Introduction to Apache Kafka

**Kafka** is a distributed messaging engine designed for high throughput environments.

Terminologies in Apache Kafka:

* Kafka works on Pub-Sub model (Publishers and Subscribers). These are called Producers and Consumers in Kafka world.
* At the core, Kafka has Brokers. Broker is a software process/daemon service that runs on a machine, which is responsible for passing on the messages to respective consumers. Broker uses file system to persist messages.
* Kafka Producer produce messages into channels called as Topics, to which, clients can subscribe in order to receive messages.
* A group of Brokers (on same or different machines) is known as a cluster.

------------------------------ Distributed Systems --------------------------------

Characteristics of a distributed system:

Independent Worker/Nodes/Workernodes: Controller, Leaders, and Followers.

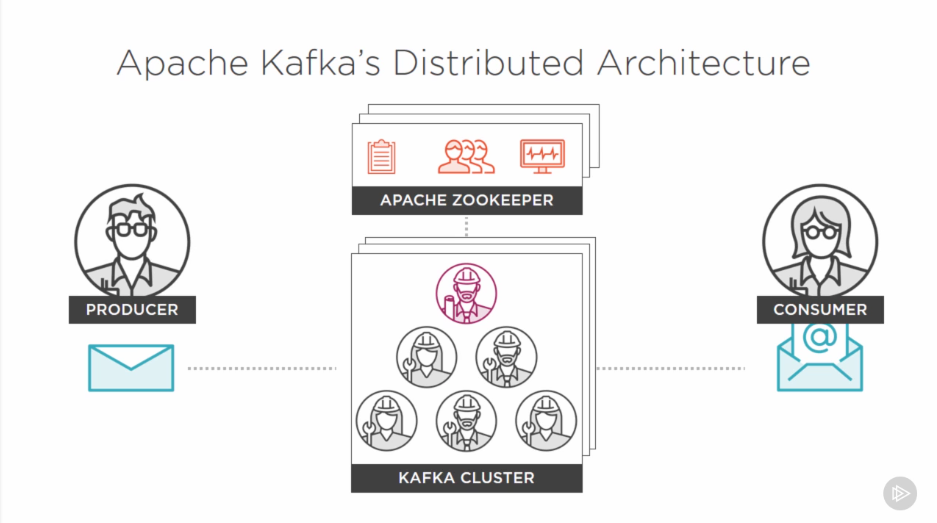
Reliability through replication

Consensus-based communication

Apache Zookeeper is responsible for these things:

1. Inter-worker communication
2. Worker node membership and naming
3. Global configuration management (Maintaining meta-data of the cluster)
4. Leader election
5. Setting up and maintaining global health and meta data for the brokers in a cluster.

All these add to the scalability and reliability of systems.



Kafka Topics are logical entities.

Use Central Kafka Abstraction

Physically represented as a log file

A topic can span the entire cluster and hence is visible to all brokers in a cluster, also it can be limited/visible to a few brokers only. This helps in achieving reliability and fault tolerance.

Producer/Consumer do not care what/where/how messages are stored inside of a broker.

Producer produces a message and is put into a topic (which is append only) in an ordered sequence (by time) and stores immutable facts as events. If an invalid fact is received, kafka then waits for a more correct message to the previous one, before it processes them. This style is called event sourcing.

Every message has a

1. Timestamp
2. Reference able identifier
3. Payload (binary)

How consumers maintain autonomy?

Using message offset. It is a placeholder which stores position of last read message position. Message offset helps consumers to read messages at their own pace and process them independently.

The Offset is maintainers by a consumer ONLY.

Once new messages are added, events are fired and consumers can act on these events to retrieve new messages

Kafka overcomes slow consumer issues by retaining messages for a long period of time inside broker which is configurable in the Message Retention Policy. KAFKA RETAINS ALL PUBLISHED MESSAGES REGARDLESS OF CONSUMPTION.

Retention period is configurable. Default is 168 hours/7 days.

Retention period is per-topic basis.

Retention is also limited/bounded by available storage

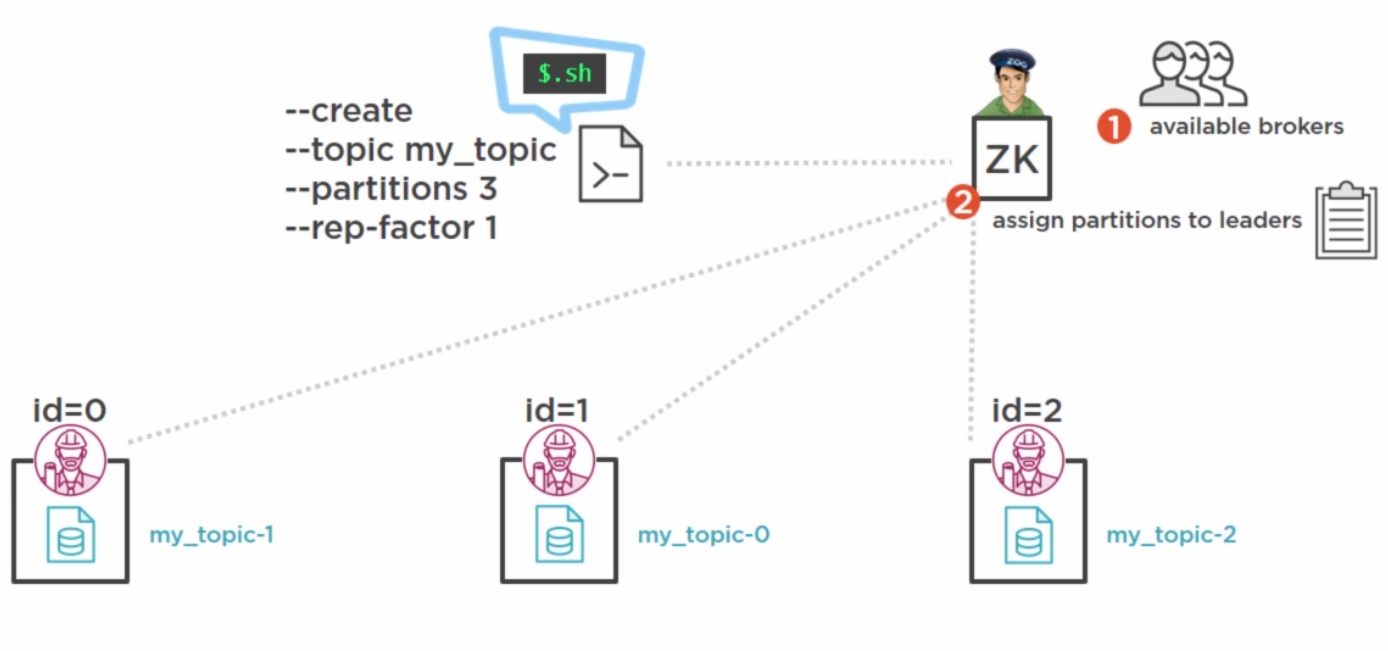
**Partitions:**

Partitions are the physical representation of topics. A topic can have multiple partitions. Every partition carries its own append-only log file. The logs for every apache cluster are stored in **/tmp/kafka-logs/<topic\_name> - <partition\_number>**

When using single partition: Can result in scalability issues. (Because one partition is limited in terms of computation, storage, IO constrains, etc.) A partition cannot be split across multiple servers.

When using multiple partitions (hence results can be handled by multiple broker nodes): Every partition maintains a mutually exclusive log of events. Every partition gets unique messages from a kafka producer, this increases parallelism. All logs are time-ordered sequence of events.

**How Kafka producer splits messages?**

A partitioning scheme is used for this. If nothing specified, producer uses Round Robin. 

Zookeeper receives a request to create a topic with multiple partitions. These partitions are then created and leader is assigned. ZK also maintains metadata and a subset of this metadata is maintained by every partition, which is used by producers to send messages to the appropriate broker.

**How Kafka consumers consume messages?**

The consumer enquires the ZK, which broker owns which partition. **Consumer also gets some additional meta data from the ZK which affects how consumer consumes the data from the brokers. (Particularly when there are large groups of consumers sharing the workload).** If a consumer consumes from different partitions, the messages are received in different orders and hence the consumer has to manage the ordering of messages. Multiple partitions are great for scalability.

**Drawbacks of multiple partitions:**

1. Since ZK has to manage the meta-data and config, every partitions needs creation of new entries in ZK, which leads to memory constraints, etc.
2. Message ordering might become very complex. If a consumer needs global message ordering, it becomes extremely complex to handle and the consumer has to manage ordering.
3. More the partitions, longer the leader fail-over time.

**How are partitions fault-tolerant? Potential threats in a multi-broker system where one broker is responsible for one or more partitions are**

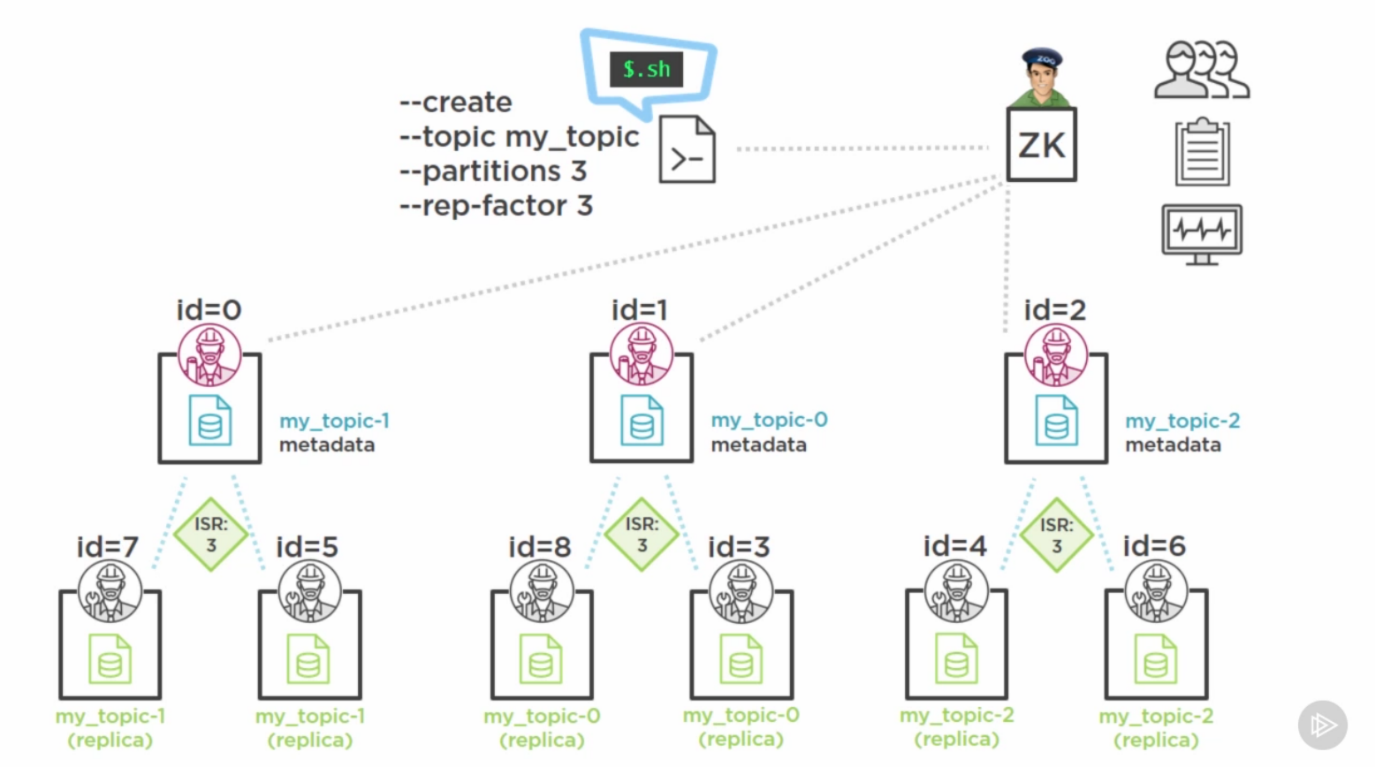
1. Broker Fails
2. Network issue
3. Disk Fail

This is the job of Zookeeper. In case a broker fails, Zookeeper is responsible for spawning/assigning a new broker. If there is no redundancy between the nodes, there will be some data loss (at the time when previous broker goes down and a new one comes). Kafka is shipped with a property to do replication in order to prevent from data loss. This property is responsible for replicating data across cluster. This command to create a topic with some defined replication factor is:

~$ bin/kafka-topic.sh --create --topic my\_topic --zookeeper localhost:2181 --partition 3 **--replication-factor 1**

**Why use replication should be used?**

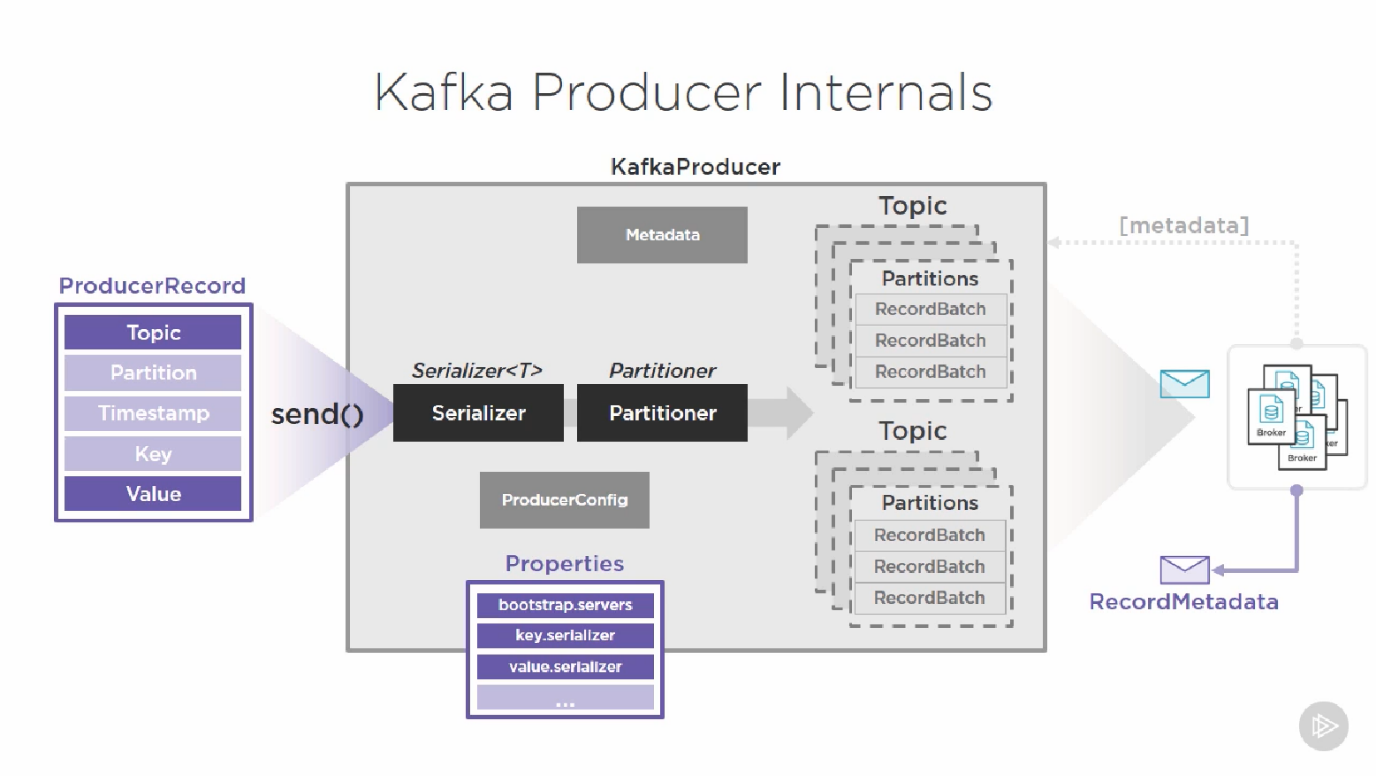
1. Enables reliable work distribution
   1. Messages are stored redundantly
   2. Makes the cluster more Resilient and Fail-tolerant, hence mitigates the risk of data loss
2. What replication guarantees to save?
   1. N-1 broken failure
   2. 2-3 minimum brokers recommended
3. Set at topic level

**How does it look while working?**

It is the job of leader to assign peer brokers in order to achieve replication. When all the data has been copied and a full-synchronized replica set is in place, it is reported to the entire cluster that the number of ISR (In-sync replicas) are up and are equal to the number replication factor for that topic and each partition within it. If a peer does not come up, Manual Intervention is needed (vigilant monitoring and compensatory actions are needed).

In case a broker goes down, ISR decreases in number, but replicas remain the same. The sequence is maintained. If the leader goes down, other broker will take over and become the leader.

# Producer:



Kafka producers need three config items:

1. bootstrap.servers - Cluster membership: partition leaders, etc.
2. key.serializer - For message serialization and deserialization
3. value.serializer - For message serialization and deserialization

(Any serializing scheme can be used. Most common is StringSerializer.)

In producer, there is a type called ProducerConfig. Props are used to instantiate an object of ProducerConfig class. This object is responsible for initialization internal values of serializations classes and to let them know that the type of data produces is string. It provides a type-safe contract between instance of kafka-producer and the message spec it is configured to produce. This contract is extended to the consumer along with the message specification.

How does Producer produce?

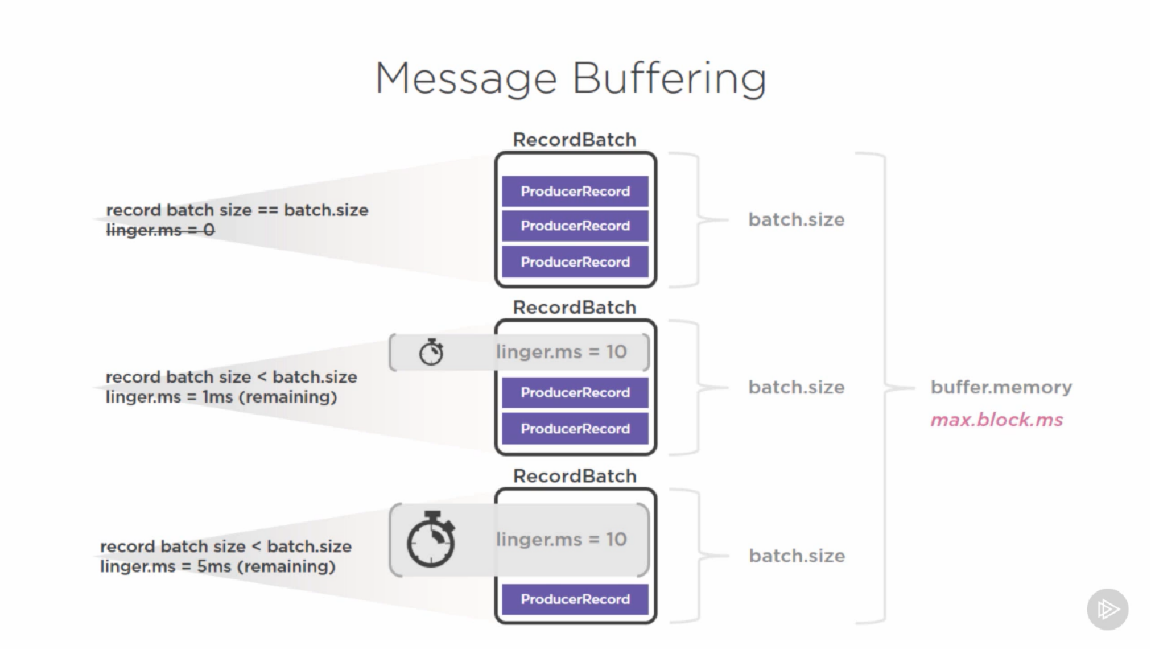
THERE IS NO MESSAGE CLASS. A producer produces ProducerRecord. In order it to be valid; it needs two values (minimum). These are topic and value. (Optional are partition, timestamp, key).

1. Topic: topic to which the messages are produced
2. Value: The message content, which matches the serializer type of value. In case there serialization class is different from the value, the producer will generate a runtime serialization exception. Type spec of a Producer must match the serialization type of key and value, otherwise throws exception.
3. Partition: Specific partition within a topic to send ProducerRecord
4. Timestamp: the Unix timestamp applied to the record. These are of two types:
   1. CreateTime: TS is taken when a message is produces hence producer-set TS
   2. LogAppendTime: TS is taken when a message is appended to commit log hence broker-set TS.
5. Key: It is a value, which is, when present, will determine how and to which partition within a topic the kafka producer will send the message to.   
   **BEST PRACTICE: DEFINE A KEY**
   1. Ups:
      1. Additional info in the message, which can be used to make processing decisions.
      2. It can influence the manner how messages are routed to the partitions
   2. Downs:
      1. Additional overhead
      2. Depends on the serializer used.

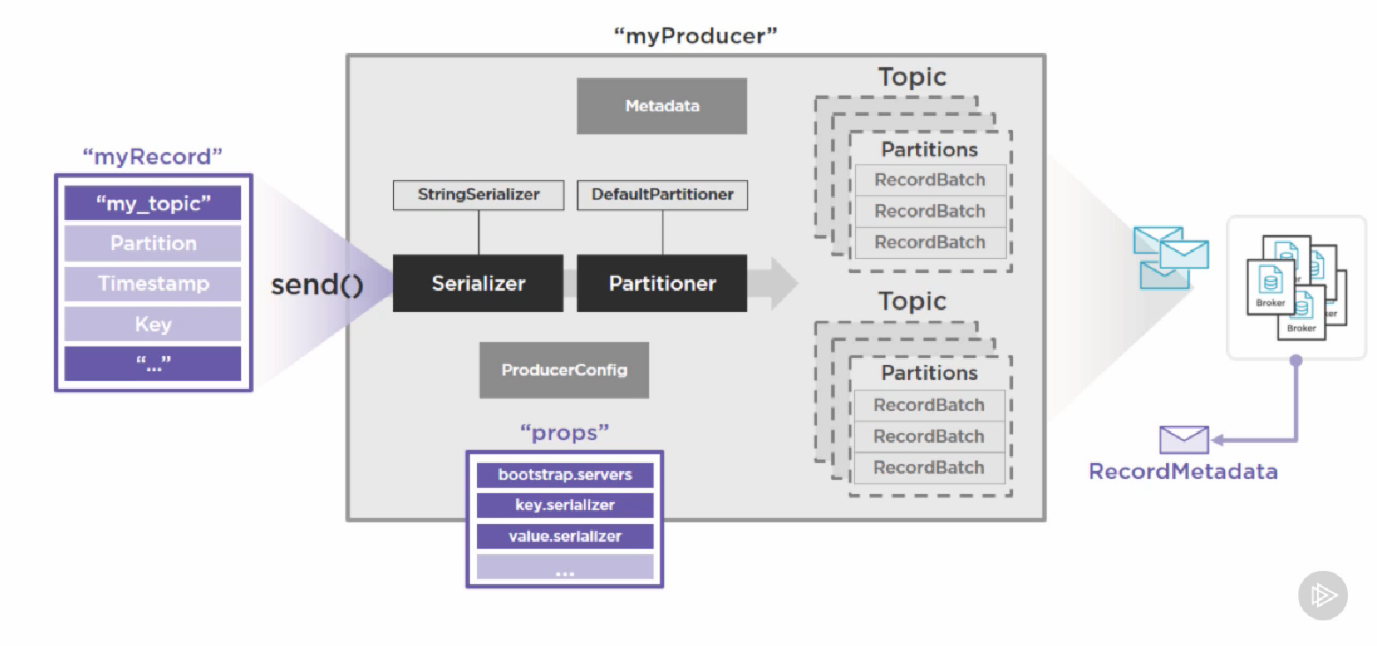
**What happens on calling the send method?**

The producer will reach out to the cluster, using the bootstrap servers list to discover the cluster membership. The response comes as metadata that contains topic, their partitions, and their managing brokers. This is used to instantiate a meta-data object inside the producer, which is used to keep the information of the cluster throughout the producer lifecycle. This meta object is kept fresh! A pseudo-processing pipeline is then engaged inside the producer, it has the following steps:

1. Serializer - Here the record is passed through the serializer using the configured serializer. (We are using string)
2. Partitioner - Which partition to send the record to? Here the producer can employ different partitioning strategies that are passed to it in the ProducerRecord and the info it has regarding cluster membership (this is at the bottom in detail). (For us, since we are providing a key, default strategy will be used)
3. RecordAccumulator - in-memory queue-like data structure. This is where the messages come after Partitioner. It is a low-level object having a lot of complexity. It gives the producer the ability to micro-batch the records that are intended to be sent at high volumes and frequencies. Once the ProducerRecord has been assigned to a partition, it is handed over to a record accumulator where it will be added to a collection of Record-Batch objects for each Topic-Partition combination needed by the producer instance. These batches are then sent to the broker. These batch sizes are determined at the producer level via some adv settings. Examples are:
   1. Each RecordBatch has limit to how many ProducerRecords can be buffered. This is not determined by number, but a config setting called “batch.size” whose value contains the max **number of BYTES** a RecordBatch can contain.
   2. On-top, there’s a “buffer.memory” prop which determines the max memory to be used by buffer.
   3. If the above threshold is met, then “max.block.ms” prop is used to determine how many seconds the send message will be blocked for? This blocking contingency is intended to force backpressure on the thread that is responsible for sending more ProducerRecords onto the buffer. The idea is to block so that existing contents are transmitted and buffer is freed-up and enable more records to be en-queued.
   4. When records get sent to a RecordBatch, they will wait around until one of the two following things happen:
      1. If the Total of a RecordBatch becomes equal to the “batch.size”, the records are sent immediately. This optimizes the overhead associated with transferring the PageCache bytes to the network socket. (This is one great power of micro batching). In parallel, other records are being dispatched to other accumulators and other buffers.
      2. The second threshold is a config setting called “linger.ms”, the number of ms a partially filled buffer should wait before it is sent. For high-frequency partitions, whose buffers are being filled rapidly, the “linger.ms” prop does not come into play.



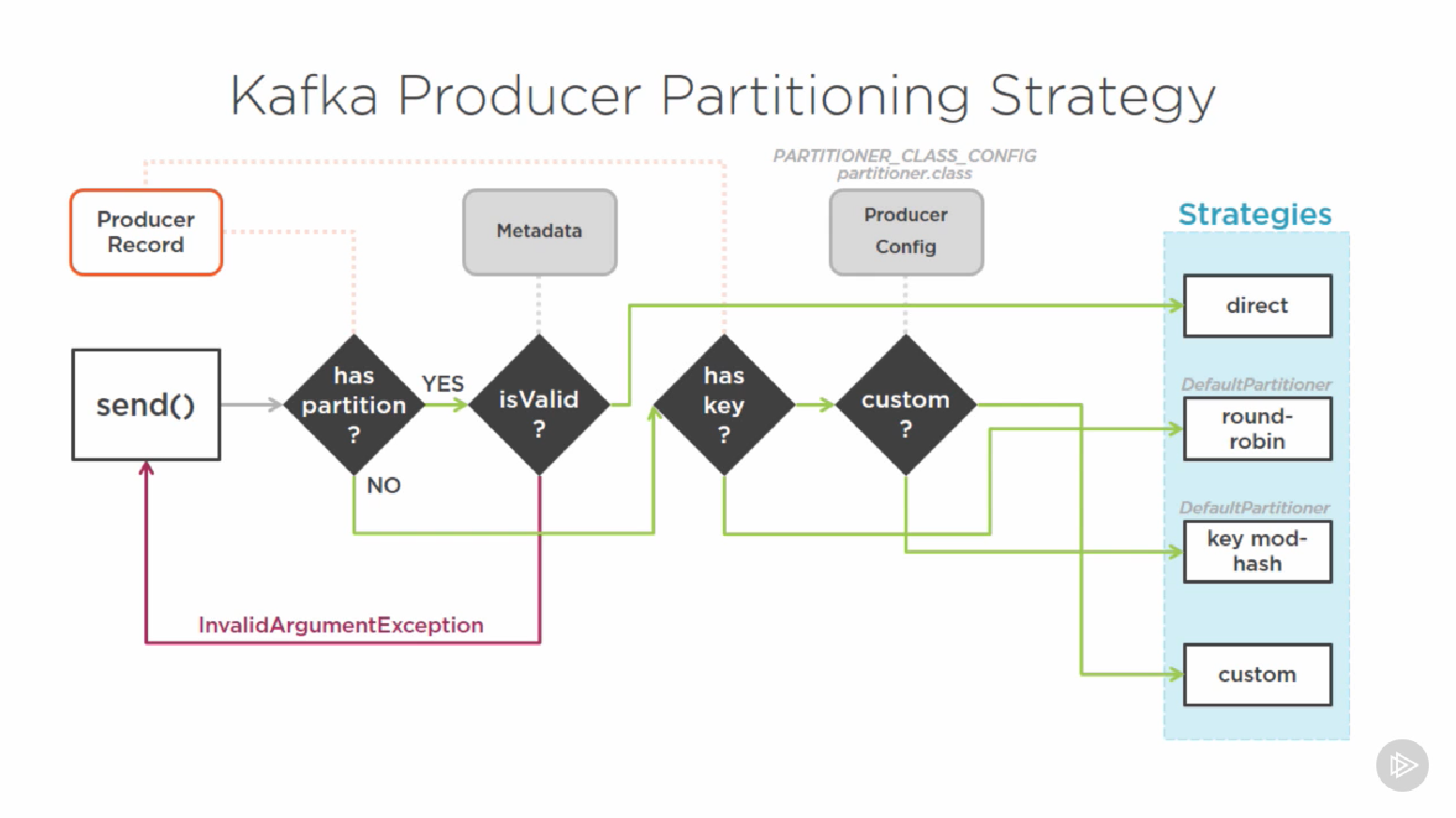
1. After this, the messages are finally dispatched to the broker and the response comes as a RecordMetadata containing info about records if they were received successfully or not.



**How producer distributes messages to partitions?**

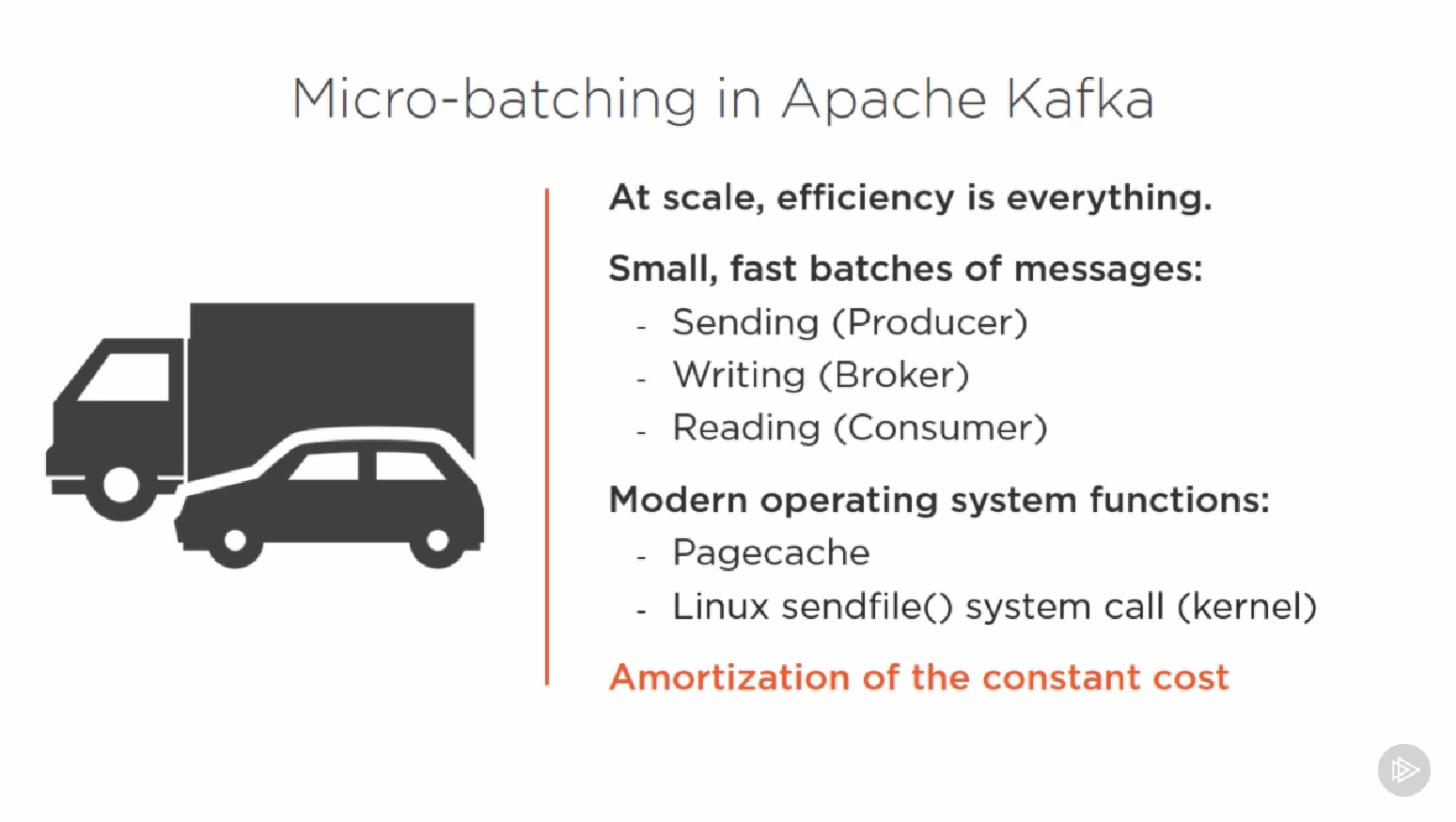
Between the time the send() is invoked and the time is received by the broker. Serialization is done and then at Partitioning, there are 4 major strategies. To resolve which strategy to use, these steps happen:

1. The kafka producer looks at the ProducerRecord contents if it has a partitions and if the partition is valid? Validity of partition is checked via the Meta-Data. In-case the partition is unavailable or is invalid, the application will throw InvalidArgumentException. Otherwise, the producer will add the ProducerRecord object to a specific partition buffer for the topic, where it will, on a separate thread, await the actual send to the broker leader of that specified partition.
2. If no partition is specified, the kafka producer checks if a key was provided in the ProducerRecord.
   1. If the answer is no, then message will be routed using a round-robin strategy which evenly distributes the messages across all partitions.
   2. If the answer is yes, whether a custom, non-default, partitioner class was provided as part of the configuration properties provided to instantiate the kafka producer.
      1. If there is nothing provided Partitioner Interface (this is default), a key-based partitioning scheme is used to route the messages (provided as a default impl for the partitioner iface). This key-based partitioning is done using a murmur-based mod-hash (mod value is the number of partitions).
      2. Sometimes a custom key-based partitioning scheme is needed. This is when a custom partitioner implementation is developed and added to the classpath and specify the class as partitioner.class.



Micro-batching in kafka

Every time we send/receive/persist resource overhead is incurred. In high throughput systems, this affects the performance, reliability, and overall throughput of the system. More overhead = less efficient. Kafka micro-batches messages on the producer, broker, and consumer, whenever they are sent/received/persisted. Kafka is designed to rapidly queue batched-up requests to send/persist/read in flexibly bound memory buffers that can take advantage of modern day OS features like PageCache and Linux SendFile()



**How does kafka ensure message delivery?**

There are various props, which are set at producer level.

1. Producer can specify the level of acknowledgement it expects from the broker. This prop is known as “acks”. Options are:
   1. 0: Fire and forget -> No ack whatsoever by broker. It is fastest but is not reliable; esp when there’s an issue with broker that prevents logging. Supported for lossy streams.
   2. 1: Leader acknowledged -> Leader acknowledges. This offers great balance of performance and reliability providing the cluster settings employ appropriate replication.
   3. 2: replication quorum acknowledged -> Most reliable, but non-performance. However, this depends on replication topology and possible changes in cluster membership.
2. Broker responds with an error:
   1. Perform reties using “reties” prop.
   2. “retry.backoff.ms” allows time-period between reties.

**How does kafka ensure ordering?**

Message orders are only preserved in a given partition. If the producer sends the messages in a specific order the broker logs them in same order and consumer reads the log in that order.

Message order by partition in multiple partition system

* There is not global order of messages across partitions; hence, it is the responsibility of the consumer to take care of ordering.

Errors make things worse in ordering

* If “retries”, “retry.backoff.ms” is set, there’s a possibility that message is sent and ack is not received, causing a retry to happen, but before the retry can be sent, the second message can be sent and successfully succeed. The only way to prevent this, but at cost of high throughput is the “max.in.flight.request.per.connection” prop to 1. This tells producer, at any time, only one request can be made.

Delivery semantics

* This is determined by the combination of above two factors. Achievable configs are At-most-once, at-least-once, and only-once message delivery assurance, but only with a design, that carefully considers the settings available at all three component members of the system (Producer, Broker, Consumer).

**Things not covered in this section**

1. Custom Serializers: Why and how?
2. Custom Partitioners: Why and how?
3. Asynchronous Send
4. Compression
5. Advanced Settings

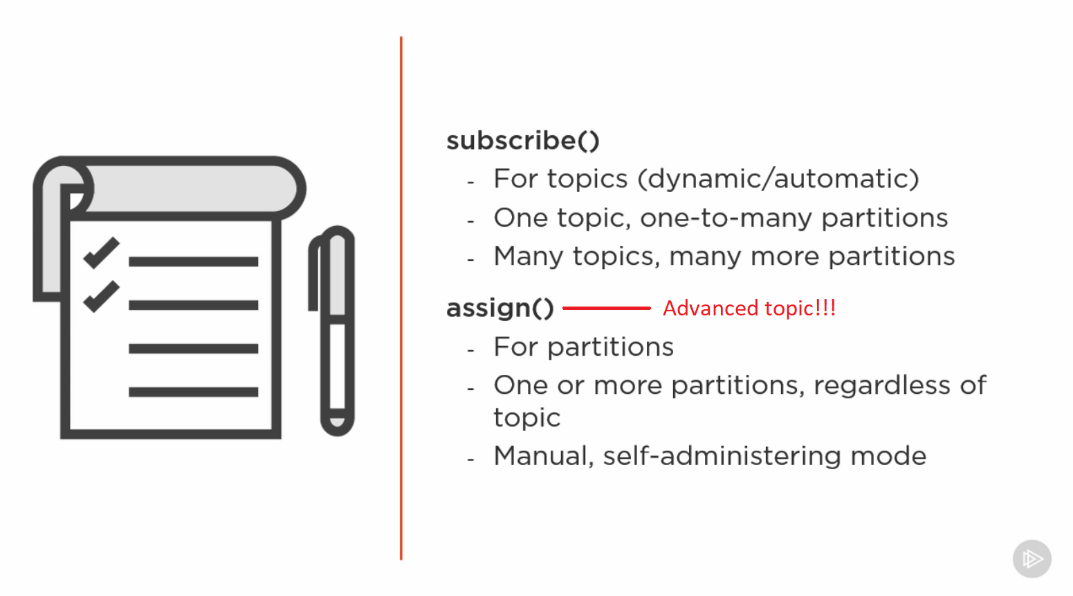
# Consumers

Similar to KafkaProducer, KafkaConsumer class is responsible for making a consumer. The Kafka consumer class needs minimum three properties to instantiate:

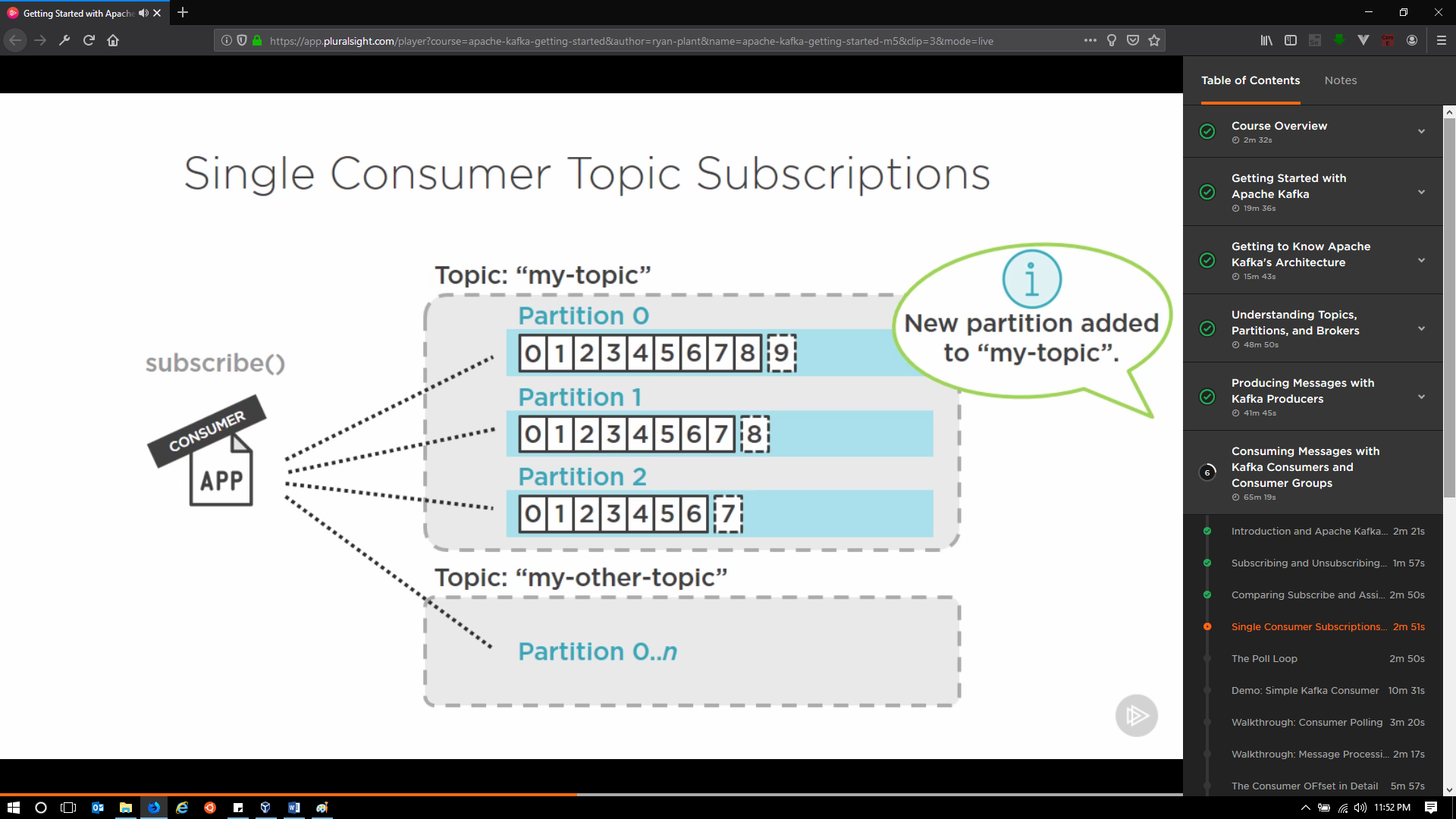
1. bootstrap.servers - Cluster membership: partition leaders, etc.
2. key.deserializer - For message serialization and deserialization
3. value.deserializer - For message serialization and deserialization

**How to subscribe to topic programmatically?**

An object of KafkaConsumer class has a method called subscribe which is used to subscribe to topics. It can take collection of topic names or a pattern (naming pattern followed by topics). CALL TO SUBSCRIBE ARE NOT INCREMENTAL. To unsubscribe, there’s a method called unsubscribe (which does not require args). Other way to unsubscribe is to pass empty string array to subscribe method. We can subscribe to individual partition as well!



When a subscriber calls the **subscribe method**, it constantly polls the topic (and thus the partitions) in the topics for new messages. A class called SubscriptionState manages the partition management of partitions. If a change is made to the partitions, this class is notified and it determines if the consumer need to account for the new partition or now.



In contrast to this, the **assign method** gives complete control of which partition to get messages from. (Similar to hard-coding partitions list for a consumer). It does not care if new partitions are added, although it gets the metadata.

**What is Poll Loop? ( poll( <arg: time in ms> ) )**

Poll is the primary function of the Kafka Consumer. Its job is to continuously poll brokers for new messages. Poll is a single API for handling all Consumer-Broker interactions (Not limited to only messages).

Poll Loop needs to go inside an actual loop (while, for). No interaction happens until the poll method is called. It’s a good idea to move poll() into a try .. catch block as it performs network operations and needs to be closed.

When assign method is used, the topic names and partitions passed to the method are stored in SubscriptionState which acts as a single source of truth as to which topics/partitions a consumer has subscribed to. Most of the things happening inside of the Consumer, go through the SubscriptionState Object. This object also works with ConsumerCoordinator to manage the message offsets.

When poll() is called, the bootstrap server to pinged to get the metadata about the cluster. This is where everything starts. The fetcher component acts as the responsible object for most of the communication between the Consumer and the Broker, but it’s not directly responsible for the communication. Inside of fetcher there are multiple components that contact the cluster. The communication is actually taken care by the “Consumer Network Client”. This component is responsible for transmitting/receiving the TCP Packets. **The consumer sends heartbeats, hence the cluster knows which are still connected**. Initial request is made to get the metadata of the cluster an this metadata is used to instantiate an internal metadata object inside of the KafkaConsumer. When the consumer has knowledge of the cluster metadata, the ConsumerCoordinator can do its job of coordinating b/w the cluster and the consumer. It can perform two major operations now:

1. To be aware of the partition changes and automatic/dynamic partition assignments and notification of assignment changes to SubscriptionState object.
2. To commit offsets to the cluster and provide confirmation to update the SubscriptionState object, that the SubscriptionState is always aware of the status of partitions/topics.

The fetcher needs to know which topics/partitions it should be asking for to get the data. It gets this info from SubscriptionState object. Once the fetcher has this data, it will poll the server for new messages. Here the argument passed to poll function is used. **It is the number of ms the network client will spend polling the cluster for messages before returning**, hence determines the minimum time each retrieval time will take. When the timer is elapsed, a batch of records are returned and added to an in-memory buffer for parsing, deserialization, and grouped into consumer records for particular topics/partitions and are put for processing.

**There is only one single-threaded poll loop per kafka consumer. The consumer is kept simple and concurrency is achieved by other methods.**

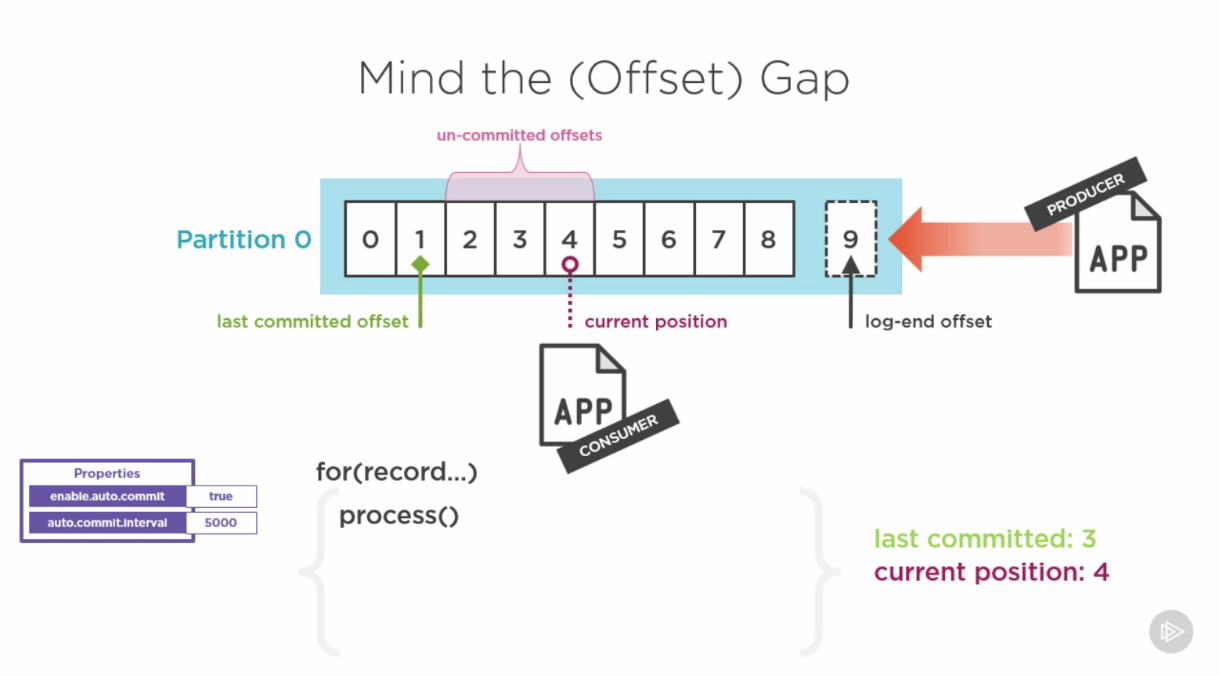
**What happens after the poll() has returned the data for processing?**

The return type of a poll() is a collection of consumer records. Ever collection is iterated over and developer decides what logic is applied to every records. Since poll() is single threaded, the records that take time to process might affect the performance of the cluster. However, kafka’s architecture prevents the Producer, Broker, and other consumers from such ‘slow consumers’. Since a consumer can poll data from multiple topics/cluster and the poll() is single threaded, it’s only rational to have multiple consumers.

**What are offsets and why are they used?**

Offsets determine/indicate the position from where last message was read in a partition. Every partition maintains anatomy of offsets across various partitions.

Categories of offsets:

Last committed offset: This offset is to indicate if an individual has or has not read a message from a partition. It represents the last record that the consumer has confirmed to have processed. The current position indicated the position in the log which will undergo processing now. The messages between the last-committed-offset and the current position are called the un-committed offset. The last offset of the log is called the log-end offset. Kafka consumer is responsible to maintain how un-committed offsets are maintained, but it is driven by the property called “enable.auto.commit” which has default value of true. When kafka is committing automatically, kafka has no way to know as to how records should be logically marked “committed” hence, in this scenario, kafka needs to establish an interval of time between the commit actions that faithfully commit based on a frequency. This frequency is set by another property called “auto.commit.interval” and is set to 5000ms by default. This is a bad number for high throughput systems. But eventually it depends on the processing logic, inside of the application.  


If a message takes more than the specified number of ms for processing, kafka will go ahead and commit the record and mark as committed, regardless if it’s done or not. (This decreases system consistency, but is okay most of the times, unless system should be reliable).

Account for inconsistency:

If an error occurs, kafka has no way to know if the messages that were marked committed were actually processed or now. This affects based on the consumer topology

**Offset behavior:**

**Read != Committed**

**Offset behavior is configurable:**

* **Enable.auto.commit = true (default)**
* **Auto.commit.interval.ms = 5000 (default)**
* **Auto.offset.reset = “latest” default ->** Strategy used when a consumer starts reading from a new partitions. The default is to read from the latest known committed offset. Other options are “earliest”. There’s an option to set it “none”, which will result in an exception and let the user to decide what to do.
* **Depends on Single vs Consumer Groups**

**How are offsets stored in Kafka?**

Kafka stored the offsets in a separate topic called “\_\_consumer\_offsets” which has 50 partitions. This partition is visible with describe command. The ConsumerCoordinator class is responsible for producing the offset messages in the “\_\_consumer\_offsets” topic. This means a consumer is sort of a producer.

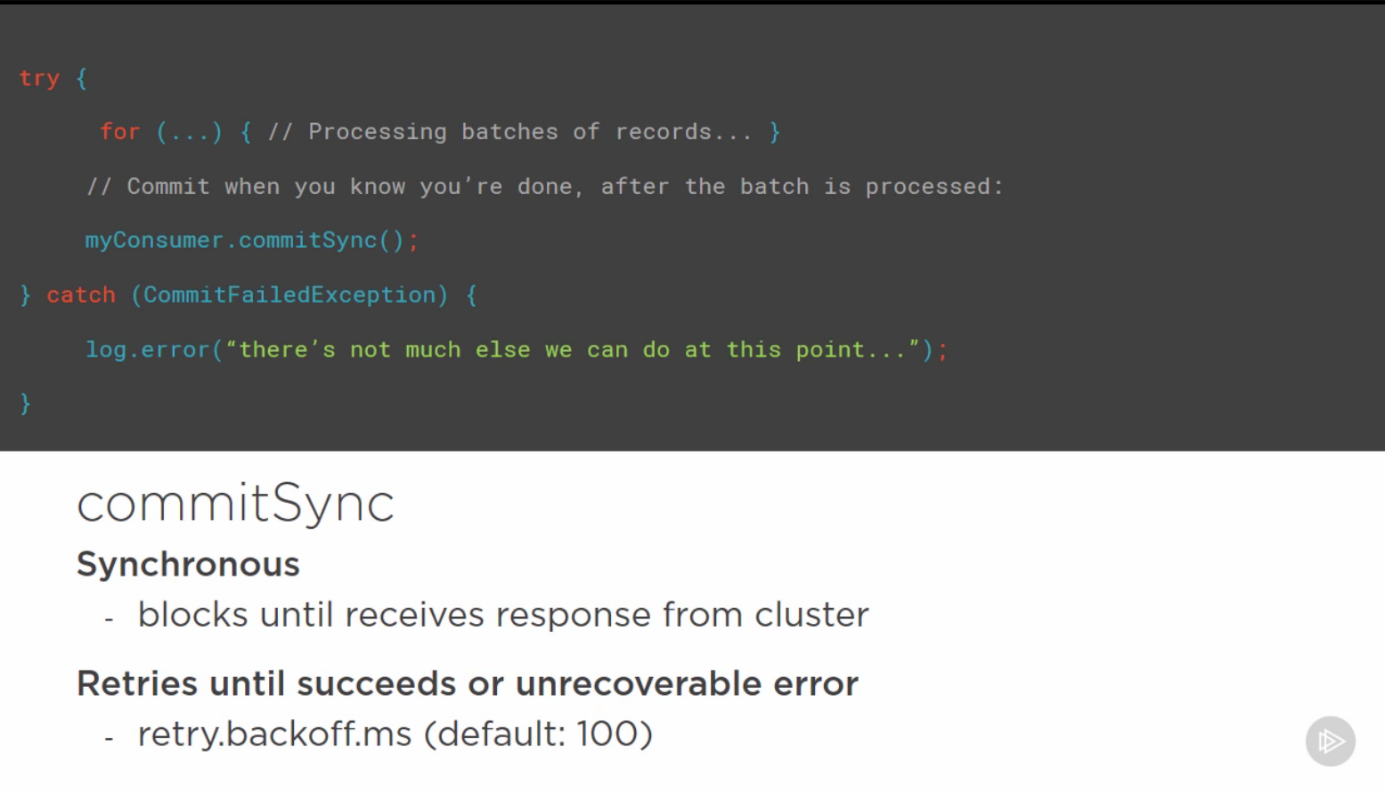
**Offset management:**

* **Automatic vs Manual**
  + To switch to manual, set enable.auto.commit = false (after doing this, the prop for setting the interval is invalid now)
  + When manual mode is selected, you are taking full responsibility of offset commits.

**How do commits happen for message offsets?**

Two methods enable this: commitSync and commitAsync

commitSync:

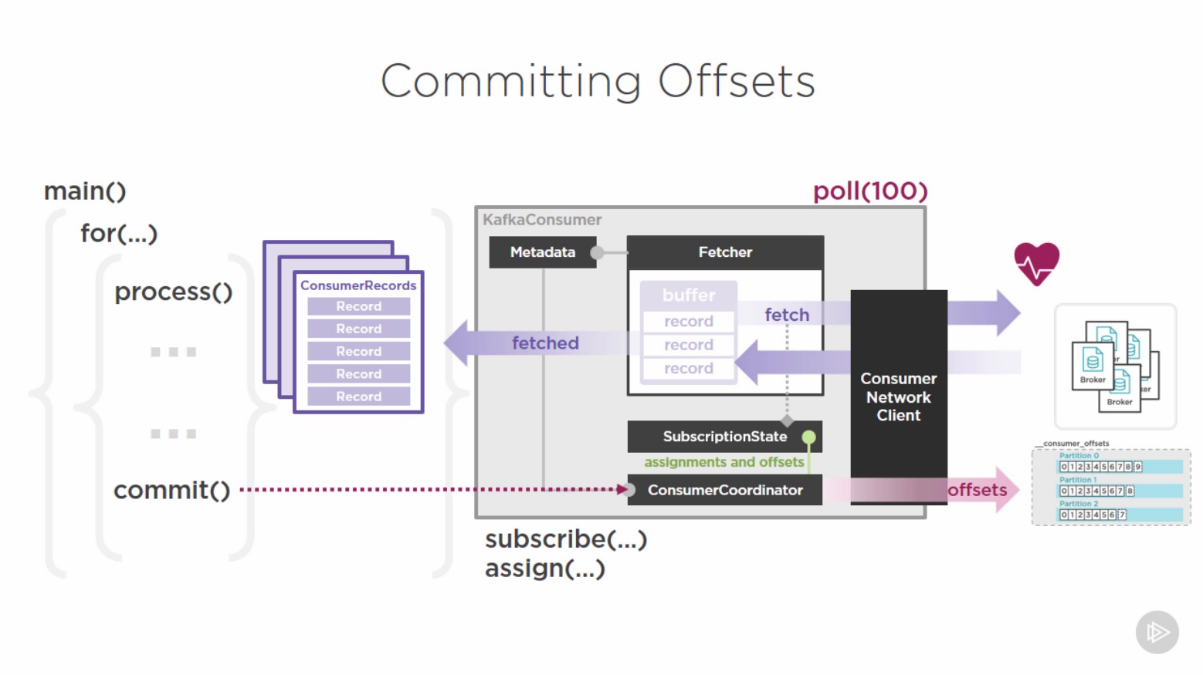


commitAsync:



|  |  |
| --- | --- |
| Commit Sync | Commit Async |
| Sync, hence blocks processing till it receives | Non-blocking but non-deterministic |
| Frequent retries happen based on “retry.backoff.ms” prop. | No retires as it cannot be determined whether the previous commit happened or not. Also, this might result in duplication and wrong ordering of messages. Callback can be used to determine the response. |
| Trading throughput and performance for control over the consistency | Throughput and Performance are significantly high, but required a great callback management mechanism, otherwise this can lead to worse situations. |

**What happens after commit has happened?**

The offset management occurs after the poll() has been timed out and presented records for processing, whether auto-commit and manual commit, it will take the processed batch, determine their offsets, and ask the ConsumerCoordinator to commit them to the Kafka cluster via the consumer network client, which it does immediately. After this, the ConsumerCoordinator updates the subscription state object, so that the fetcher knows what offsets have been committed and what records it should fetch.

**Why to take manual control of offsets?**

**Consistency control**

* When is “done” (auto-commit is not reliable)

**Atomicity**

* Exactly once vs At-least-once

To gain entire transactional control of the scope, it is required to manage offsets, content of the message and/or the result of its processing in the same store.

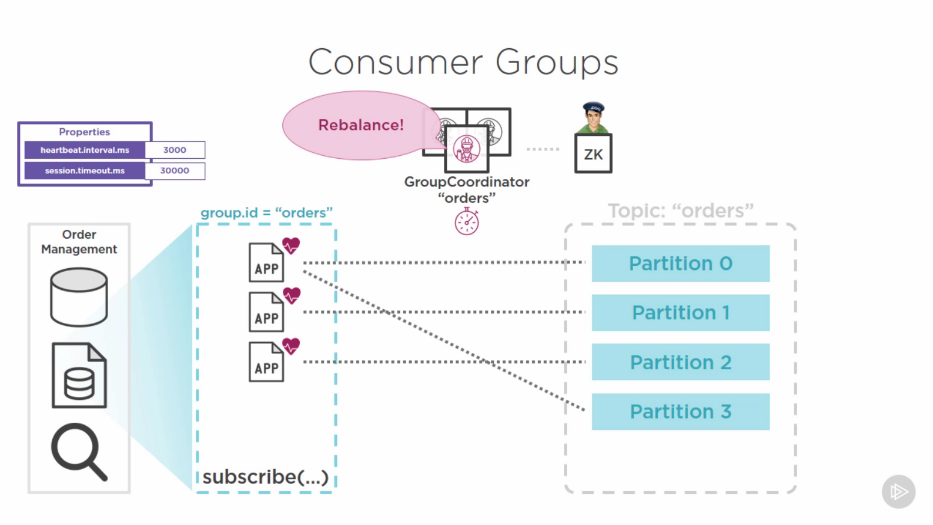
## Consumer groups:

## 

Individual consumers having same “group.id” and topic list form a consumer group. A designated broker is elected to serve as the coordinator. This coordinator works with Cluster Coordinator and Zookeeper to assign the monitor specific topics to specific members of a consumer group.

When a group forms, every member sends regular heartbeat to the coordinator based on the “heartbeat.interval.ms” property (default 3000ms). Other property “session.timeout.ms” a property which is used to determine if the consumer is live or not. If a response is not returned in the interval specified in this setting, the consumer is declared dead and corrective actions are taken. If a consumer is not available or failed, its partitions are assigned to some other consumer and this procedure is called **consumer rebalance**.

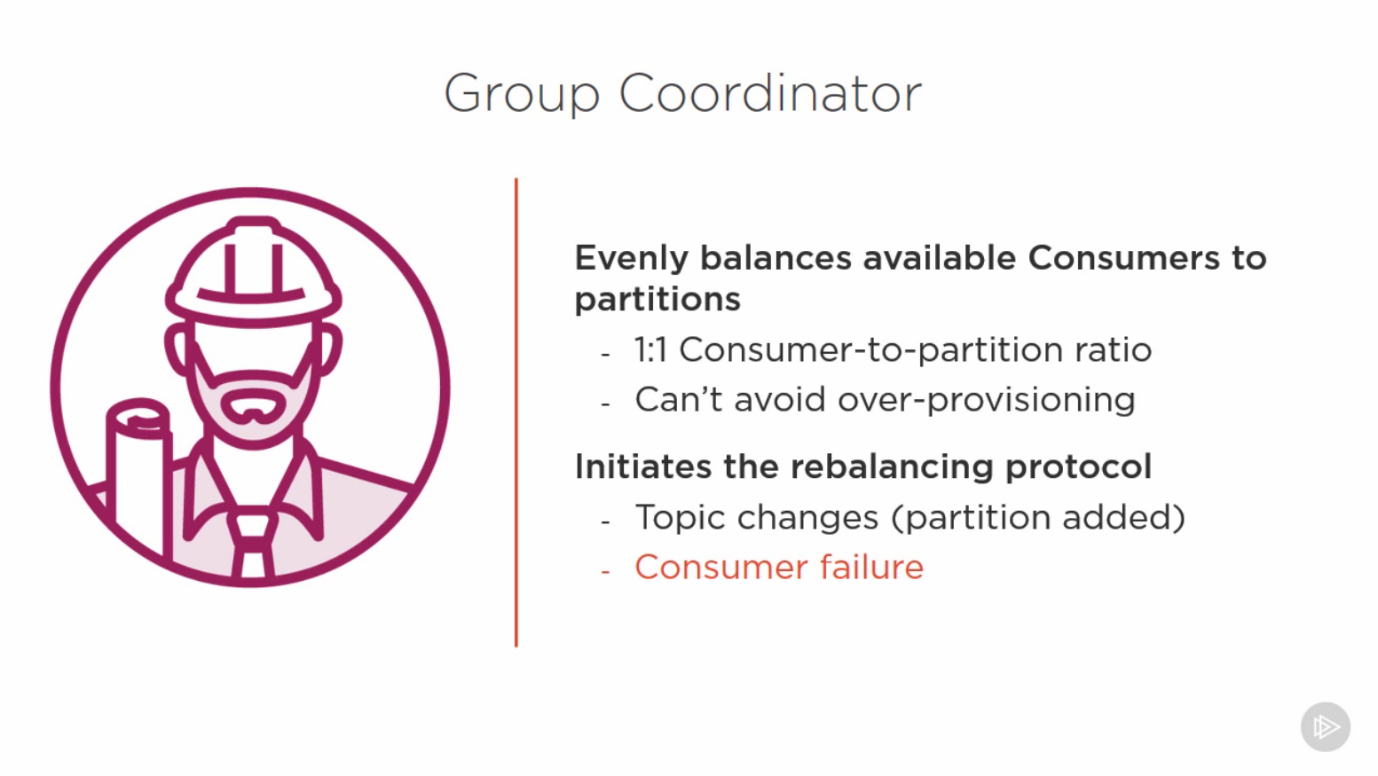
When Consumer Rebalancing happens, the failed consumer’s work is assigned to new consumer and it has to take twice the load it was taking earlier. The rebalancing process includes checking where the previous consumer left-off. Hopefully, avoiding processing duplicate records. Offset management plays a major role here. If offsets are not managed properly, the ability for the consumer group to failover and rebalance itself can be compromised.



**What happens when a consumer group rebalances?**

When a new consumer is assigned a previous partition, it needs to know which offset should it start from? The ConsumerState object usually has last committed position cached, the new consumer can start from the next offset. This behavior is driven by “auto.offset.resetting” property (default latest). Since the default is latest, the new consumer will start reading from the latest committed offset in the partition. Duplicates can be there if a consumer was processing records while rebalancing happened.

**Responsibilities of a group coordinator:**

****

**Some Important performance settings:**

* fetch.min.bytes – Minimum number of bytes returned from the poll. This settings helps in saving poll cycles. (Similar to batch.size settings on producer)
* max.fetch.wait.ms – Max ms to wait in a poll if fetch.min.bytes is not met. (Similar to linger.ms on producer)
* max.partition.fetch.bytes – Max number of bytes a poll can retrieve per cycle (Processing more records might lead to session timeout)
* max.poll.records – Maximum number of records in a poll. (Processing more records might lead to session timeout)

**Advanced topics not covered in this section:**

* **Consumer position control:** Which position should the consumer consume from in a partition?
  + seek() – expects an offset arg to read from
  + seekToBeginning()
  + seekToEnd()
* **Flow control:** These settings let the consumer pause or resume consuming (For example on priority basis). This is specifically useful when a single consumer has to read from multiple topics and partitions.
  + pause()
  + resume()
* **Rebalance listeners:** Responsible for notifying a rebalance happened

# What’s there for kafka in future? Kafka is evolving.

Strengths of Kafka:

* Primary choice for connecting disparate source of data. (Many connections in many languages)
* Large-scale data movement pipeline
* “Big Data” integration

Challenges remaining:

* Governance and Data Evolution:
  + In advanced systems, it is infeasible to restrict the type-contracts solely based on the built-in serializers. Diverse data requires more serializers. This also needs more de-serializers.
  + Kafka lacks a common means of cataloging, registering, and reconciling the disparate message specification and compatibility mappings between the serializing producer and deserializing consumers.
  + Confluent has made a kafka schema registry to overcome this shortcoming.
    - Apache Avro serialization format. This is a self-described industry format that has broad industry adoption.
    - Serialize to Avro and deserialize it when required.
    - Schema can be versioned and maintained inside of the Schema registry, centrally, within the Kafka cluster (Enables easy restful service-based discovery and version compatibility conciliation)
    - Open source with Apache License
* Consistency and Productivity
  + Lot of duplication of effort in terms of writing producer and consumer applications that connect the sources and targets together.
  + Kafka lacks a consistent way to provide a common framework for integrating data sources and targets. It is left for the engineers to write their own producer and consumer API on top of the generic ones. This increase a development and maintenance cost.
  + From kafka 0.10 version, a new framework is integrated into kafka to solve this issue. Kafka Connect and Connector Hub.
    - This is an API for Developers making this job easier and more consistent
    - Standardization of common approaches.
    - Helps in easily writing Producer and Consumer application.
    - Core connectors can be built, currently there are 50+ designed to connect to various products and services
    - Connector Hub hosts are the connectors, is maintained by Confluenct.
* Big and Fast Data
  + Kafka acts as a central piece for big data systems. All other components introduce their own model for development and operation. All have their own API and cluster-based management approach to distributed systems (this includes Kafka)
  + All these techs under one roof is trying to achieve one thing: Analyze more data in real-time. This introduces consistency and productivity challenges in integrating these all together
  + To achieve high throughput and connected to these distributed systems, a new client for stream-based processing was added to Kafka, called the Kafka Steams.
    - Businesses, who have Kafka working on their system, now do not have to maintain separate platforms for streaming platforms, etc., for working with Kafka
    - Kafka CAN, standalone, be a solution when stream-based processing is required.
    - This enables businesses to lower the cost (As they can essentially work with Hadoop + Spark + Kafka for their Big-Data Cluster)
    - Kafka stream is a client library just like KafkaProducer and KafkaConsumer. These can also be embedded in java-based applications (hence lowers the barrier for adoption)

**Finally, who is using, building, and living kafka?**

* Linkedin
* Netflix
* Uber
* Twitter
* Confluent

# Commands:

* To start the kafka server:

**bin/kafka-server-start.sh config/server.properties**

* To start multiple kafka servers:

**bin/kafka-server-start.sh config/server.properties**

**bin/kafka-server-start.sh config/server-1.properties**

**bin/kafka-server-start.sh config/server-2.properties**

**NOTE: broker.id, PORT, and LOG.DIR should be different**

* To create a new topic:

**bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partition 1 --topic my-topic**

* To alter a topic (here we’re modifying the number of partitions):

**bin/kafka-topics.sh --bootstrap-server localhost:9092 --alter --topic my-topic --partitions 4**

* To get topic info like Leader, Replicas, ISR, Partition

**bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-topic**

* To start a producer:

**bin/kafka-console-producer.sh --broker-list localhost:9092,localhost:9093 --topic my-topic**

* To start a consumer:

**bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic --from-beginning**

* To run performance tests on producer:

**bin/kafka-producer-perf.test.sh <determine arg list from the script>**