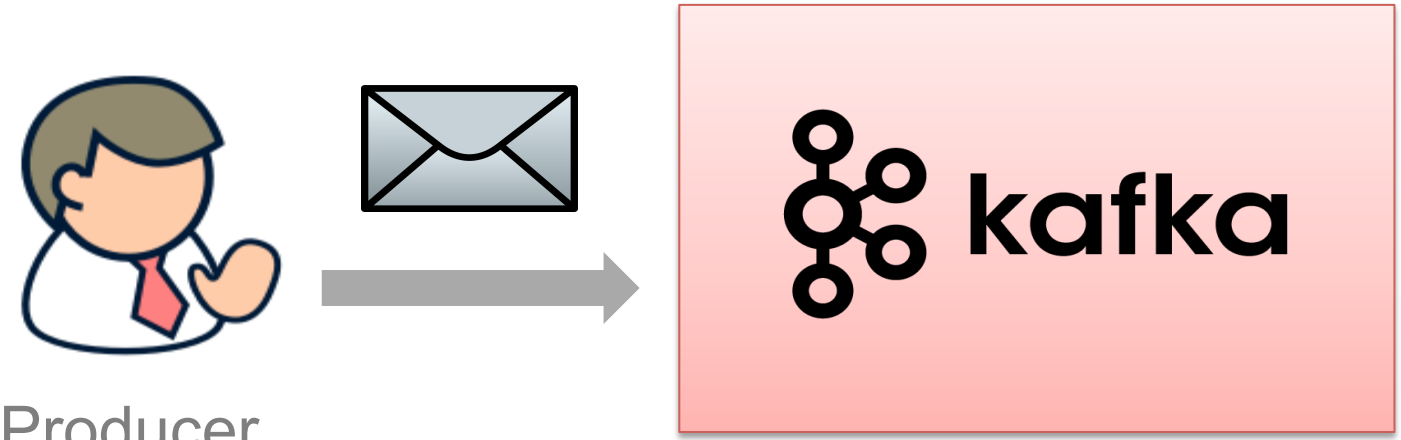
Introduction to Kafka

Outline

- Producers
  - What is a producer?
  - The producer API
- What is a Consumer?
  - What is a consumer and a consumer group?
  - The consumer API
- Anatomy of messages

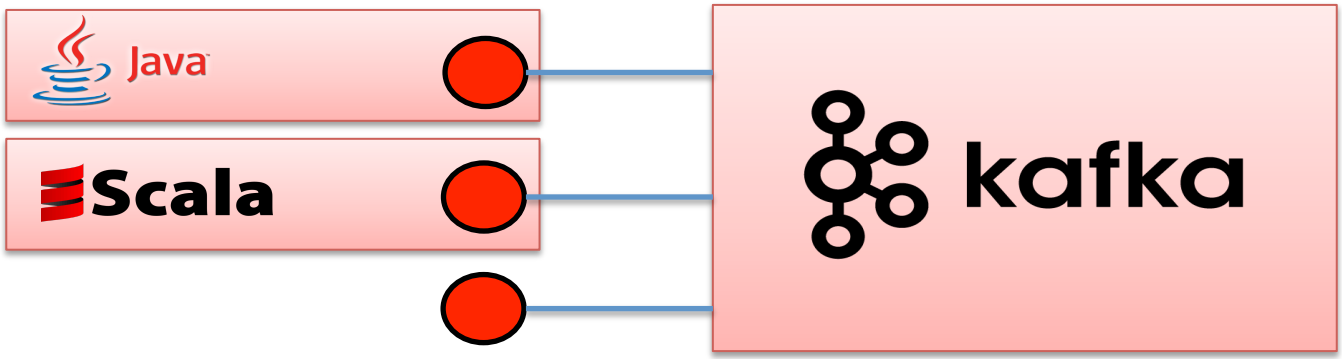
What is a Producer?



› Producer

- Producers produces the data sent to the Kafka clusters
  - Sent via topics
  - Directly involved in load-balancing
  - Controls the resiliency of messages

Kafka APIs



- Kafka ships with built in client APIs for developers to use with applications
  - Kafka ships with a Java client that is recommended
  - Legacy Scala clients are still included
  - Kafka also includes a **binary wire protocol**
  - Many tools in other languages that implement this wire protocol

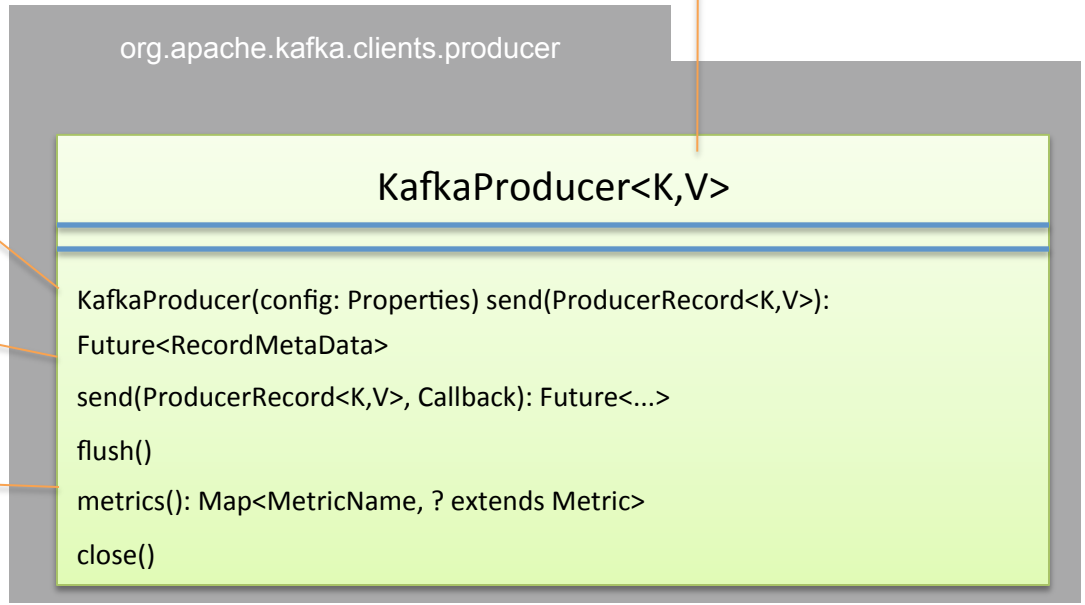
The Java API

Generic sender where:
K = Type of key
V = Type of message

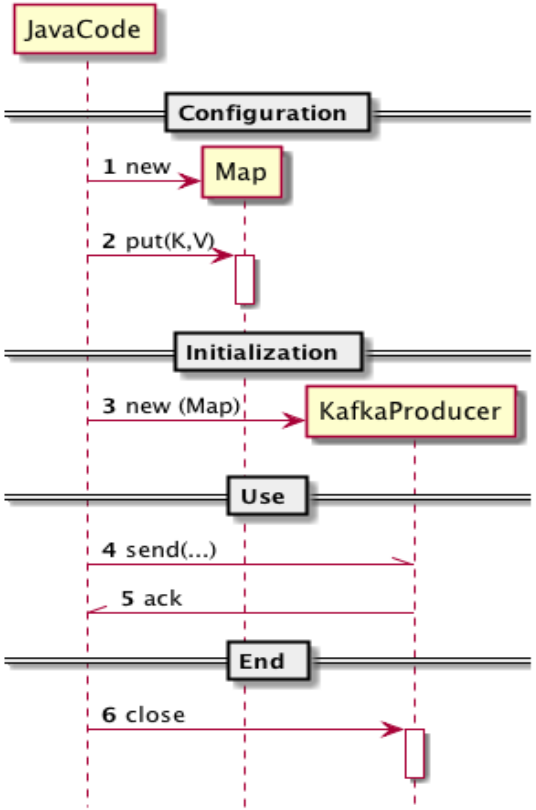
Constructor takes a configuration (mostly a hashmap of options)

Send a messages (with or without callbacks)

Get metrics for this producer



Java API Behavior




```
// Configuration
Properties kp = new Properties();
kp.put("bootstrap.servers",
    "mybroker1:9092,mybroker2:9092");
kp.put("key.serializer", "...");



// Initialization
KafkaProducer<String, String> producer =
    new KafkaProducer<String, String>(kp);

// Use
Future<...> f = producer.send(...);
... f.get(); // when acked


// End
producer.close();
```

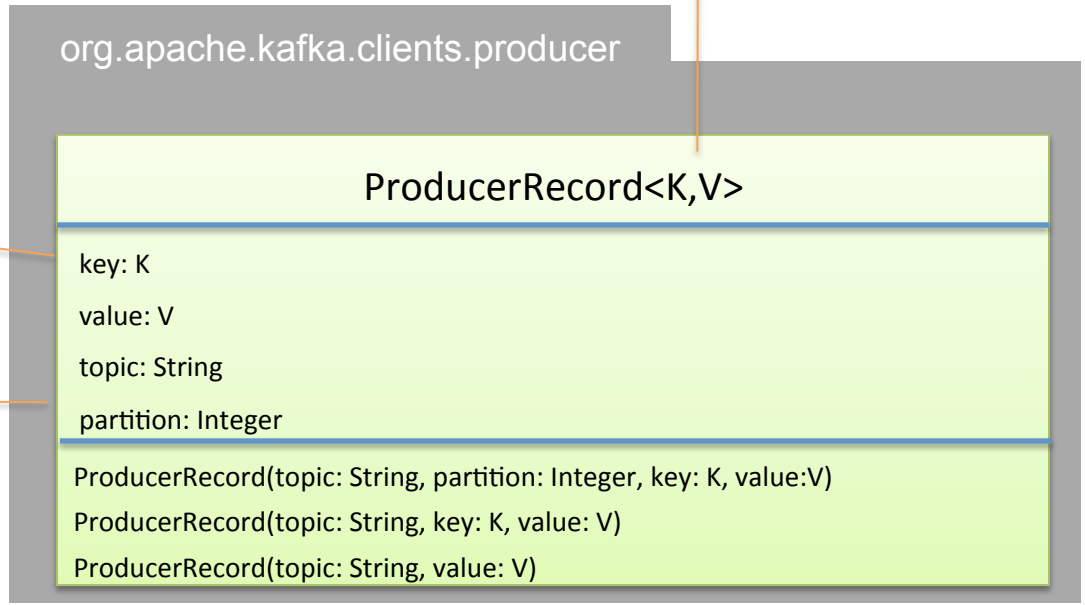
SciSpike

Creating a Kafka Producer

- Constructing a Kafka producer requires 3 mandatory properties
  - `bootstrap.servers` – list of `host:port` pairs of Kafka brokers. This doesn't have to include all brokers in the cluster as the producer will query about additional brokers. It is recommended to include at least 2 in case one broker goes down
  - `key.serializer` – should be set to a class that implements the `Serializer` interface that will be used to serialize **keys**
  - `value.serializer` - should be set to a class that implements the `Serializer` interface that will be used to serialize **values**

The ProducerRecord

Generic record where:
K = message key
V = Type of message



Messsage Key is optional

Partition Key is optional

Sending a Message

```
ProducerRecord<String,String> record =
  new ProducerRecord<String,String>(
    "someTopic", "someKey, "someValue");

producer.send(record,  new Callback() {
   public void onCompletion(
      RecordMetadata metadata, Exception e) {
         if(e != null) e.printStackTrace();
         System.out.println(
           "Offset: " + metadata.offset());
      }
});
```

Producer Controls Message Guarantees

- As a producer you can determine what guarantees you want Kafka to give you when sending a message
  - Controlled by acknowledgement
- Different use cases requires different guarantees
  - Web page clicks log → Don't care if I loose a few messages
  - Credit card payment → I want best possible guarantee
- Kafka provides options
  - The better guarantee, the lower latency

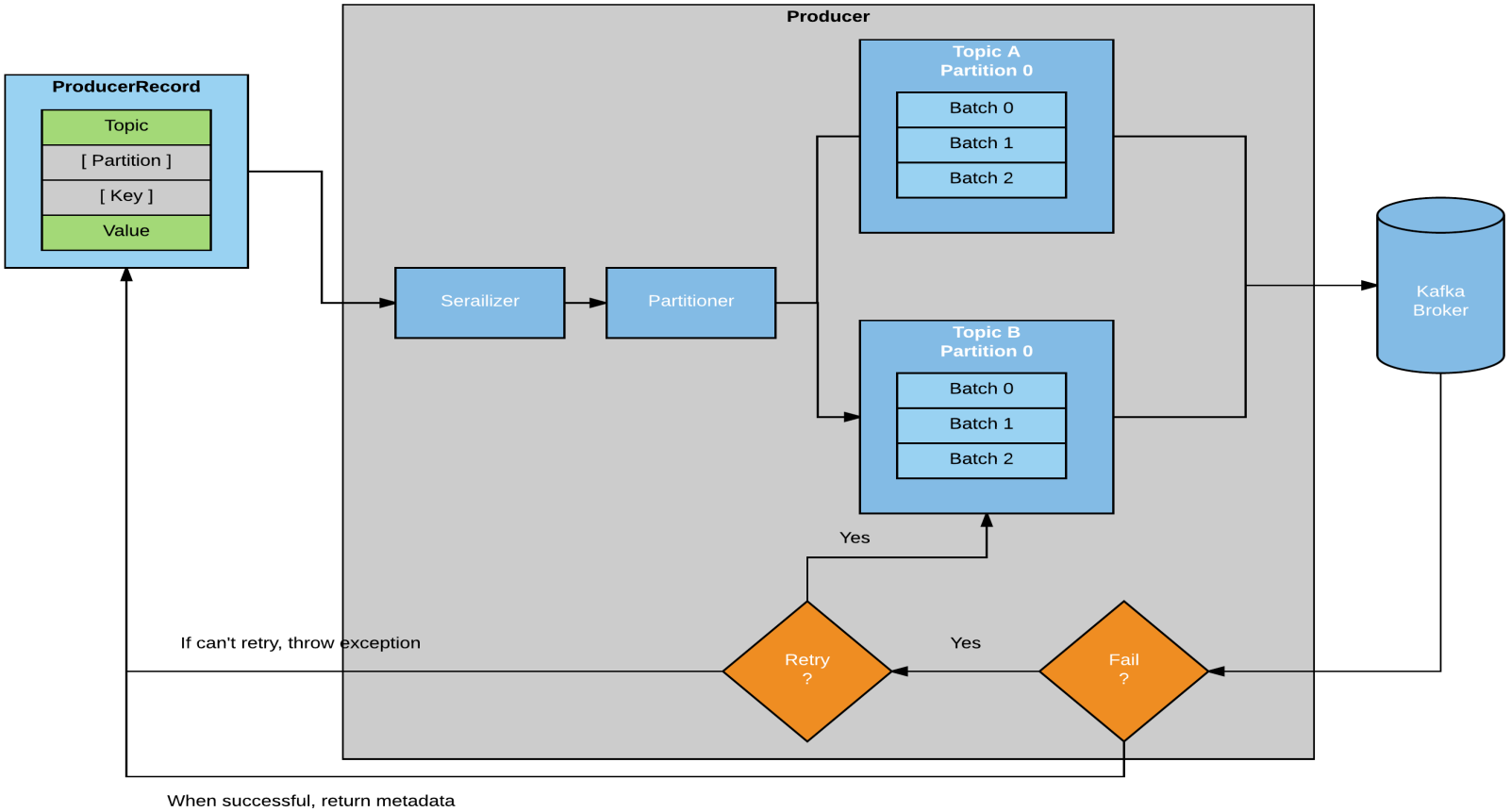
Producer API

- Different use case requirements will influence the way the producer API is used to write messages to Kafka and its configuration
- Three primary ways of sending messages
  - Fire-and-forget – send a message and don't really care if it arrived successfully or not. Most of the time it will arrive successfully but it's possible that some messages will get lost
  - Synchronous send – message is sent and a `Future` object is returned which can be used to see if the `send()` was successful
  - Asynchronous send – the `send()` method has a callback function which is triggered when a response is received from the Kafka broker

Acknowledgement of Messages

- No ack (0)
  - Kafka will most likely receive the message
  - Producer will not wait for any reply from the broker before assuming the message was sent successfully
  - If something goes wrong, producer will not know and message is lost
  - Because producer is not waiting for a response, high throughput can be achieved
- Ack from N replicas (1..N)
  - A message is not considered consumed by the Kafka cluster unless N replicas holding the message has acknowled
- Ack from all replicas (-1)
  - Every replica must acknowledge the message

Producer Overview



Serialization

- Kafka messages are byte arrays (to Kafka)
  - Key → Array of bytes
  - Value → Array of bytes
- The Java API allows you to pass any object as key or value
  - Makes the code readable, but…
  - … requires serializes and deserializes

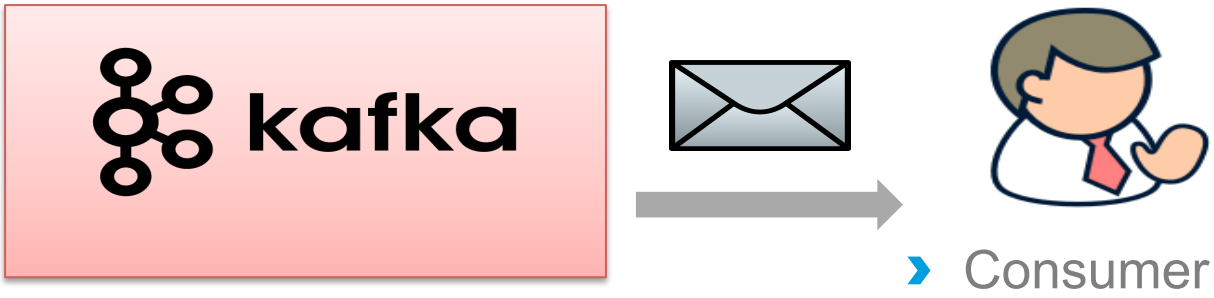
- Kafka includes an interface for this
  `org.apache.kafka.common.serialization.Serializer`

Built-in Serializers

- Kafka includes serializes for common types:
  - `ByteArraySerializer`
  - `StringSerializer`
  - `IntegerSerializer`
  - `...`

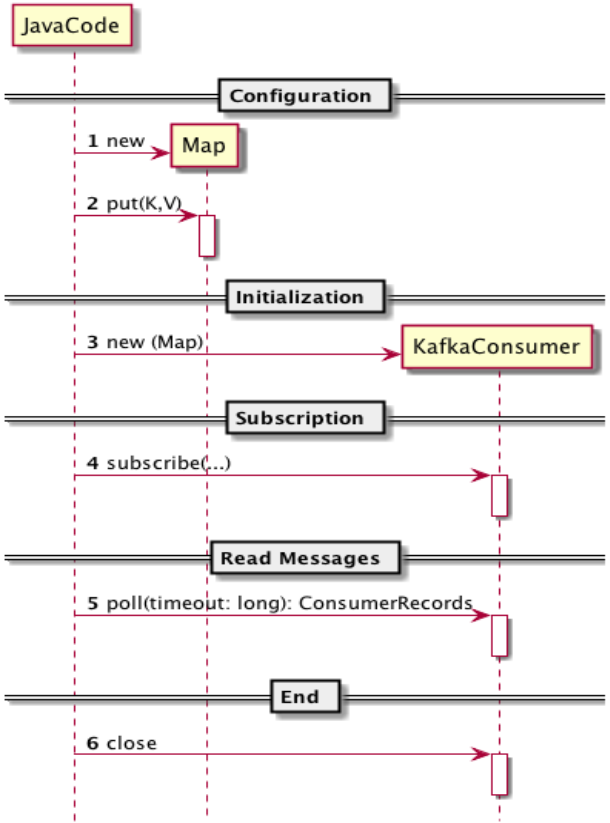
- Most organizations settle on some standard serialization strategy
  - JSON, XML, Apache Avro, Protobuf

Consumers and Consumer Groups



- Applications that read data from Kafka are consumers
  - Subscribes to topics
  - Use `KafkaConsumer` to read messages from these topics
  - Kafka consumers are usually part of a *consumer group*
  - **The main way consumption of data from a Kafka topic is scaled is by adding more consumers to a consumer group**

Java API Behavior




```
// Configuration
Properties kp = new Properties();
kp.put("bootstrap.servers",
     "mybroker1:9092,mybroker2:9092");
kp.put("key.deserializer", "...");


// Initialization
KafkaConsumer<...> consumer=
     new KafkaConsumer<...>(kp);


// Subscription
consumer.subscribe("interesting.*");


// Read messages
ConsumerRecords<...> records = producer.poll(100);
for (ConsumerRecord<...> cr : records) {
     // cr.value(); cr.key(); cr.offset();
}

// End
consumer.close()
```

SciSpike

The Java API

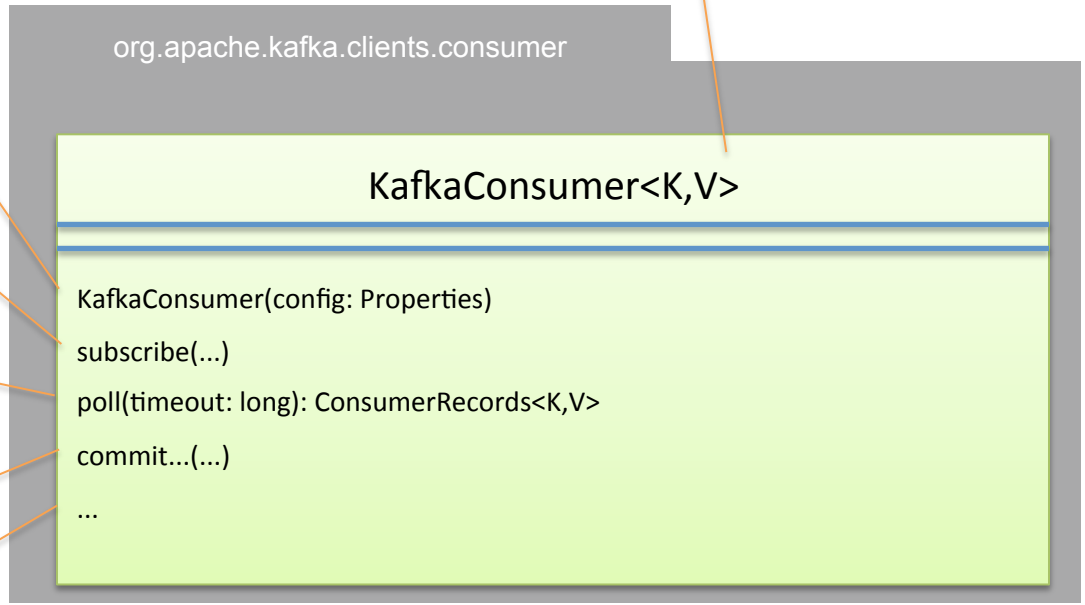
Generic sender where:
K = Type of key
V = Type of message

Constructor takes a configuration (mostly a hashmap of options)

Multiple ways to subscribe. Subscribe by list, wildcard, etc.

Read messages from Kafka

Multiple (sync/async) ways to confirm reception to consumer groups



A set of other methods such as: metrics(), pause(...), assign(...), close(), etc

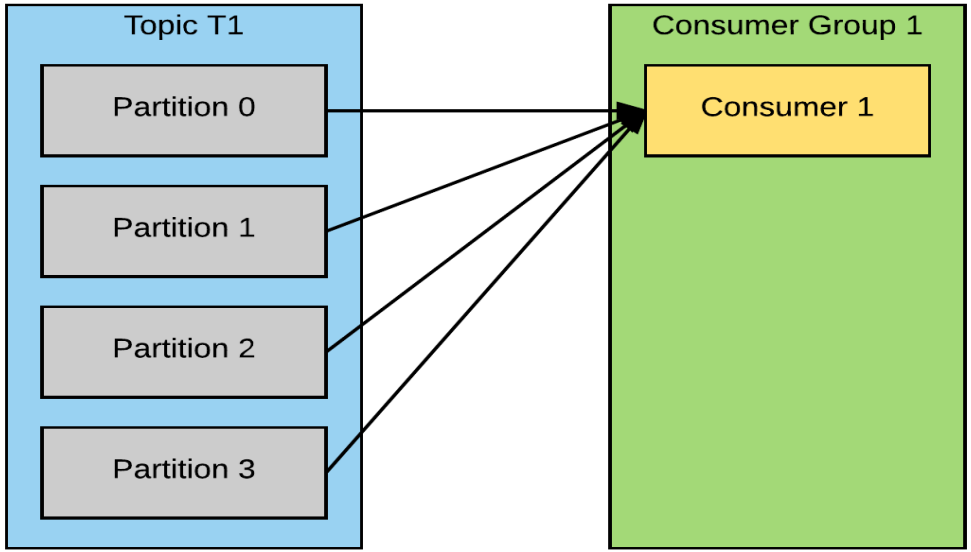
How to Use the API?

- The ***org.apache.kafka.clients.consumer.KafkaConsumer*** acts as a proxy for the consumer

- Some key issues to to resolve
  - Setup of consumer groups
  - Which topics to subscribe to
  - Which partitions to subscribe to (optional)
  - Manual or automatic offset management
  - Multi-threaded or single-threaded consumption

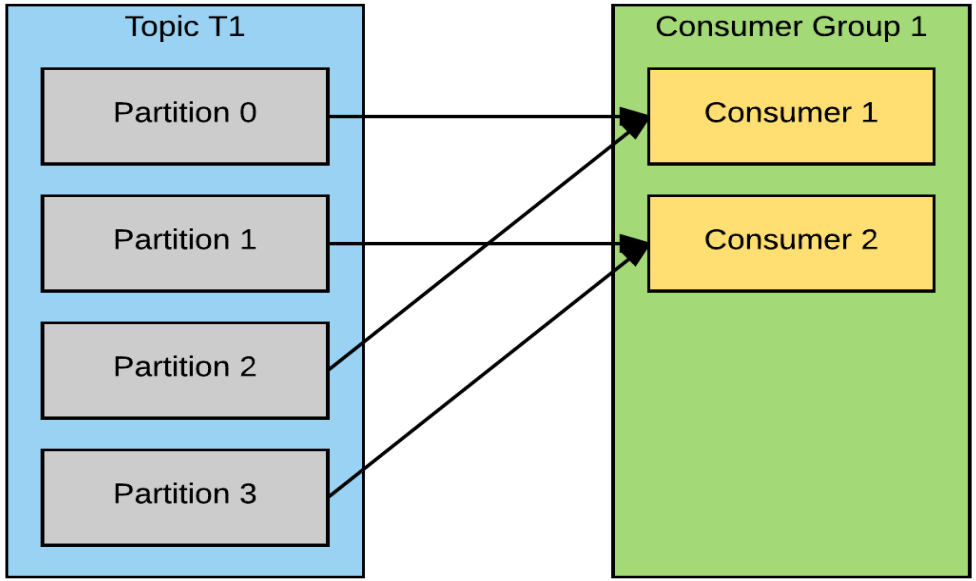
Consumer Group

- The consumer will receive all messages from all four partitions



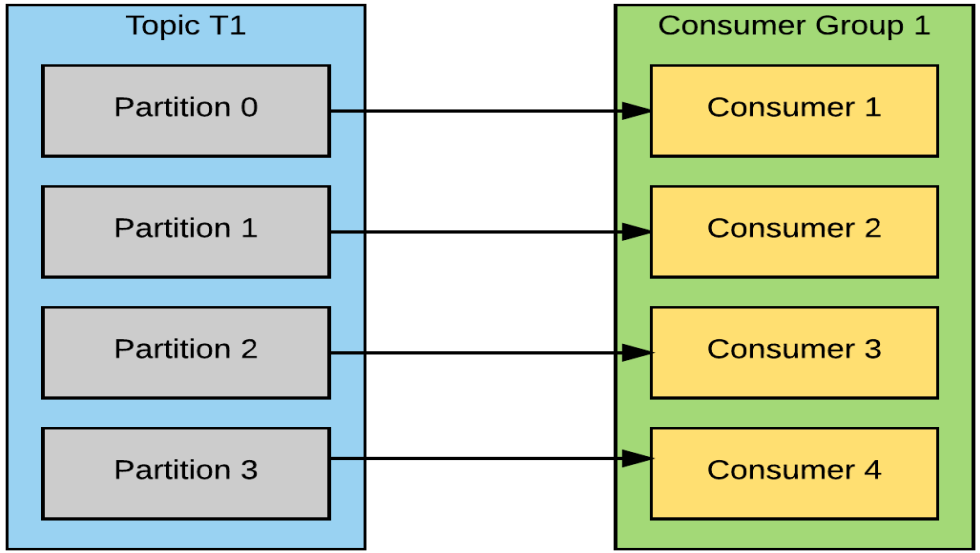
Consumer Group

- Each consumer will only get messages from two partitions



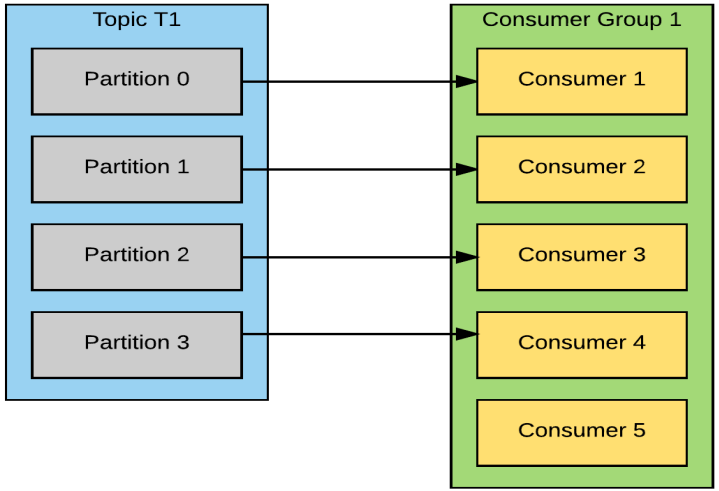
Consumer Group

- If the consumer group has the same number of consumers as partitions, each will read messages from a single partition



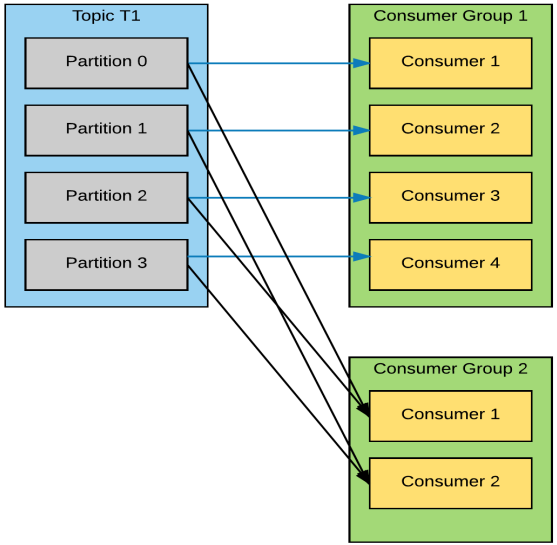
Consumer Group

- If there are more consumers than partitions for a topic, some consumers will be idle and receive no messages
- Create topics with a large number of partitions to allow adding more consumers when load increases



Multiple Consumer Groups

- One of the main design goals of Kafka was to allow multiple applications the ability to read data from the same topic
- Make sure each application has its own consumer group for this purpose



Partition Rebalance

- Consumers in a consumer group share ownership of the partitions in the topics they subscribe to
- When a new consumer is **added** to the group, it consumes messages from partitions which were previously consumed by another consumer
- When a consumer **leaves** the group (shuts down, crashes, etc), the partitions it used to consume will be consumed by one of the remaining consumers
- Reassignment of partitions to consumers can also happen when topics are modified – an administrator adds new partitions, for example

Partition Rebalance

- The event in which partition ownership is moved from one consumer to another is called a *rebalance*
- During a rebalance, consumers can't consume messages!
- Steps can be taken to safely handle rebalances and avoid unnecessary rebalances

Creating a Kafka Consumer

- Creating a `KafkaConsumer` is similar to creating a `KafkaProducer`
- Like the producer, you must specify `bootstrap.servers`, `key.deserializer`, and `value.deserializer` in a `Properties` object
- You must also specify a `group.id` which specifies the consumer group for which the `KafkaConsumer` instance belongs to

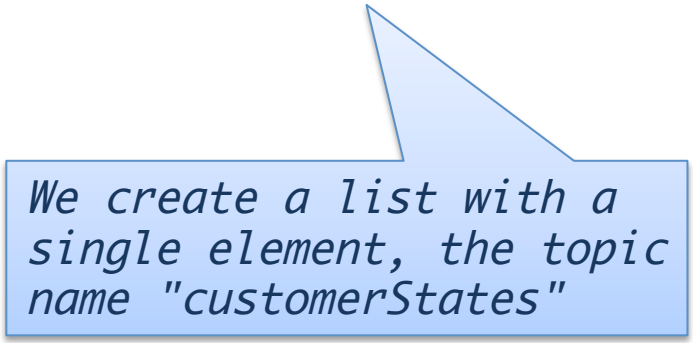
Setup of KafkaConsumer Example

```
private properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers", "mybroker1:9092,mybroker2:9092");

kakfaProps.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
kakfaProps.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("group.id", "StateCounter");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(kafkaProps);
```

Subscribing to Topics

- Once a consumer is created, you can subscribe to one or more topics

```
consumer.subscribe(Collections.singletonList("customerStates"));
```



Subscribing with Regular Expressions

- You can also subscribe to topics using regular expressions
- The expression can match multiple topic names
- If a new topic is created with a name that matches, a rebalance will happen and consumers will start consuming from the new topic
- Useful for applications that need to consume from multiple topics
- Example: subscribe to all test topics
  - `consumer.subscribe("test.*");`

Consumer Poll Loop

- Once a consumer subscribes to topics, a loop polls the server for more data

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(),
record.key(), record.value());
} finally {
  consumer.close();
}
```

Consumer Poll Loop

- Once a consumer subscribes to topics, a loop polls the server for more data

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : reco
    System.out.printf("offset = %d, key = %s, va                    ", record.offset(),
record.key(), record.value());
} finally {
  consumer.clo
}
```

Consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group

SciSpike

Poll Method

- `poll()` returns a list of records that contain:
  - Topic and partition the record came from
  - Offset of the record within the partition
  - Key and value of the record

- Takes a timeout parameter that specifies how long it will take to return, with or without data
  - *How fast do you want to return control to the thread that does the polling?*

Poll Internals

- Behind the scenes of `poll()`:
  - First time `poll()` is called with a new consumer: finds the `GroupCoordinator`, joins the consumer group, and receives a partition assignment
  - If a rebalance is triggered, it is also handled inside the poll loop
  - Heartbeats that keep consumers alive are sent

- Processing between iterations must be fast and efficient

Multithreading Considerations

- **One consumer per thread**
- To run multiple consumers in the same group in one application, each must run in its own thread
- You can wrap the consumer logic in its own object and then use Java's `ExecutorService` to start multiple threads each with its own consumer

Commits

- Unlike other JMS queues, Kafka does not track acknowledgements from consumers
- When `poll()` is called, it returns records that consumers in the group have not yet read

- The records that have been read by a consumer of the group are tracked by their position (offset) in each partition

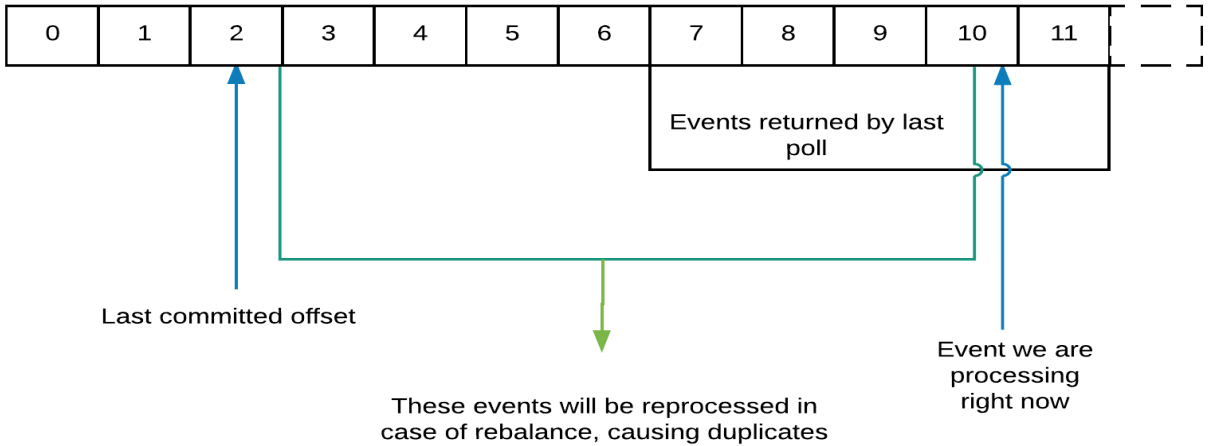
- When the current position in the partition is updated, it is known as a *commit*

Consumers Commit Offsets

- Consumers send a message to Kafka to a reserved topic with the committed offset for each partition
- If a consumer crashes or a new consumer joins the consumer group, a rebalance is triggered
- After a rebalance, a consumer may be assigned a new set of partitions and must read the latest committed offset of each partition and continue from there

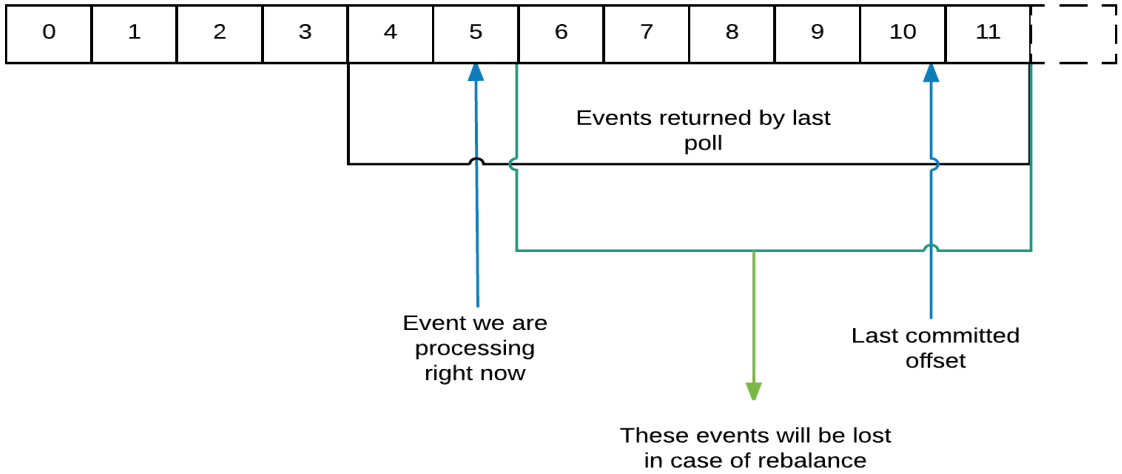
Messages Processed Twice

- If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice



Messages Missed

- If the committed offset is larger than the offset of the last message the client processed, messages between the last processed and the committed offset will be missed by the consumer group



Automatic Commit

- If you configure `enable.auto.commit=true`, the consumer will commit the largest offset your client received from `poll()` every 5 seconds by default
- Whenever there is a poll, the consumer checks if its time to commit and if so, will commit the offsets it returned in the last poll

- Convenient but don't give developers enough control to avoid duplicate messages

Commit Current Offset

- Developers usually want to exercise control over the time offsets are committed to eliminate possibility of missing messages **and** reduce the number of duplicate messages during rebalancing

- Setting `auto.commit.offset=false` means that offsets will only be committed explicitly

- Consumer has a `commitSync()` API to commit the latest offset returned by `poll()`

commitSync Example

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(),
record.key(), record.value());

  try {
    consumer.commitSync();
  } catch (CommitFailedException e) {
    log.error("commit failed", e);
  }

} finally {
  consumer.close();
```

Once we are done processing all records in the current batch, commitSync is called before polling for more

Asynchronous Commit

- The application is blocked until the broker responds to the commit request with commitSync() limiting throughput
- An alternative is to use commitAsync() which commits the last offsets and continues

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> rec
    System.out.printf("offset = %
record.key(), record.value());

  consumer.commitAsync();
}
```

You can also pass a callback which is invoked by the consumer when the commit finishes (either successfully or not)

SciSpike

Summary

- Producers
  - API
  - Sending messages
  - Serialization
- Consumers
  - API
  - Subscribing
  - Consumer groups
  - Rebalance