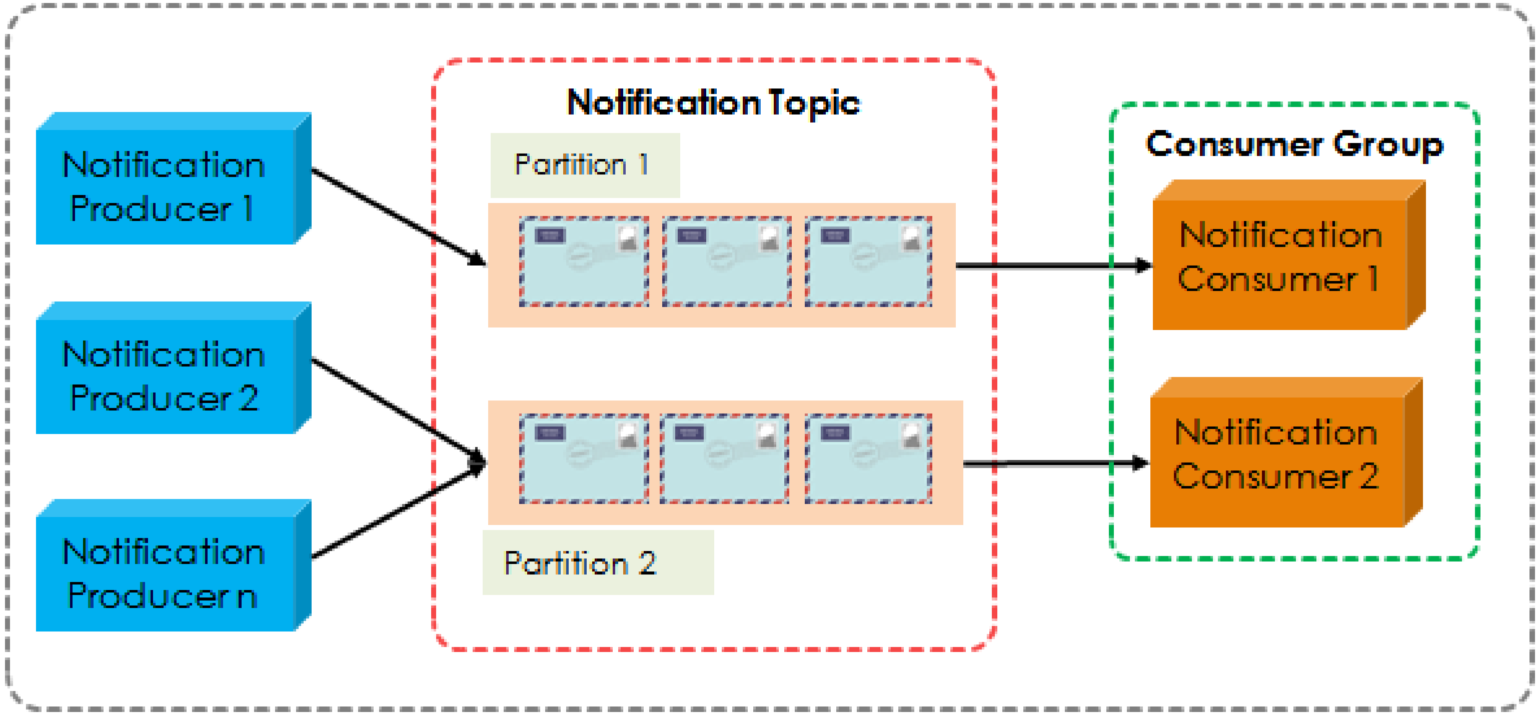
>> This pauses or resumes consumption on specified assigned partitions for future consumer.poll(long) calls

❑ Use cases where consumers may want to first focus on fetching from some subset of assigned partitions at full speed, and only start fetching other partitions when these partitions have few or no data to consume

>> Priority queue like behaviour from traditional MOM

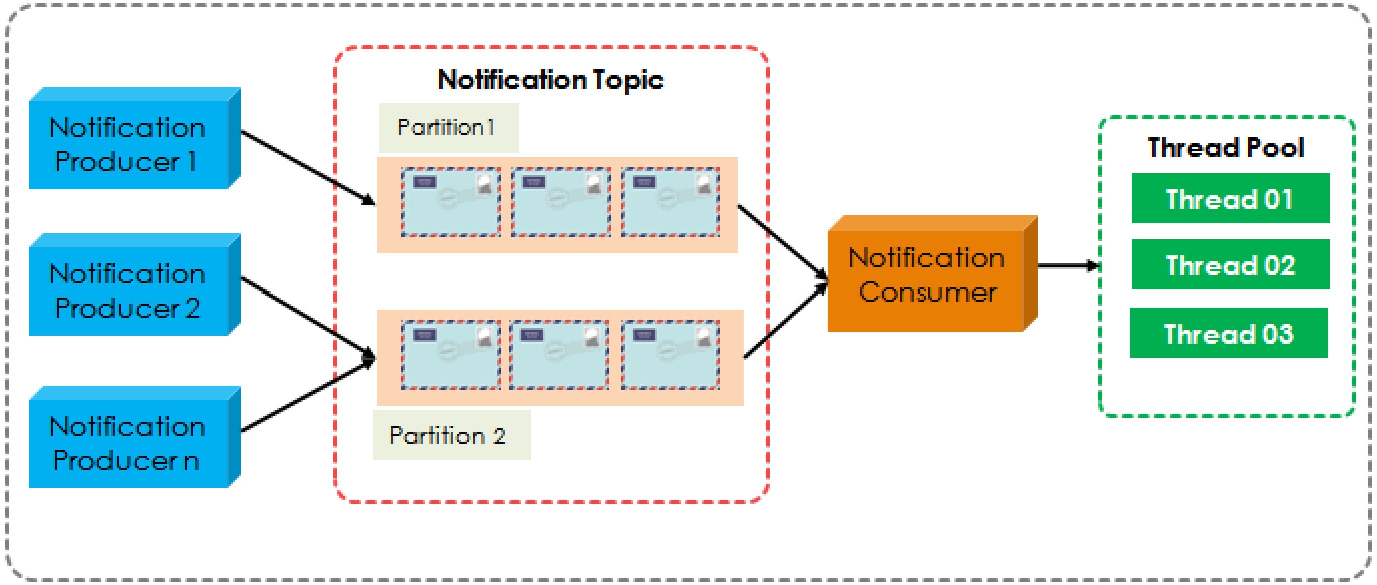
❑ Other cases is stream processing if preforming a join and one topic stream is getting behind another.

Multiple consumers with their own threads



Pros	**Cons**
Easy to implement	The total of consumers is limited the total partitions of the topic.
Implementing in-order processing on per-partition is easier.	More TCP connections to the brokers

Single consumer, multiple worker processing threads



Pros	**Cons**
Be flexible in scale out the number of processing thread	It’s not easy to implement in-order processing on per partition. Let’s say there are 2 messages on the same partitions being processed by 2 different threads. To guarantee the order, those 2 threads must be coordinated somehow.

AVRO

JSON

Advantages:

- ❑ Data can take any form (arrays, nested elements)
- ❑ JSON is a widely accepted format on the web
- ❑ JSON can be read by any language
- ❑ JSON can be easily shared over a network

Disadvantages:

- ❑ Data has no schema enforcing
- ❑ JSON Objects can be quite big in size because of repeated keys
- ❑ No comments, metadata, documentation

AVRO

- ❑ It is defined by a schema (written in JSON)
- ❑ Avro is JSON with a schema attached to it

AVRO

Advantages

❑ Data is fully typed
❑ Data is compressed automatically
❑ Schema (defined using JSON) comes along with the data
❑ Documentation is embedded in the schema
❑ Data can be read across any language
❑ Schema can evolve over time, in a safe manner (schema evolution)

Disadvantages

❑ Can't print the data without using avro tools (because it is compressed and serialised)

```
$ curl -i -X POST -H "Content-Type:
application/vnd.kafka.avro.v1+json" --data '{
  "value_schema": "{\"type\": \"record\", \"name\": \"User\",
\"fields\": [{\"name\": \"username\", \"type\": \"string\"}]}",
  "records": [
    {"value": {"username": "testUser"}},
    {"value": {"username": "testUser2"}}
  ]
}' \
http://localhost:8082/topics/avrotest
```

Note : Both $key_schema and $value_schema parameters are optional and provide JSON strings that represent Avro schemas to use to validate and serialize key(s) and value(s).

This sends an HTTP request using the POST method to the endpoint http://localhost:8082/topics/avrotest, which is a resource representing the topic avrotest.

The content type, application/vnd.kaka.avro.v1+json, indicates the data in the request is for the Kafka proxy (application/vnd.kafka), contains Avro keys and values (.avro), using the first API version (.v1), and JSON encoding (+json).

The payload contains a value schema to specify the format of the data (records with a single field username) and a set of records. Records can specify a key, value, and partition, but in this case we only include the value.

The values are just JSON objects because the REST Proxy can automatically translate Avro's JSON encoding, which is more convenient for your applications, to the more efficient binary encoding you want to store in Kafka.

The server will respond with:

```
HTTP/1.1 200 OK
Content-Length: 209
Content-Type: application/vnd.kafka.v1+json
Server: Jetty(8.1.16.v20140903)
{
  "key_schema_id": null,
  "value_schema_id": 1,
  "offsets": [
    {"partition": 0, "offset":0, "error_code": null, "error": null},
    {"partition": 0, "offset":1, "error_code": null, "error": null}
  ]
}
```

This indicates the request was successful (200 OK) and returns some information about the messages.

Schema IDs are included, which can be used as shorthand for the same schema in future requests.

Information about any errors for individual messages (error and error_code) are provided in case of failure, or are null in case of success.

Successfully recorded messages include the partition and offset of the message.

Primitive Types

The set of primitive type names is:

null: no value
boolean: a binary value
int: 32-bit signed integer
long: 64-bit signed integer
float: single precision (32-bit) IEEE 754 floating-point number
double: double precision (64-bit) IEEE 754 floating-point number
bytes: sequence of 8-bit unsigned bytes
string: unicode character sequence

```
{"type": "string"}
```

Avro Record Schemas

It is defined using JSON

It has some common files:

Name: Name of the schema
Namespace : (equivalent of package in java)
Doc: Documentation to explain the schema
Aliases: Optional other names for the schema
Fields :
  Name : Name of the field
  Doc: Documentation for that field
  Type: Data type for that field (can be a primitive type)
  Default: Default value for that field

Complex Types

Avro complex types are:

Enum
Arrays
Maps
Unions
Calling other schemas as types

Avro Complex Types:

Enums Example :

```
{"type": "enum", "name": "AccountType", "symbols" :  ["Savings" ,
"Current", "Loan"]}
```

Arrays Example:

```
{"type": "array", "items": "string",  "value": ["one", "two", "three"]}
```

Maps Example:

```
{"type": "map", "values": "int", "value": ["one": 1, "two": 2, "three": 3]}
```

Logical Types

Avro has a concept of logical types used to give more meaning to already exisiting primitive types:

date (int) - number of days since unix epoch (Feb 3rd 1970)

time-millis (long) - number of milliseconds after midnight 00:00:00.000

timpestamp-millis (long) - the number of milliseconds from the unix epoch (3 February 1970 00:00:00.000 UTC)

Example : Customer Signup timestamp:
{"name":"signup_ts", "type":"long", "logicalType":"timestamp-mills"}

Avro in Java

A **GenericRecord** is used to create an avro object from schema, the schema being referenced as "file / string".

It is not the most recommended way of creating Avro objects because things can fail at runtime, but it the most simple way.

Specific Record

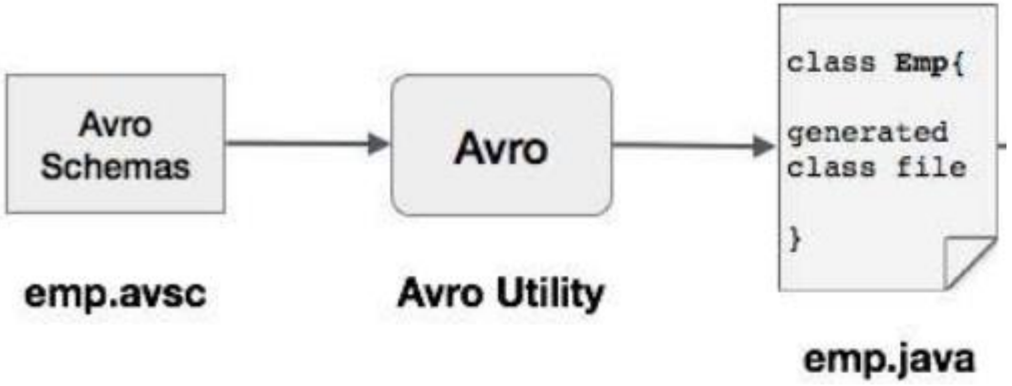
It is also an Avro object, but is obtained using code generation from an Avro schema.

There are different plugins for different build tools (gradle, maven,sbt etc.,)

Avro Schema -> Maven Plugin -> Generation Code

Specific Record

Compile the schema using Avro utility & Generate a java class



1. Download avro-tools-1.8.2.jar

2. Create emp.avsc

```
{
  "namespace": “demo.employee",
  "type": "record",
  "name": "emp",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "id", "type": "int"},
     {"name": "salary", "type": "int"},
     {"name": "age", "type": "int"},
     {"name": "address", "type": "string"}
  ]
}
```

3. java -jar <path/to/avro-tools-1.8.2.jar> compile schema <path/to/schema-file> <destination-folder>

Avro Reflection

We can use Reflection in order to build Avro schemas from the existing class.

This is a less common scenario but still a valid one. It is useful when we want to add some classes to Avro objects.

Existing Java Class -> Reflection -> Avro Schema and Object

Schema Evolution

Avro enables us to evolve our schema over time, to adapt with the changes from the business.

Example:

Initial version(v1) of the customer request contains First Name and Last Name. But, future version(v2) may require email, phone number etc.,

We should make the schema evolve without breaking programs reading our stream of data.

Schema Evolution

There are 4 types:

Backward : it is compatible change is when a new schema can be used to read old data.

Forward : it is compatible change is when an old schema can be used to read new data.

Full : which is both backward and forward

Breaking: which is none of those

Backward Compatible

We can read old data with new schema with a default value.
In case the field doesn't exist, Avro will use the default.

With backwards, we can successfully perform queries over old and new data using new schema.

Old schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"}
    ]}
```

New schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "phone", "type": "string",
           "default" : "000-000-000"}
    ]}
```

Forward compatible

We can read new data with the old schema, Avro will just ignore new fields.
Forward compatible is used, when we want to make a data stream evolve without changing our downstream consumers.

Old schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"}
    ]}
```

New schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "email", "type":“string"}
    ]}
```

Fully compatible

Only add fields with defaults.
Only remove fields that have defaults.
When writing you schema changes, most of the time
you want to target full compatibility.

Old schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"}
    ]}
```

New schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "phone", "type": "string",
           "default" : "000-000-000"}
    ]}
```

Not compatible

Here are examaples of changes that are NOT compatible:
Adding / Removing elements from an Enum
Changing the type fo a field (string => int etc.,)
Renaming a required field (field without defaultl)

Don't do that.

Retention period: 2 weeks

Assuming 100 messages per second.

Then 6000 messages per minute and 360000 per hour.

Assuming each message is of size 1kb then we need 360000 kb or 360MB storage per hour.

Assuming 2 weeks retention it will be around 120960 MB per 2 weeks i.e 120.96 GB of storage per 2 weeks.

JVM Heap Size

Make sure you allocate at least 6-8 GB of ram to JVM heap.

On an operating system, you should increase the file descriptor limit to anywhere between 100K-150K. This helps in increasing the Kafka performance.