
kafka


# Module 2: Kafka Producer

# Objectives

**After completing of this module, you should be able to:**

- ✓ Configure Producers
- ✓ Construct Kafka Producer
- ✓ Send Messages to Kafka
- ✓ Synchronous and Asynchronous messages
- ✓ Serialize Messages using Avro
- ✓ Create and handle Partitions

# Let’s see how to configure Single-Node Multi-broker Kafka Cluster

# Objectives

**After completing of this module, you should be able to:**

- ✓ Configure Producers
- ✓ Construct Kafka Producer
- ✓ Send Messages to Kafka
- ✓ Synchronous and Asynchronous messages
- ✓ Serialize Messages using Avro
- ✓ Create and handle Partitions

# Multi-Broker Cluster Setup

For *setting up multiple brokers* on a *single node*, separate *server property files* are required for *each broker*.
Each property file will *define different values* for the following properties: *broker.id, listeners, log.dir*

***Steps:***

- Go to Kafka directory
- Open ***config*** folder
- Make two separate server property files in config folder
- Make changes in the files created as shown on next slide

# Change in Server Property Files

server-1.properties

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1

# Switch to enable topic deletion or not, default value is false
#delete.topic.enable=true

############################# Socket Server Settings #############################

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://localhost:9093

############################# Log Basics #############################

# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-1

# The default number of log partitions per topic. More partitions allow greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=1
```

server-2.properties

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=2

# Switch to enable topic deletion or not, default value is false
#delete.topic.enable=true

############################# Socket Server Settings #############################

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://localhost:9094

############################# Log Basics #############################

# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-2

# The default number of log partitions per topic. More partitions allow greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=1
```

Now we start each new broker in a separate console window

# Let’s take a look at different Producer Application Use-cases

# Kafka Producer Applications Use-cases

1 Recording User Activities for Auditing or Analysis


Recording Metrics 2

3 Storing Log Messages


# Kafka Producer Applications Use-cases

4 Recording Information from Smart Appliances

Communicating Asynchronously with other Applications 5

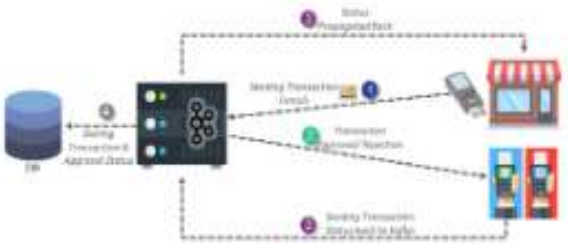
6 Buffering Information before Writing to a Database

# Let’s take a look at different Kafka Producer Scenarios

# Kafka Producer Scenarios

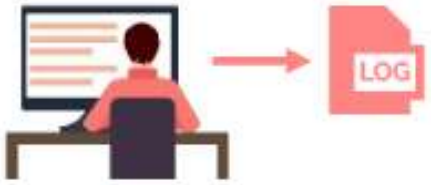
*Scenario 1: Credit Card Transaction Processing*

- It is critical, since you never lose a single message nor duplicate any messages
- Latency should be low (can be tolerated up to 500ms)
- Throughput should be very high (process a million messages a second)



*Scenario 2: Clickstream Analysis*

- Loss or a few duplicate messages can be tolerated
- Latency can be high as long as there is no impact on the user experience
- Throughput will depend on the level of activity




Different requirements will influence the way you use the producer API to write messages to Kafka and the configuration you use.

# High Level Architecture of Kafka Producer

# High Level Architecture of Kafka Producer

We can start producing messages to Kafka by creating a *ProducerRecord*


# High Level Architecture of Kafka Producer

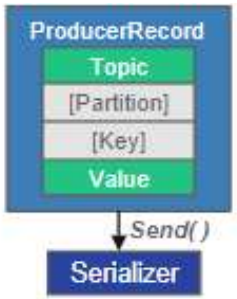
It must include the *topic* we want to send the record to and a *value*


# High Level Architecture of Kafka Producer

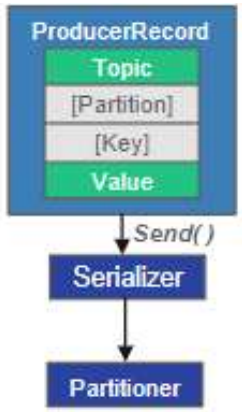
We can also specify a *key* and/or a *partition*


# High Level Architecture of Kafka Producer

First the producer will *serialize* the key and value objects to *ByteArrays*




# High Level Architecture of Kafka Producer

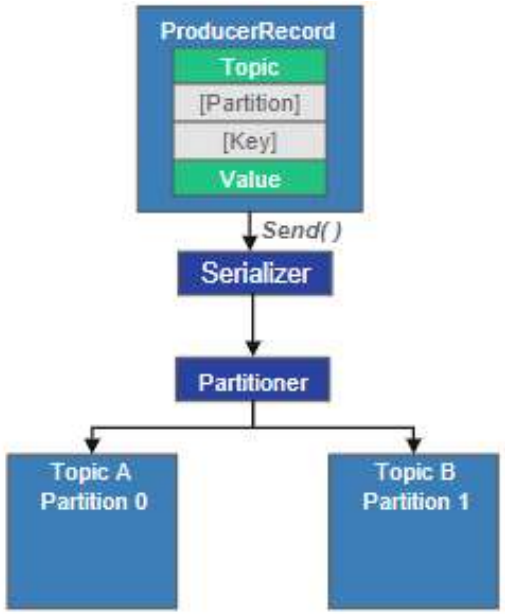



Next, the data is sent to a *partitioner*

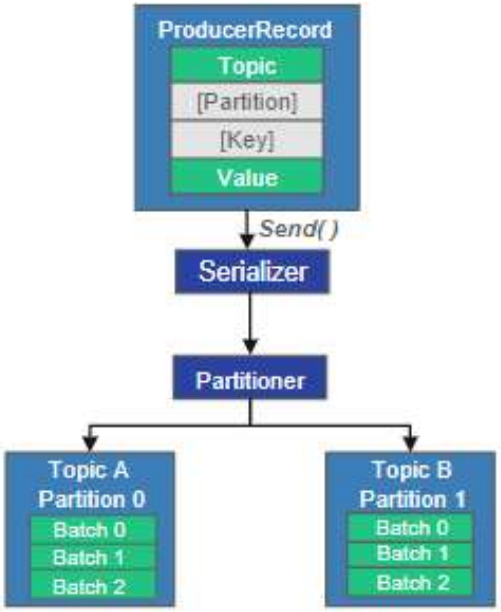
# High Level Architecture of Kafka Producer

If partition is specified in *ProducerRecord*, the *partitioner* returns the *partition* we specified

As partition is selected, producer knows the topic and partition where the record will go

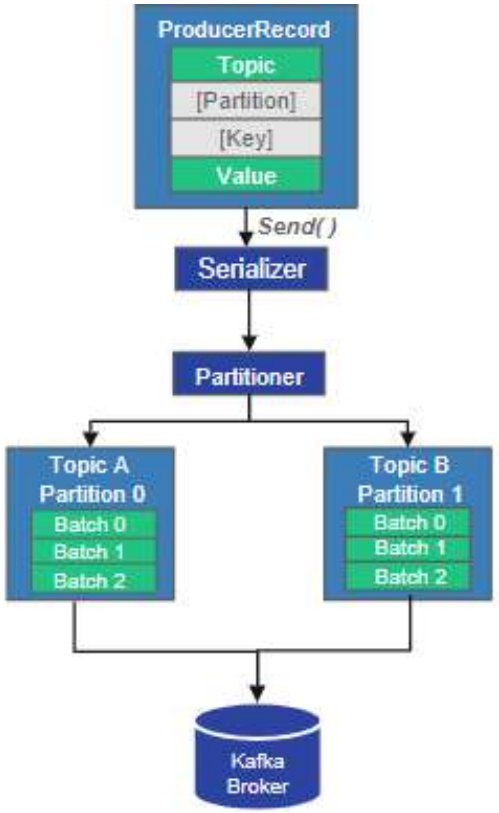



# High Level Architecture of Kafka Producer




Adds the record to a *batch of records* that will also be sent to the same topic and partition

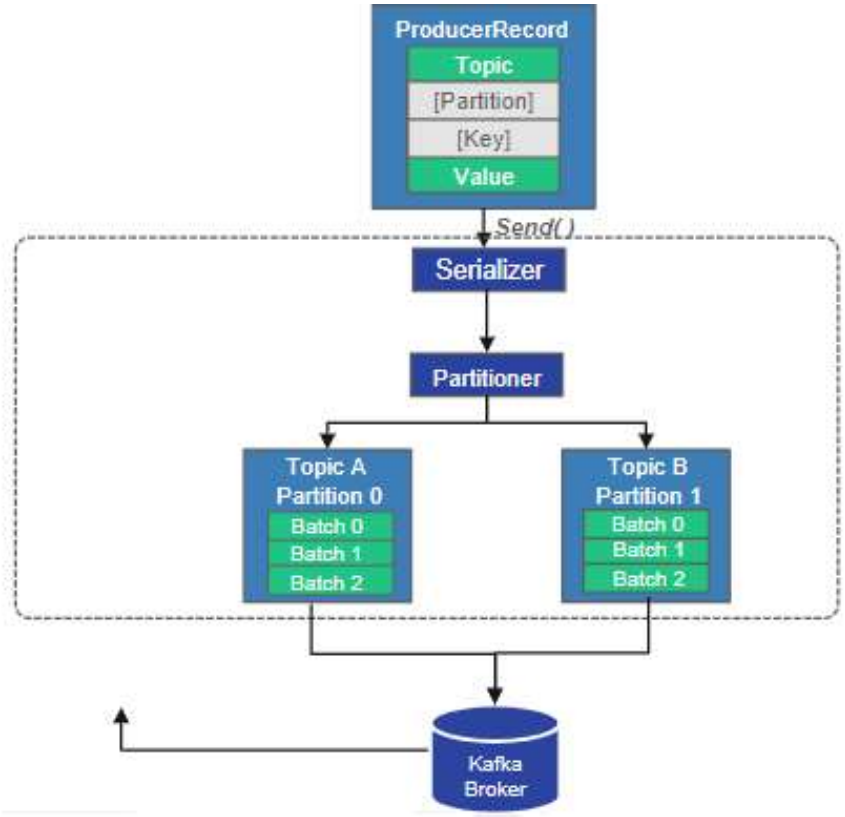
# High Level Architecture of Kafka Producer




Separate thread is responsible for sending those batches of records to the appropriate *Kafka brokers*

# High Level Architecture of Kafka Producer

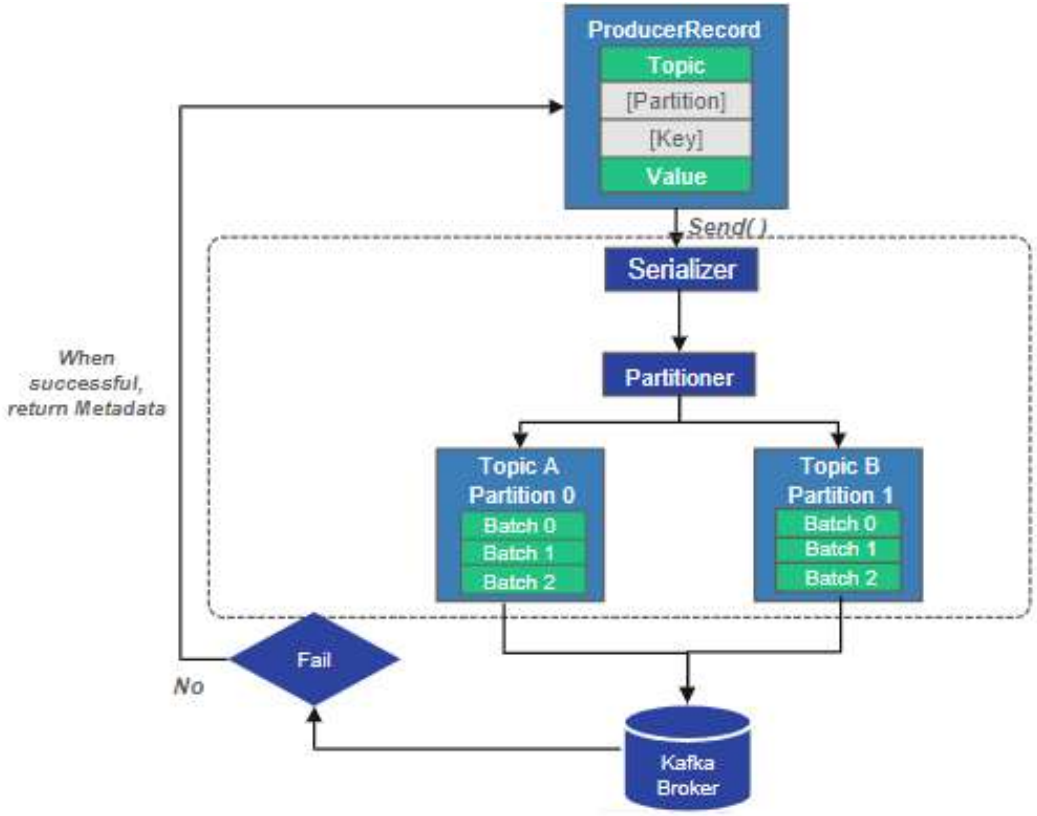
When the broker receives the messages, *it sends back a response*




# High Level Architecture of Kafka Producer

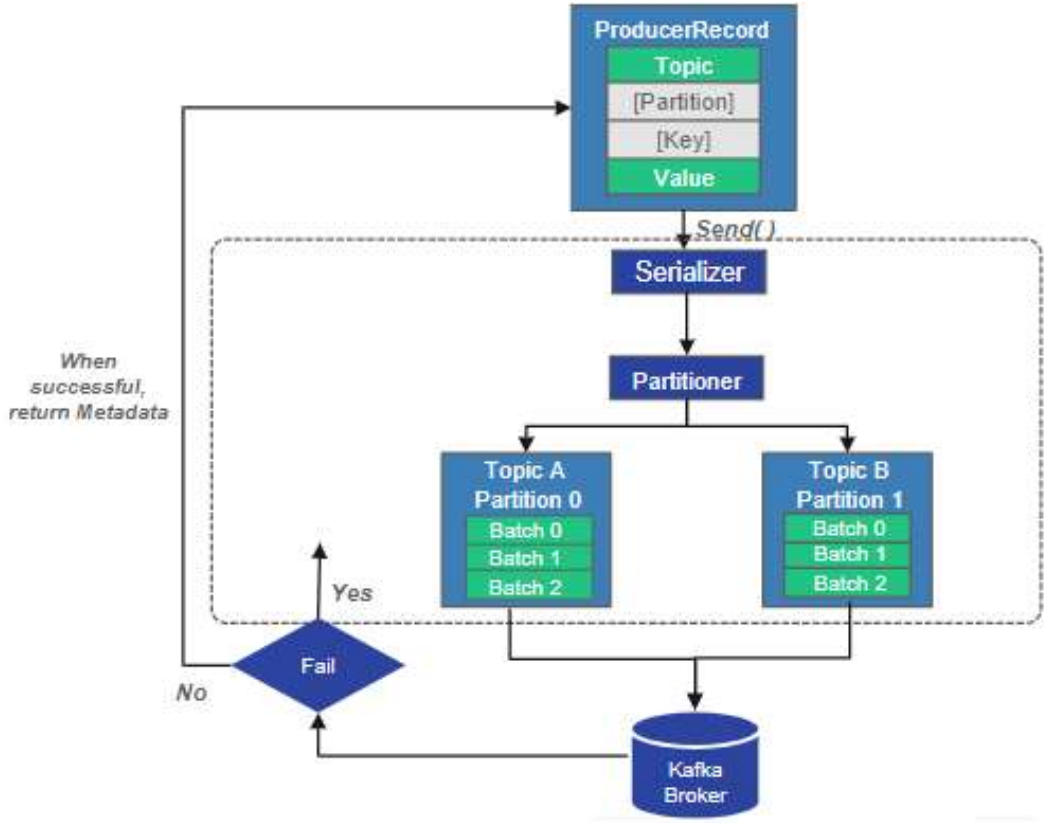
If the messages were successfully written, it will return a *RecordMetadata*

*RecordMetadata* contains the topic, partition, and the offset of the record within the partition



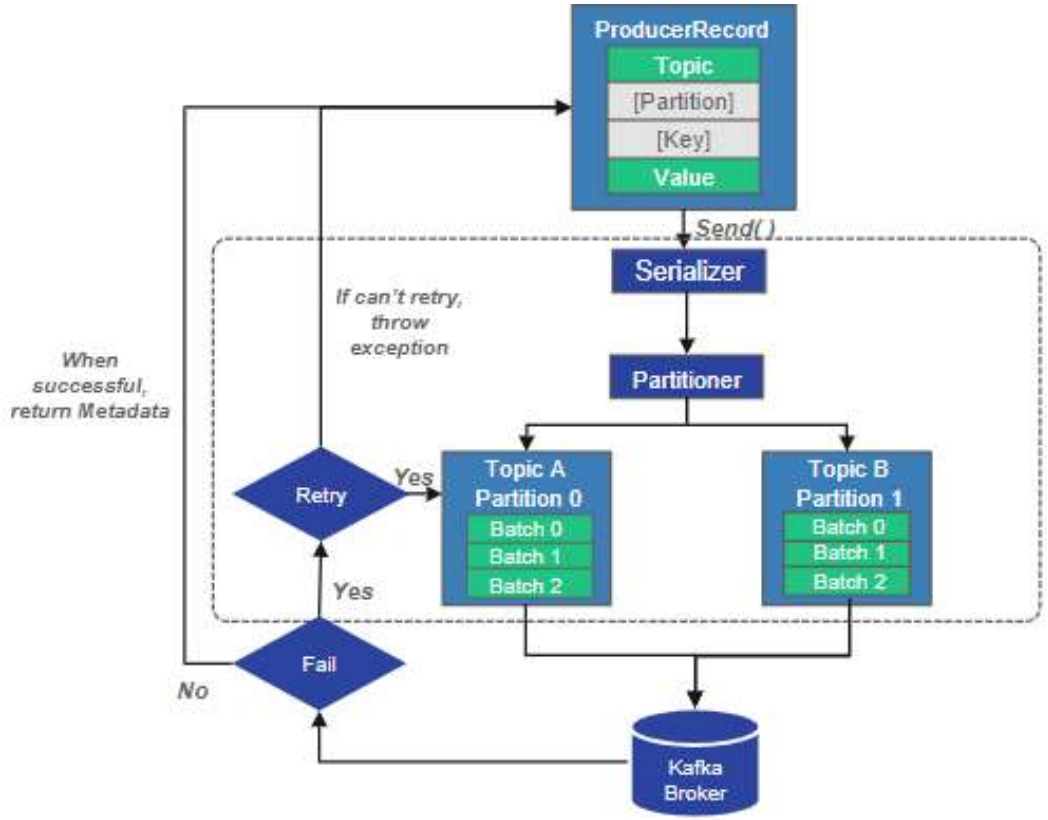

# High Level Architecture of Kafka Producer

If the *broker failed* to write the messages, it will *return an error*




# High Level Architecture of Kafka Producer

When the producer receives an error, it *retries sending the message* a few more times before returning an error



# Let’s have a look at some important Kafka Producer Configuration Properties

# Kafka Producer Configurations

Kafka producer has three mandatory properties:

***bootstrap.servers*** ***key.serializer*** ***value.serializer***

- List of *host:port* pairs of brokers, that the producer will use to establish initial connection
- No need to include all brokers, producer will get information after the initial connection
- Recommended to include at least two, in case one broker goes down

# Kafka Producer Configurations

Kafka producer has three mandatory properties:

**bootstrap.servers**

**key.serializer**

**value.serializer**

- Name of a class that will be used to serialize the keys of the records
- Brokers expect byte arrays as keys and values of messages
- Serializer class should implements *org.apache.kafka.common.serialization.Serializer* interface.
- Producer will use this class to serialize the key object to a byte array
- Kafka client package includes ByteArraySerializer, StringSerializer, and IntegerSerializer,
- Setting *key.serializer* is required even if you intend to send only values

# Kafka Producer Configurations

Kafka producer has three mandatory properties:

**bootstrap.servers**

**key.serializer**

**value.serializer**

- Name of a class that will be used to serialize the values of the records
- Similarly as you set *key.serializer* to a name of a class
- Set *value.serializer* to a class that will serialize the message value object

# Let’s see how to create a Kafka Producer

# Create a Kafka Producer

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers", "broker1:9092, broker2:9093");
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);
```

We start with a Properties object

Here we Create a new producer by setting the appropriate key and value types & passing the Properties object

Here we are using the built-in *StringSerializer*

Let’s take a look at different types of errors

# Types of Errors

*KafkaProducer* has two types of errors:

**Retriable Error**

- *Retriable* errors are those that can be resolved by sending the message again
  - For example, a connection error can be resolved by reestablishing a connection, "no leader" error can be resolved when a new leader is elected for the partition
- *KafkaProducer* can be configured to retry those errors automatically

**Non-Retriable Error**

- Some errors will not be resolved by retrying
- In those cases, KafkaProducer will not attempt a retry & will return the exception immediately
  - For example, "message size too large"

Let’s see different ways to send messages

# Ways to send Messages - Fire & Forget

Kafka producer has three ways for sending messages:

***Fire-and-forget*** ***Synchronous send*** ***Asynchronous send***

- We send a message to the server and don't really care if it arrives successfully or not
- Generally, messages arrive successfully, as Kafka is highly available
- Producer will retry sending messages automatically
- Some messages will get lost using this method

# Ways to send Messages – Synchronous Send

Kafka producer has three ways for sending messages:

***Fire-and-forget***

***Synchronous send***

***Asynchronous 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

# Ways to send Messages – Asynchronous Send

Kafka producer has three ways for sending messages:

***Fire-and-forget***

***Synchronous send***

***Asynchronous send***

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

# Sending Message in Fire & Forget way

```
ProducerRecord<String, String> record =
                    new ProducerRecord<>("Employee", "Name", "John");

try {
                    producer.send(record); }

catch (Exception e) {
                    e.printStackTrace(); }
```

- Producer accepts *ProducerRecord* objects
- *ProducerRecord* has multiple constructors
- Requires the name of the topic where we are sending data
- Always a string, key and value are also strings
- Key and value must match our serializer and producer objects

# Sending Message in Fire & Forget way

```
ProducerRecord<String, String> record =
                    new ProducerRecord<>("Employee", "Name", "John");

try {
                    producer.send(record); }

catch (Exception e) {
                    e.printStackTrace(); }
```

- Use producer object send() method to send the *ProducerRecord*
- Message will be placed in a buffer and will be sent to the broker in a separate thread
- *send()* method returns a Java Future object with *RecordMetadata*
- *RecordMetadata* tells whether the message was sent successfully or not

# Sending Message in Fire & Forget way

```
ProducerRecord<String, String> record =
                    new ProducerRecord<>("Employee", "Name", "John");

try {
                    producer.send(record); }

catch (Exception e) {
                    e.printStackTrace(); }
```

- Use producer object send() method to send the *ProducerRecord*
- Message will be placed in a buffer and will be sent to the broker in a separate thread
- *send()* method returns a Java Future object with *RecordMetadata*
- *RecordMetadata* tells whether the message was sent successfully or not

# Sending Message Synchronously

```
ProducerRecord<String, String> record = new ProducerRecord<>("Employee", "Name", "James");

try {
        producer.send(record).get();
}

catch (Exception e) {
                e.printStackTrace();
}
```

- *Future.get()* is used to wait for a reply from Kafka
- If the record is not sent successfully, method will throw an exception
- If there were no errors, it returns a *RecordMetadata* object

# Sending Message Synchronously

```
ProducerRecord<String, String> record = new ProducerRecord<>("Employee", "Name", "James");

try {
                producer.send(record).get();
}

catch (Exception e) {
                e.printStackTrace();
}
```

- It prints any exception, that has been occurred
- It can be errors before sending data, a nonretriable exceptions or available retries is exhausted

# Sending Message 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<>("Employee", "Name", "Jordan");

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

- To use callbacks, a class is needed that implements the *org.apache.kafka.clients.producer.Callback* interface
- It has a single function—*onCompletion()*

# Sending Message 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<>("Employee", "Name", "Jordan");

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

- If Kafka returned an error, *onCompletion*() will have a nonnull exception
- Production code will probably have more robust error handling functions

# Sending Message 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<>("Employee", "Name", "Jordan");

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

- We pass a *Callback object* along when sending the record

# More properties to configure Kafka Producers ..

# Configuring Kafka Producers

**ACKS**

It controls how many partition replicas must receive the record before the producer can consider the write successful
Significant impact on how likely messages are to be lost

*There are three allowed values for the acks parameter:*

***acks=0***

- Producer will not wait for a reply from the broker before assuming the message was sent successfully
- If something went wrong and the broker did not receive the message
- Producer will not know about failure and the message will be lost
- Producer sends messages as fast as the network will support, it gives very high throughput

***acks=1***

- Producer will receive a success response from the broker after leader replica receives the message
- If the message can't be written to the leader, the producer will receive an error response
- It can retry sending the message, avoiding potential loss of data
- Throughput depends on whether we send messages synchronously or asynchronously

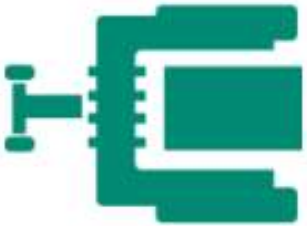
***acks=all***

- Producer will receive a success response from the broker once all in-sync replicas received the message
- It's the safest mode since you can make sure more than one broker has the message
- Message will survive even in the case of crash

# Configuring Kafka Producers

**buffer.memory**

- Configures the amount of memory, producer will use to buffer messages before sending
- If messages are sent faster than they are delivered, producer may run out of space
- *Additional send() calls will either block or throw an exception (max.block.ms parameter)*



**compression.type**

- By default, messages are sent uncompressed
- Compression algorithms will be used to compress the data before sending it to the brokers
- *It could be snappy, gzip, or lz4*

# Configuring Kafka Producers

**retries**

- If error is transient, it tells how many times the producer will retry sending the message
- By default, the producer will wait 100ms between retries
- Could be controlled by *retry.backoff.ms* parameter

**client.id**

- Used by the brokers to identify messages sent from the client
- Used in logging and metrics, and for quotas

**receive.buffer.bytes and send.buffer.bytes**

- Sizes of the TCP send and receive buffers used by the sockets when writing/reading data
- If these are set to -1, the OS defaults will be used

# Configuring Kafka Producers

**max.in.flight.requests.per.connection**

- Controls how many messages the producer will send to the server without receiving responses
- Setting this high can increase memory usage while improving throughput
- Setting it too high can reduce throughput as batching becomes less efficient
- Setting to 1 will guarantee that messages will be written to the broker in the order they were sent

**request.timeout.ms**

- Controls how long the producer will wait for a reply from the server when sending data & requesting metadata
- If timeout is reached without reply, the producer will either retry sending or respond with an error

# Configuring Kafka Producers

**max.block.ms**

- Controls how long the producer will block when calling send() & when explicitly requesting metadata via partitionsFor()
- Those methods block when the producer's send buffer is full or when metadata is not available
- When max.block.ms is reached, a timeout exception is thrown

**max.request.size**

- This setting controls the size of a produce request
- Caps both the size of the largest message that can be sent & the size of messages in a batch that the producer can send in one request

# Serialization – What & How ?

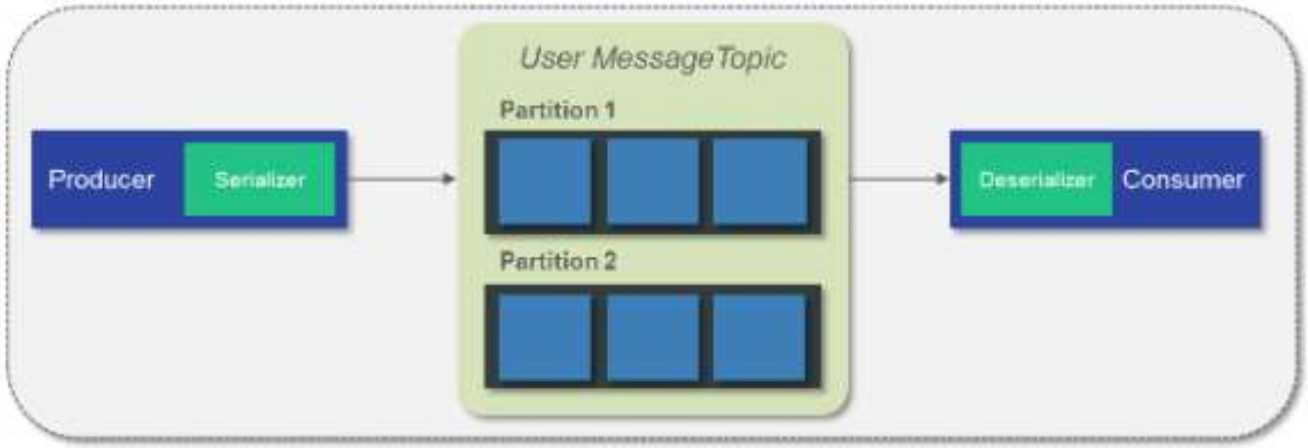
# Serializers

**Serialization** is the process of translating data structures or object state into a format that can be stored, transmitted & reconstructed later

Producer configuration includes mandatory serializers

Default String serializer can be used

Kafka also includes serializers for integers and ByteArrays




# Custom Serializers

## Custom Serializers

- If the object you need to send to Kafka is not a simple string or integer, you need *custom serializer*
- Can use generic serialization library like Avro, Thrift, or Protobuf to create records
- Can create a custom serialization for objects
- Creating a simple class to represent students:

```
public class Student {
            private int studentID;
            private String studentName;

            public Student(int ID, String name) {
                        this. studentID = ID;
                        this. studentName = name; }

            public int getID() {
                        return studentID; }

            public String getName() {
                        return studentName; }
}
```

# Creating Custom Serializer

E.g. - Creating a custom serializer for Student class

```
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;

public class StudentSerializer implements Serializer<Student> {

            @Override
  public void configure(Map configs, boolean isKey) {
            // nothing to configure
  }

  @Override
  /**
  We are serializing Student as:
  4 byte int representing studentId
  4 byte int representing length of studentName in UTF-8 bytes (0 if name is Null)
  N bytes representing studentName in UTF-8
  */
```

# Creating Custom Serializer

E.g. - Creating a custom serializer for Student class

```
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;

public class StudentSerializer implements Serializer<Student> {

        @Override
  public void configure(Map configs, boolean isKey) {
        // nothing to configure
  }

  @Override
  /**
  We are serializing Student as:
  4 byte int representing studentId
  4 byte int representing length of studentName in UTF-8 bytes (0 if name is Null)
  N bytes representing studentName in UTF-8
  */
```

- Configuring a producer with this *StudentSerializer* will allow you to define *ProducerRecord<String, Student>*
- Send *Student* data and pass *Student* objects directly to the producer

# Creating Custom Serializer

E.g. - Creating a custom serializer for Student class

```
public byte[] serialize(String topic, Student data) {
            try {
                        byte[] serializedName;
                        int stringSize;
      if (data == null)
        return null;
      else {
                                              if (data.getName() != null) {
          serializedName = data.getName().getBytes("UTF-8");
          stringSize = serializedName.length;
                                              } else {
                                                        serializedName = new byte[0];
                                                        stringSize = 0;
                                              }
                        }
```

# Creating Custom Serializer

E.g. - Creating a custom serializer for Student class

```
 ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
      buffer.putInt(data.getID());
      buffer.putInt(stringSize);
      buffer.put(serializedName);

      return buffer.array();
             } catch (Exception e) {
             throw new SerializationException("Error when serializing Student to byte[] " + e);
    }
  }

  @Override
  public void close() {
             // nothing to close
  }
}
```

# Serializer Challenges

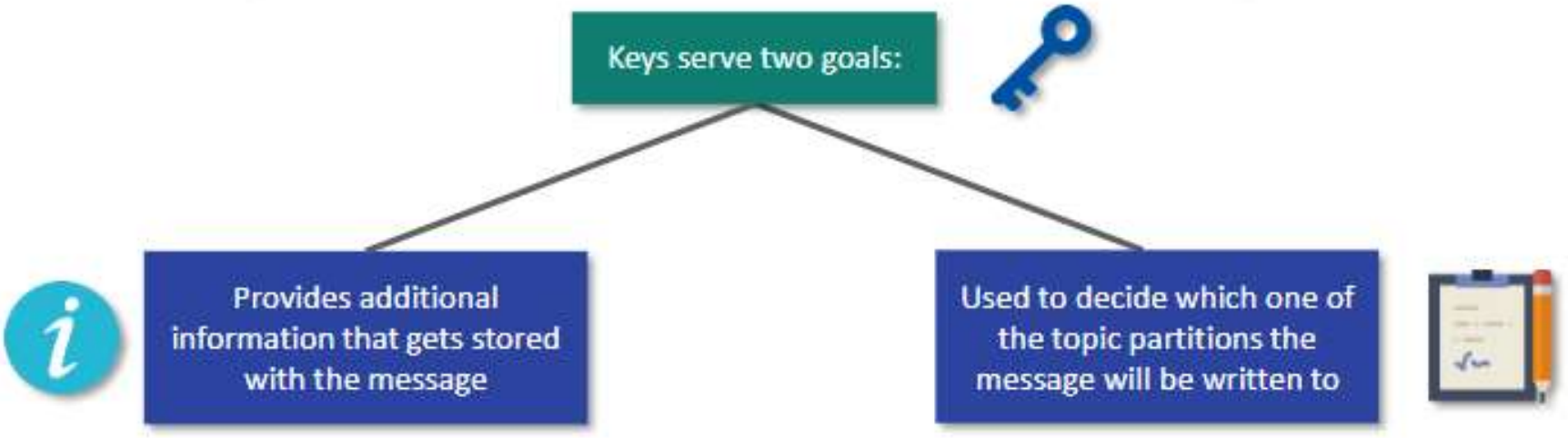
If we need to change studentID to Long, or add a startDate field to *Student*, there will be compatibility issues between old and new messages.

- Debugging compatibility issues is fairly challenging
- If multiple teams are writing Student data to Kafka, they all need to use the same serializers & deserializer
- It's recommended using existing serializers and deserializers such as JSON, Apache Avro, Thrift, or Protobuf


Let’s take a look at What is a Partition?

# Partitions

- *ProducerRecord* objects includes a topic name, key & value
- Kafka messages are key-value pairs
- *ProduceRecord* object can be with just a topic and a value
- Key is set to null by default
- Most applications produce records with keys




# Partitions

- All messages with the same key will go to the same partition.
- All the records for a single key will be read by the same process.

To create a key-value record, you simply create a ProducerRecord as follows:

```
ProducerRecord<Integer, String> record = new ProducerRecord<>("Employee", "Name", "James");
```

While creating messages with a null key, you can simply leave the key out:

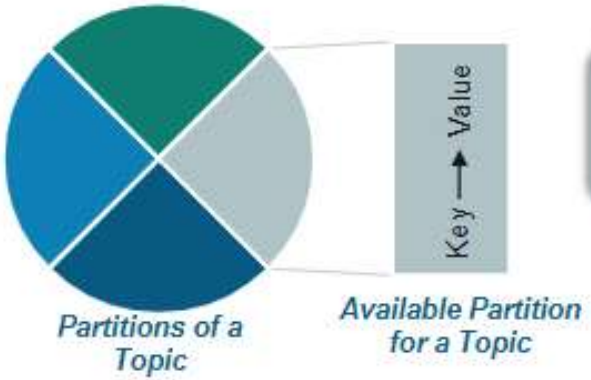
```
ProducerRecord<Integer, String> record = new ProducerRecord<>("Employee", "James");
```

Here, the key will simply be set to *null*, which may indicate that a student name was missing on a form.

# Partitions

If key is null and the default partitioner is used, the record is sent to one of the available partitions of the topic at random

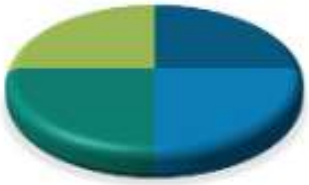
A round-robin algorithm will be used to balance the messages among the partitions




It's important that a key is always mapped to the same partition

Kafka hash the key, and use the result to map the message to a specific partition

# Partitions



As the number of partitions is constant, records of a user will be written in a particular partition

This allows all kinds of optimization when reading data from partitions


If you add *new partitions* to the topic, old records can change partition

New records will get written to a different partition

If partitioning keys is important, create topics with sufficient partitions and never add partitions

# Implementing a Custom Partitioning Strategy

Kafka allow to partition data differently.

**Scenario**

- There is a company that does so much business with their device called *"Calacs"*, that over 10% of their daily transactions are with this device.
- If you use default hash partitioning, the *Calacs* records will get allocated to the same partition as other accounts, resulting in one partition being about twice as large as the rest.

**Problem**

This can cause servers to run out of space and slows down processing.

**Solution**

To solve this problem, we need to provide *Calacs* its own partition and then use hash


# Implementing a Custom Partitioning Strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;
public class CalacsPartitioner implements Partitioner {
            public void configure(Map<String, ?> configs) {}
            public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
                        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
                        int numPartitions = partitions.size();
                        if ((keyBytes == null) || (!(key instanceOf String)))
                                    throw new InvalidRecordException("We expect all messages to have
                                    student name as key")
                        if (((String) key).equals("Calacs"))
                                    return numPartitions; // Calacs will always go to last partition
                        // Other records will get hashed to the rest of the partitions
            return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) }

            public void close() {} }
```

# Implementing a Custom Partitioning Strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;
public class CalacsPartitioner implements Partitioner {
            public void configure(Map<String, ?> configs) {}
            public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
                        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
                        int numPartitions = partitions.size();
                        if ((keyBytes == null) || (!(key instanceOf String)))
                                    throw new InvalidRecordException("We expect all messages to have
                                    student name as key")
                        if (((String) key).equals("Calacs"))
                                    return numPartitions; // Calacs will always go to last partition
                        // Other records will get hashed to the rest of the partitions
            return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) }

            public void close() {} }
```

Creating CalcsPartitioner

# Implementing a Custom Partitioning Strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;
public class CalacsPartitioner implements Partitioner {
            public void configure(Map<String, ?> configs) {}
            public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
                        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
                        int numPartitions = partitions.size();
                        if ((keyBytes == null) || (!(key instanceOf String)))
                                    throw new InvalidRecordException("We expect all messages to have
                                    student name as key")
                        if (((String) key).equals("Calacs"))
                                    return numPartitions; // Calacs will always go to last partition
                        // Other records will get hashed to the rest of the partitions
            return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) }

            public void close() {} }
```

*Calacs* will always go to last partition

# Implementing a Custom Partitioning Strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;
public class CalacsPartitioner implements Partitioner {
            public void configure(Map<String, ?> configs) {}
            public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
                        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
                        int numPartitions = partitions.size();
                        if ((keyBytes == null) || (!(key instanceOf String)))
                                    throw new InvalidRecordException("We expect all messages to have
                                    student name as key")
                        if (((String) key).equals("Calacs"))
                                    return numPartitions; // Calacs will always go to last partition
                        // Other records will get hashed to the rest of the partitions
            return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) }

            public void close() {} }
```

Other records will get hashed to the rest of the partitions


Thank you!