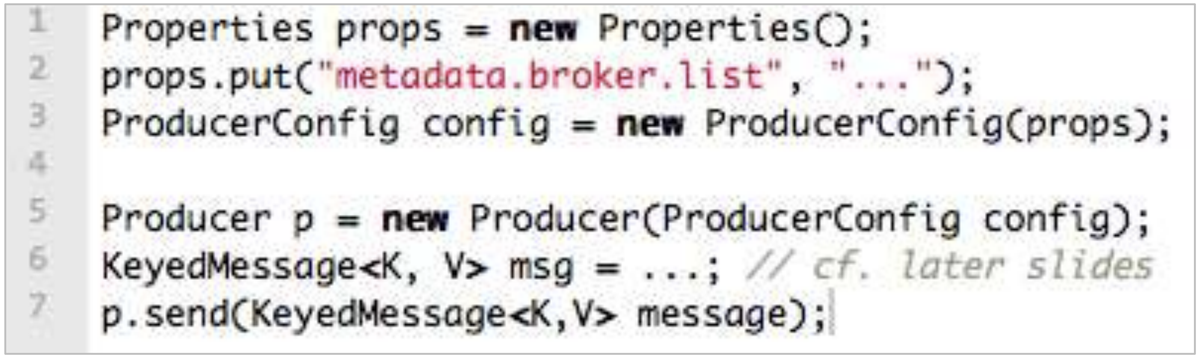
Kafka Producers

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:



```
Properties props = new Properties();
props.put("metadata.broker.list", "...");
ProducerConfig config = new ProducerConfig(props);

Producer p = new Producer(ProducerConfig config);
KeyedMessage<K, V> msg = ...; // cf. later slides
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
  - Typically based on ***key*** of ***record***
  - Kafka **default partitioner** for Java uses hash of keys to choose partitions, or a round-robin strategy if no key

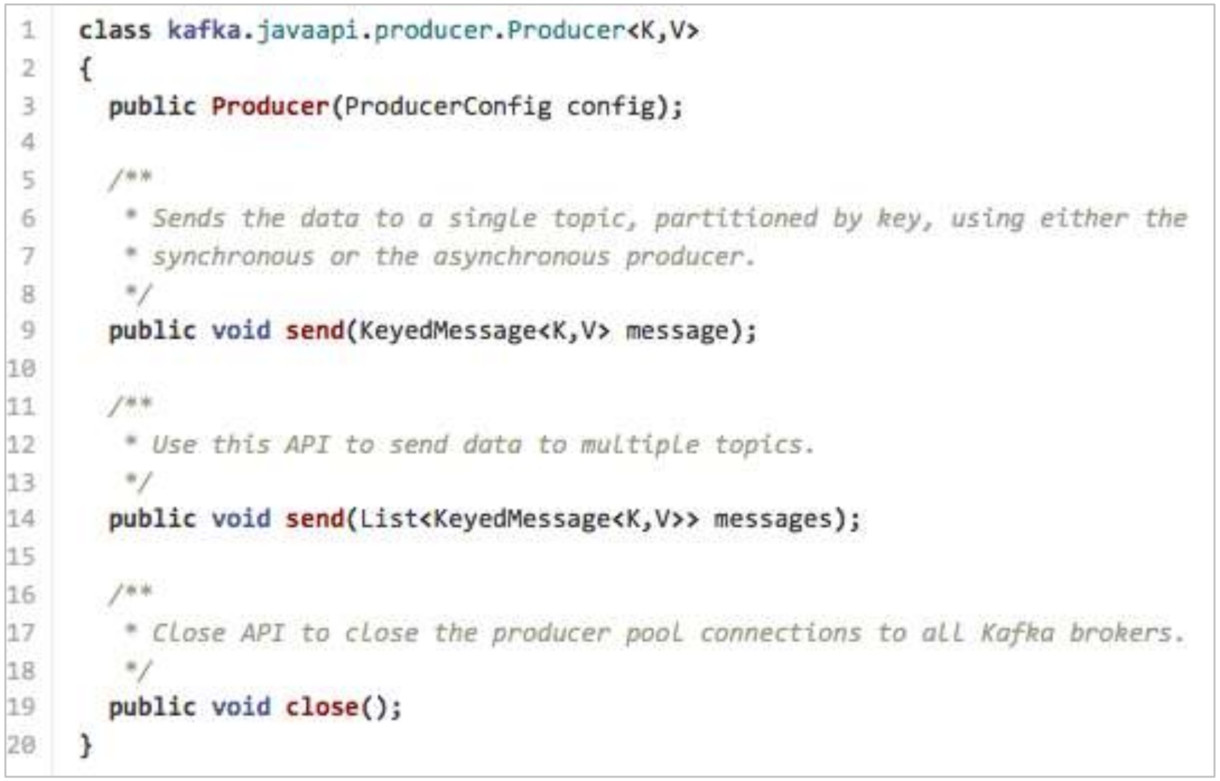
Remember! Producer picks partition.

Kafka Producers

- **Producers** write at their own cadence so order of records cannot be guaranteed across partitions.
- **Producer** configures consistency level (ack=0, ack=all, ack=1)
- **Producers** pick the **partition** such that records/messages go to a given same partition based on the data (usually key).
  - Example: have all the events of a certain employeeId go to the same partition.
  - If order within a partition is not needed, a round-robin partition strategy can be used so records are evenly distributed across partitions.

Producers

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



```java
class kafka.javaapi.producer.Producer<K,V>
{
  public Producer(ProducerConfig config);

  /**
   * Sends the data to a single topic, partitioned by key, using either the
   * synchronous or the asynchronous producer.
   */
  public void send(KeyedMessage<K,V> message);

  /**
   * Use this API to send data to multiple topics.
   */
  public void send(List<KeyedMessage<K,V>> messages);

  /**
   * Close API to close the producer pool connections to all Kafka brokers.
   */
  public void close();
}
```

Constructing a Producer

```
private Properties kafkaProps = new Properties(); [1]
kafkaProps.put("bootstrap.servers","broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");

kafkaProps.put("value.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);
```

[1] We instantiate a Properties object.

Constructing a Producer

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers","broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");❷

kafkaProps.put("value.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);
```

❷ Since we plan on using strings for message key and value, we use the built-in StringSerializer.

Constructing a Producer

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers","broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");

kafkaProps.put("value.serializer",
 "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);3
```

3 Here we create a new producer by setting the appropriate key and value types and passing the Properties object.

Ways to send messages

Fire-and-forget

We send a message to the server and don't really care if it arrives succesfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, some messages will get lost using this method.

Synchronous send

We send a message, the send() method returns a Future object, and we use get() to wait on the future and see if the send() was successful or not.

Asynchronous send

We call the send() method with a callback function, which gets triggered when it receives a response from the Kafka broker.

Sending a Message to Kafka

```
ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Precision Products", "France"); ❶

try {
  producer.send(record);
 } catch (Exception e) {
  e.printStackTrace();
}
```

❶ The producer accepts ProducerRecord objects, so we start by creating one. ProducerRecord has multiple constructors, which we will discuss later. Here we use one that requires the name of the topic we are sending data to, which is always a string, and the key and value we are sending to Kafka, which in this case are also strings. The types of the key and value must match our serializer and producer objects.

Sending a Message to Kafka

```
ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
   producer.send(record); ❷
 } catch (Exception e) {
   e.printStackTrace();
}
```

❷ We use the producer object send() method to send the ProducerRecord. The send() method returns a Java Future object with RecordMetadata, but since we simply ignore the returned value, we have no way of knowing whether the message was sent successfully or not. This method of sending messages can be used when dropping a message silently is acceptable. This is not typically the case in production applications.

Sending a Message to Kafka

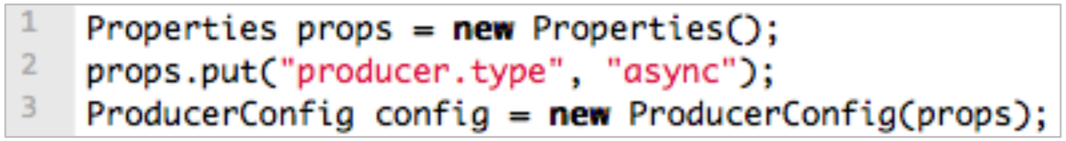
```
ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
  producer.send(record);
 } catch (Exception e) {
  e.printStackTrace(); [3]
}
```

[3] While we ignore errors that may occur while sending messages to Kafka brokers or in the brokers themselves, we may still get an exception if the producer encountered errors before sending the message to Kafka. Those can be a SerializationException when it fails to serialize the message, a BufferExhaustedException or TimeoutException if the buffer is full, or an InterruptException if the sending thread was interrupted.

Producers

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



```java
Properties props = new Properties();
props.put("producer.type", "async");
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:

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

Sync producers

- Most important thing to remember: `producer.send()` will block!

Sending a Message to Kafka **Synchronously**

```
ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
  producer.send(record).get(); ❶
 } catch (Exception e) {
  e.printStackTrace();
}
```

❶ Here, we are using Future.get() to wait for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a RecordMetadata object that we can use to retrieve the offset the message was written to.

Sending a Message to Kafka **Synchronously**

```
ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
  producer.send(record).get();
 } catch (Exception e) {
  e.printStackTrace(); ❷
}
```

❷ If there were any errors before sending data to Kafka, while sending, if the Kafka brokers returned a nonretriable exceptions or if we exhausted the available retries, we will encounter an exception. In this case, we just print any exception we ran into.

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()

Sending a Message Asynchronously

Suppose the network roundtrip time between our application and the Kafka cluster is 10ms.

If we wait for a reply after sending each message, sending 100 messages will take around 1 second. **(Synchronous)**

On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. **(Fire-and-Forget)**

On the other hand, we do need to know when we failed to send a message completely so we can throw an exception, log an error, or perhaps write the message to an “errors” file for later analysis. In order to send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. **(Asynchronous)**

Sending a Message to Kafka **Asynchronously**

```
private class DemoProducerCallback implements Callback { ❶
  @Override
  public void onCompletion(RecordMetadata recordMetadata, Exception e) {
    if (e != null) {
        e.printStackTrace();
    }
  }
}

ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");

producer.send(record, new DemoProducerCallback());
```

❶ To use callbacks, you need a class that implements the org.apache.kafka.clients.producer.Callback interface, which has a single function—onCompletion().

Sending a Message to Kafka **Asynchronously**

```
private class DemoProducerCallback implements Callback {
  @Override
  public void onCompletion(RecordMetadata recordMetadata, Exception e) {
    if (e != null) {
        e.printStackTrace();❷
    }
  }
}

ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");

producer.send(record, new DemoProducerCallback());
```

❷ If Kafka returned an error, onCompletion() will have a nonnull exception. Here we “handle” it by printing, but production code will probably have more robust error handling functions.

Sending a Message to Kafka **Asynchronously**

```
private class DemoProducerCallback implements Callback {
  @Override
  public void onCompletion(RecordMetadata recordMetadata, Exception e) {
    if (e != null) {
        e.printStackTrace();
    }
  }
}

ProducerRecord<String, String> record =
 new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");

producer.send(record, new DemoProducerCallback());❸
```

❸ And we pass a Callback object along when sending the record.

Producers

- Two aspects worth mentioning because they significantly influence Kafka performance:

  1. Message acking
  2. Batching of messages

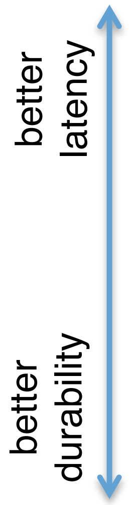
1) Message acking

- 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 the 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.
  - **`all`**: 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)`.

```
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.

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

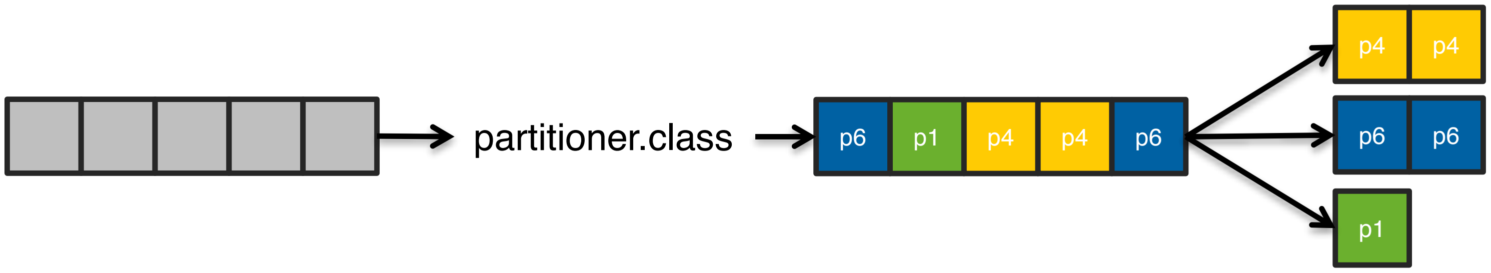
   - For async the behavior is the same as `send(listOfMessages)`.

2) Batching of messages

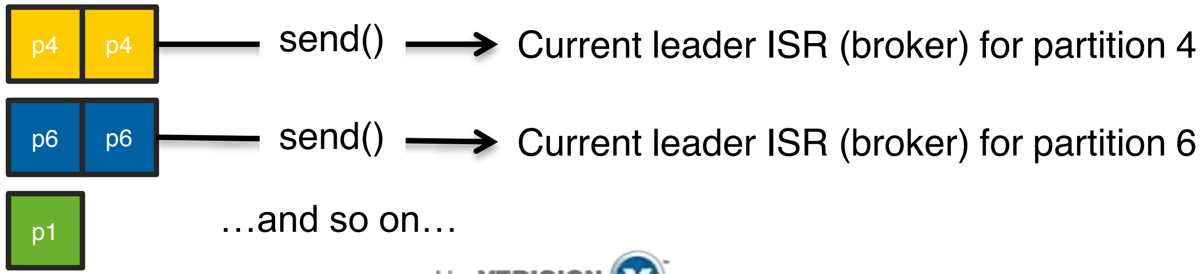
- Option 1: How `send(listOfMessages)` works behind the scenes

```
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`.


p4
p4
send()
Current leader ISR (broker) for partition 4
p6
p6
send()
Current leader ISR (broker) for partition 6
p1
…and so on…

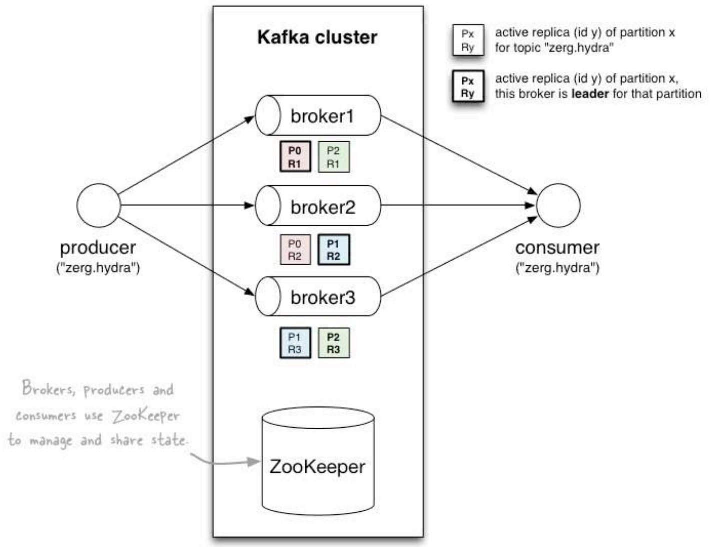
2) Batching of messages

- 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

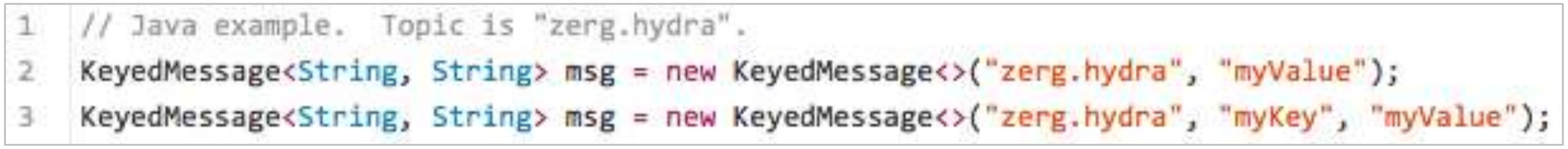
- When writing to a topic in Kafka, producers write directly to the partition leaders (brokers) of that topic
  - Remember: Writes always go to the leader ISR of a partition!


Kafka cluster
Px Ry
active replica (id y) of partition x for topic "zerg.hydra"
Px Ry
active replica (id y) of partition x, this broker is **leader** for that partition
broker1
P0 R1
P2 R1
broker2
P0 R2
P1 R2
broker3
P1 R3
P2 R3
producer ("zerg.hydra")
consumer ("zerg.hydra")
Brokers, producers and consumers use ZooKeeper to manage and share state.
ZooKeeper

- 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.



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

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.