

Kafka Producers

Writing data to Kafka

- You use Kafka “producers” to write data to Kafka brokers.
 - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
 - The Kafka project only provides the JVM implementation.
 - Has risk that a new Kafka release will break non-JVM clients.
- A simple example producer:

```
1 Properties props = new Properties();
2 props.put("metadata.broker.list", "...");
3 ProducerConfig config = new ProducerConfig(props);
4
5 Producer p = new Producer(ProducerConfig config);
6 KeyedMessage<K, V> msg = ...; // cf. later slides
7 p.send(KeyedMessage<K,V> message);
```

- Full details at:
 - <https://cwiki.apache.org/confluence/display/KAFKA/0.8.0+Producer+Example>

Kafka Producers

- **Producers** send records to topics
- **Producer** picks which partition to send record to per topic
 - Can be done *round-robin*
 - Can be based on priority
- A simple example producer:

- The Kafka project only provides the JVM implementation.
 - Has risk that a new Kafka release will break non-JVM clients.

- Full details at:
-

Producers

- The Java producer API is very simple.
 - We'll talk about the slightly confusing details next. 😊

```
1  class kafka.javaapi.producer.Producer<K,V>
2  {
3      public Producer(ProducerConfig config);
4
5      /**
6       * Sends the data to a single topic, partitioned by key, using either the
7       * synchronous or the asynchronous producer.
8       */
9      public void send(KeyedMessage<K,V> message);
10
11     /**
12      * Use this API to send data to multiple topics.
13      */
14     public void send(List<KeyedMessage<K,V>> messages);
15
16     /**
17      * Close API to close the producer pool connections to all Kafka brokers.
18      */
19     public void close();
20 }
```

Producers

- Two types of producers: “async” and “sync”

```
1 Properties props = new Properties();  
2 props.put("producer.type", "async");  
3 ProducerConfig config = new ProducerConfig(props);
```

- Same API and configuration, but slightly different semantics.
 - What applies to a sync producer almost always applies to async, too.
 - Async producer is preferred when you want higher throughput.
- Important configuration settings for either producer type:

| | |
|------------------------------------|--|
| <code>client.id</code> | identifies producer app, e.g. in system logs |
| <code>producer.type</code> | async or sync |
| <code>request.required.acks</code> | acking semantics, cf. next slides |
| <code>serializer.class</code> | configure encoder, cf. slides on Avro usage |
| <code>metadata.broker.list</code> | cf. slides on bootstrapping list of brokers |

Sync producers

- Straight-forward so I won't cover sync producers here
 - Please go to <https://kafka.apache.org/documentation.html>
- Most important thing to remember: `producer.send()` will block!

Async producer

- Sends messages in background = no blocking in client.
- Provides more powerful batching of messages (see later).
- Wraps a sync producer, or rather a pool of them.
 - Communication from async->sync producer happens via a queue.
 - Which explains why you may see `kafka.producer.async.QueueFullException`
 - Each sync producer gets a copy of the original async producer config, including the `request.required.acks` setting (see later).
 - Implementation details: [Producer](#), [async.AsyncProducer](#), [async.ProducerSendThread](#), [ProducerPool](#), [async.DefaultEventHandler#send\(\)](#)

Producers

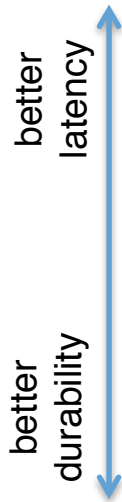
- Two aspects worth mentioning because they significantly influence Kafka performance:
 1. Message acking
 2. Batching of messages

1) Message acking

- Background:
 - In Kafka, a message is considered *committed* when “any required” ISR (in-sync replicas) for that partition have applied it to their data log.
 - Message acking is about conveying this “Yes, committed!” information back from the brokers to the producer client.
 - Exact meaning of “any required” is defined by `request.required.acks`.
- Only **producers** must configure acking
 - Exact behavior is configured via `request.required.acks`, which determines when a produce request is considered completed.
 - Allows you to trade **latency (speed) <-> durability (data safety)**.
 - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don't need to worry about potentially seeing a message that could be lost if the leader fails.

1) Message acking

- Typical values of `request.required.acks`

- 
- **0**: producer never waits for an ack from the broker.
 - Gives **the lowest latency** but the weakest durability guarantees.
 - **1**: producer gets an ack after the leader replica has received the data.
 - Gives better durability as we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.
 - **-1**: producer gets an ack after *all* ISR have received the data.
 - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one ISR remains.

- Beware of interplay with `request.timeout.ms`!

- "The amount of time the broker will wait trying to meet the `request.required.acks` requirement before sending back an error to the client."
- Caveat: Message may be committed even when broker sends timeout error to client (e.g. because not all ISR ack'ed in time). One reason for this is that the producer acknowledgement is independent of the leader-follower replication, and ISR's send their acks to the leader, the latter of which will reply to the client.

2) Batching of messages

- Batching improves throughput
 - Tradeoff is data loss if client dies before pending messages have been sent.
- You have two options to “batch” messages in 0.8:
 1. Use `send(listOfMessages)`.

```
1 producer.send(List<KeyedMessage<K,V>> messages);
```

- Sync producer: will send this list (“batch”) of messages *right now*. Blocks!
- Async producer: will send this list of messages in background “as usual”, i.e. according to batch-related configuration settings. Does not block!

2. Use `send(singleMessage)` with async producer.

```
1 producer.send(KeyedMessage<K,V> message);
```

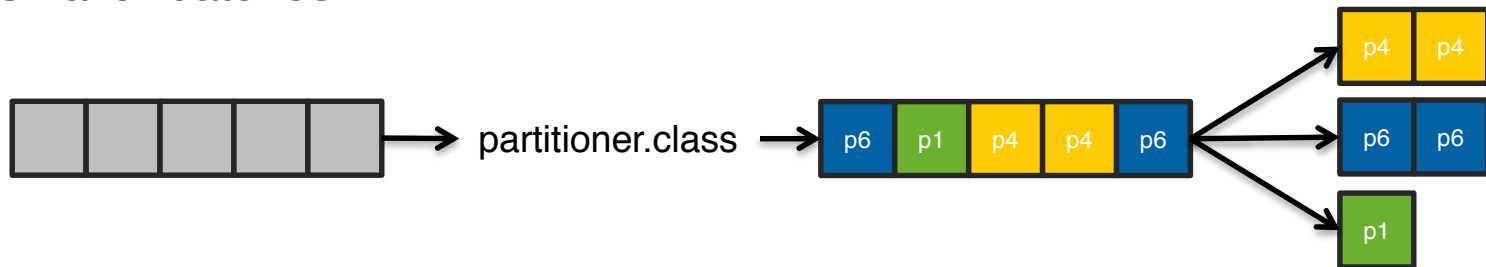
- For async the behavior is the same as `send(listOfMessages)`.

2) Batching of messages

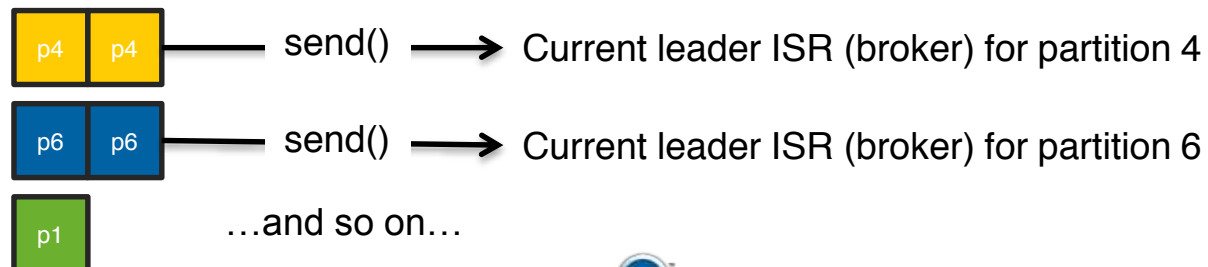
- Option 1: How `send(listOfMessages)` works behind the scenes

```
1 producer.send(List<KeyedMessage<K,V>> messages);
```

- The original list of messages is partitioned (randomly if the default partitioner is used) based on their destination partitions/topics, i.e. split into smaller batches.



- Each post-split batch is sent to the respective leader broker/ISR (the individual `send()`'s happen sequentially), and each is acked by its respective leader broker according to `request.required.acks`.

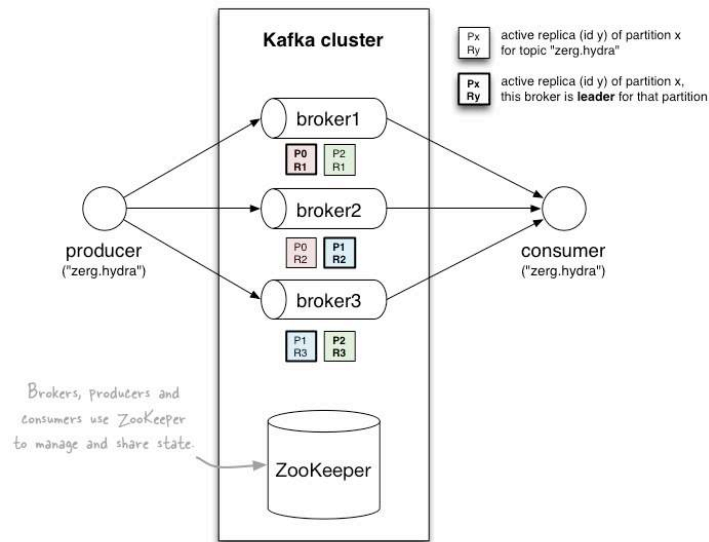


2) Batching of messages

- Option 2: Async producer
 - Standard behavior is to batch messages
 - Semantics are controlled via producer configuration settings
 - `batch.num.messages`
 - `queue.buffering.max.ms + queue.buffering.max.messages`
 - `queue.enqueue.timeout.ms`
 - And more, see [producer configuration docs](#).
- Remember: Async producer simply wraps sync producer!
 - But the batch-related config settings above have no effect on “true” sync producers, i.e. when used without a wrapping async producer.

Write operations behind the scenes

- When writing to a topic in Kafka, producers write directly to the partition leaders (brokers) of that topic
 - Remember: Writes always go to the leader ISR of a partition!



- This raises two questions:
 - How to know the “right” partition for a given topic?
 - How to know the current leader broker/replica of a partition?

1) How to know the “right” partition when sending?

- In Kafka, a producer – i.e. the **client** – decides to which target partition a message will be sent.
 - Can be random ~ load balancing across receiving brokers.
 - Can be semantic based on message “key”, e.g. by user ID or domain name.
 - Here, Kafka guarantees that all data for the same key will go to the same partition, so consumers can make locality assumptions.

```
1 // Java example. Topic is "zerg.hydra".
2 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
3 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- But there’s one catch with line 2 (i.e. no key) in Kafka 0.8.

Keyed vs. non-keyed messages in Kafka 0.8

- If a key **is not** specified:

```
2 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myValue");
```

- Producer will *ignore* any configured partitioner.
- It will pick a random partition from the list of *available* partitions **and stick to it** for some time before switching to another one = NOT round robin or similar!
 - Why? To reduce number of open sockets in large Kafka deployments ([KAFKA-1017](#)).
 - Default: 10mins, cf. `topic.metadata.refresh.interval.ms`
 - See implementation in `DefaultEventHandler#getPartition()`
- If there are fewer producers than partitions at a given point of time, some partitions may not receive any data. How to fix if needed?
 - Try to reduce the metadata refresh interval `topic.metadata.refresh.interval.ms`
 - Specify a message key and a customized random partitioner.
- In practice [it is not trivial to implement a correct “random” partitioner](#) in Kafka 0.8.
 - Partitioner interface in Kafka 0.8 lacks sufficient information to let a partitioner select a random and *available* partition. Same issue with `DefaultPartitioner`.

Keyed vs. non-keyed messages in Kafka 0.8

- If a key **is** specified:

```
3 KeyedMessage<String, String> msg = new KeyedMessage<>("zerg.hydra", "myKey", "myValue");
```

- Key is retained as part of the msg, will be stored in the broker.
- One can design a partition function to route the msg based on key.
- The *default partitioner* assigns messages to a partition based on their key hashes, via `key.hashCode % numPartitions`.
- Caveat:
 - If you specify a key for a message but do not explicitly wire in a custom partitioner via `partitioner.class`, your producer will use the default partitioner.
 - So without a custom partitioner, messages with the same key will still end up in the same partition! (cf. default partitioner's behavior above)

2) How to know the current leader of a partition?

- Producers: broker discovery aka bootstrapping
 - Producers don't talk to ZooKeeper, so it's not through ZK.
 - Broker discovery is achieved by providing producers with a “bootstrapping” broker list, cf. `metadata.broker.list`
 - These brokers inform the producer about all alive brokers and where to find current partition leaders. The bootstrap brokers do use ZK for that.
- Impacts on failure handling
 - In Kafka 0.8 the bootstrap list is static/immutable during producer run-time. This has limitations and problems as shown in next slide.
 - The current bootstrap approach will improve in Kafka 0.9. This change will make the life of Ops easier.