Chapter 3. Kafka Producers - Writing Messages to Kafka

Whether you use Kafka as a queue, a message bus or a data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka or an application that serves both roles.

For example, in a credit card transaction processing system, there will be a client application, perhaps an online store, responsible for sending each transaction to Kafka immediately when a payment is made. Another application is responsible for immediately checking this transaction against a rules engine and determining whether the transaction is approved or denied. The approve / deny response can then be written back to Kafka and the response can propagate back to the online store where the transaction has been initiated. A third application can read both transactions and the approval status from Kafka and store them in a database where analysts can later review the decisions and perhaps improve the rules engine.

Apache Kafka ships with built in client APIs that developers can use when developing applications that interact with Kafka.

In this chapter we will learn how to use the Kafka Producer, starting with an overview of its design and components. We will show how to create a KafkaProducer and ProducerRecord objects, how to send records to Kafka and how to handle the errors that Kafka may return. We'll then review the most important configuration options used to control the producer behavior. We'll conclude with a deeper look at how to use different partitioning methods and serializers, and how to write your own serializers and partitioners.

In the next chapter we will look at Kafka's consumer client and reading data from Kafka.

> **NOTE**
>
> In addition to the built-in clients, Kafka has a binary wire protocol. This means that it is possible for applications to read messages from Kafka or write messages to Kafka simply by sending the correct byte sequences to Kafka's network port. There are multiple clients that implement Kafka's wire protocol in different programming language, giving simple ways to use Kafka not just in Java applications but also in languages like C++, Python, Go and many more. Those clients are not part of Apache Kafka project, but a list of those is maintained in the project wiki. The wire protocol and the external clients are outside the scope of the chapter.

Producer overview

There are many reasons an application will need to write messages to Kafka: Recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, asynchronous communication with other applications, buffering information before writing to a database and much more.

Those diverse use-cases also imply diverse requirements: Is every message critical, or can we tolerate loss of messages? Are we ok with accidentally duplicating messages? Are there any strict latency or throughput requirements we need to support?

In the credit-card transaction processing example we introduced earlier, we can see that it will be critical to never lose a single message nor duplicate any messages, latency should be low but latencies up to 500ms can be tolerated, and throughput should be very high - we expect to process up to a million messages a second.

A different use-case can be to store click information from a website. In that case, some message loss or few duplicates can be tolerated, latency can be high - as long as there is no impact on the user experience - in other words, we don't mind if it takes few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicked on a link. Throughput will depend on the level of activity we anticipate on our website.

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

While the producer APIs are very simple, there is a bit more that goes on under the hood of the producer when we send data. In Figure 3-1 you can see the main steps involved in sending data to Kafka.

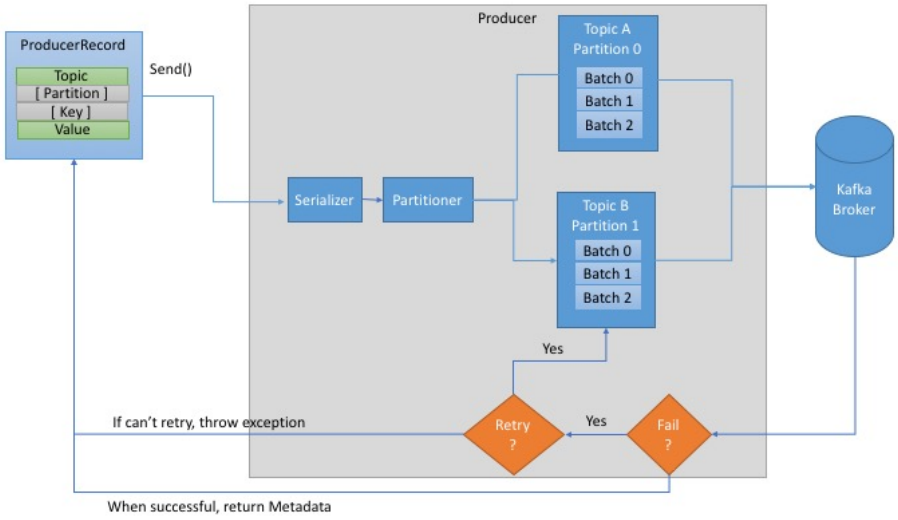

Figure 3-1. High level overview of Kafka Producer components

We start by creating a ProducerRecord, which must include the topic we want to send the record to and a value we are sending. Optionally, we can also specify a key and / or a partition. Once we send the ProducerRecord, the first thing the producer will do is serialize the key and value objects to ByteArrays, so they can be sent over the network.

Next, the data is sent to a partitioner. If we specified a partition in the ProducerRecord, the partitioner doesn't do anything and simply returns the partition we specified. If we didn't, the partitioner will choose a partition for us, usually based on the ProducerRecord key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a RecordMetadata object with the topic, partition and the offset of the record within the partition. If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message few more times before giving up and returning an error.

Constructing a Kafka Producer

The first step in writing messages to Kafka is to create a producer object with the properties you want to pass to the producer. Kafka producer has 3 mandatory properties:

- `bootstrap.servers` - List of host:port pairs of Kafka brokers. This doesn't have to include all brokers in the cluster, the producer will query these brokers for information about additional brokers. But it is recommended to include at least two, so in case one broker goes down the producer will still be able to connect to the cluster.
- `key.serializer` - Kafka brokers expect byte arrays as key and value of messages. However the Producer interface allows, using parameterized types, to send any Java object as key and value. This makes for very readable code, but it also means that the Producer has to know how to convert these objects to byte arrays. `key.serializer` should be set to a name of a class that implements `org.apache.kafka.common.serialization.Serializer` interface and the Producer will use this class to serialize the key object to byte array. The Kafka client package includes `ByteArraySerializer` (which doesn't do much), `StringSerializer` and `IntegerSerializer`, so if you use common types, there is no need to implement your own serializers. Setting `key.serializer` is required even if you intend to send only values.
- `value.serializer` - the same way you set `key.serializer` to a name of a class that will serialize the message key object to a byte array, you set `value.serializer` to a class that will serialize the message value object. The serializers can be identical to the `key.serializer`, for example when both key and value are Strings or they can be different, for example Integer key and String value.

The following code snippet shows how to create a new Producer by setting just the mandatory parameters and using default for everything else:

```
private Properties kafkaProps = new Properties(); ❶
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); ❸
```

❶ We start with a Properties object

❷ Since we are planning on using Strings for message key and value, we use the built-in StringSerializer

❸ Here we create a new Producer by setting the appropriate key and value types and passing the Properties object

With such a simple interface, it is clear that most of the control over Producer behavior is done by setting the correct configuration properties. Apache Kafka documentation covers all the configuration options (http://kafka.apache.org/documentation.html#producerconfigs), and we will go over the important ones later in this chapter.

Once we instantiated a producer, it is time to start sending messages. There are three primary methods of sending messages:

- Fire-and-forget - in which we send a message to the server and don't really care if it arrived 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 receive a response from the Kafka broker.

In the examples below we will see how to send messages using the methods we mentions and how to handle the different types of errors that could occur.

While all the examples in this chapter are single threaded, a producer object can be used by multiple threads to send messages. You will probably want to start with one producer and one thread. If you need better throughput, you can add more threads that use the same producer. Once this ceases to increase throughput, you can add more producers to the application to achieve even higher throughput.

Sending a Message to Kafka

The simplest way to send a message is as follows:

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "France"); ❶
try {
  producer.send(record); ❷
} catch (Exception e) {
        e.printStackTrace(); ❸
}
```

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.

❷ We use the Producer object send() method to send the ProducerRecord. As we've seen in the Producer architecture diagram, the message will be placed in a buffer and will be sent to the broker in a separate thread. The send() method returns a Java Future object (http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) 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.

❸ 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 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.

Sending a Message Synchronously

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
        producer.send(record).get(); ❶
} catch (Exception e) {
                e.printStackTrace(); ❷
}
```

❶ Here, we are using Future.get() to wait until the reply from Kafka arrives back. The specific Future implemented by the Producer will throw an exception if Kafka broker sent back an error and our application can handle the problem. If there were no errors, we will get a RecordMetadata object which we can use to retrieve the offset the message was written to.

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

KafkaProducer has two types of errors. Retriable errors are those that can be resolved by sending the message again. For example connection error can be resolved because the connection may get re-established, or "no leader" error can be resolved when a new leader is elected for the partition. KafkaProducer can be configured to retry those errors automatically, so the application code will get retriable exceptions only when the number of retries was exhausted and the error was not resolved. Some errors will not be resolved by retrying. For example, "message size too large". In those cases KafkaProducer will not attempt a retry and will return the exception immediately.

Sending Messages 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. 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. In most cases, we really don't need a reply - Kafka sends back the topic, partition and offset of the record after it was written and this information is usually not required by the sending app. 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. Here is an example of how we use a callback:

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

ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA"); ❸
producer.send(record, new DemoProducerCallback()); ❹
```

❶ To use callbacks, you need a class that implements `org.apache.kafka.clients.producer.Callback` inteface, which has a single function - `onCompletion`

❷ If Kafka returned an error, onCompletion will have a non-null exception. Here we "handle" it by printing, but production code will probably have more robust error handling functions.

❸ The records are the same as before

❹ And we pass a Callback object along when sending the record

Configuring Producers

So far we've seen very few configuration parameters for the producers - just the mandatory `bootstrap.servers` URI and serializers.

The producer has a large number of configuration parameters, most are documented in Apache Kafka documentation (http://kafka.apache.org/documentation.html#producerconfigs) and many have reasonable defaults so there is no reason to tinker with every single parameters. Few of the parameters have significant impact on memory use, performance and reliability of the producers. We will review those here.

ACKS

The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful. This option has significant impact on how likely it is that messages will be lost. The options are:

- If `acks = 0` the Producer will not wait for any reply from the broker before assuming the message was sent successfully. This means that if something went wrong and the broker did not receive the message, the producer will not know about this and the message will be lost. However, because the producer is not waiting for any response from the server, it can send messages as fast as the network will support, so this setting can be used to achieve very high throughput.
- If `acks = 1` the producer will receive a success response from the broker the moment the leader replica received the message. If the message can't be written to the leader (for example, if the leader crashed and a new leader was not elected yet), the Producer will receive an error response and can retry sending the message, avoiding potential loss of data. The message can still get lost if the

leader crashes and a replica without this message gets elected as the new leader (via unclean leader election). In this case throughput depends on whether we send messages synchronously or asynchronously. If our client code waits for reply from the server (by calling `get()` method of the Future object returned when sending a message) it will obviously increase latency significantly (at least by a network round-trip). If the client uses callbacks, latency will be hidden, but throughput will be limited by the number of in-flight messages (i.e. how many messages the producer will send before receiving replies from the server).

- If `acks = all` the Producer will receive a success response from the broker once all in-sync replicas received the message. This is the safest mode since you can make sure more than one broker has the message and that it will survive even in case of crash (More information on this in Chapter 5). However, the latency we discussed in the `acks = 1` case will be even higher, since we will be waiting for more than just one broker to receive the message.

BUFFER.MEMORY

This sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers. If messages are sent by the application faster than they can be delivered to the server, this may cause the producer to run out of space and addition `send()` calls will either block or throw an exception, based on `block.on.buffer.full` parameter (replaced with `max.block.ms` in release 0.9.0.0 which allows blocking for a certain time and then throwing an exception).

COMPRESSION.TYPE

By default, messages are sent uncompressed. This parameter can be set to `snappy`, `gzip` or `lz4` in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers. Snappy compression was invented by Google to provide decent compression ratio with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. Gzip compression will typically use more CPU and time but result in better compression ratios, so it recommended in cases where network bandwidth is more restricted. By enabling compression you reduce network utilization and storage, which is often a bottleneck when sending messages to Kafka.

RETRIES

When the producer receives an error message from the server, the error could be transient (for example, lack of leader for a partition). In this case, the value of `retries` parameter will control how many times the producer will retry sending the message before giving up and notifying the client of an issue. By default the producer will wait 100ms between retries, but you can control this using `retry.backoff.ms` parameter. We recommend testing how long it takes to recover from a crashed broker (i.e how long until all partitions get new leaders), and setting the number of retries and delay between them such that the total amount of time spent retrying will be longer than the time it takes the Kafka cluster to recover from the crash - otherwise the producer will give up too soon. Not all errors will be retried by the producer. Some errors are not transient and will not cause retries (for example “message too large” error). In general, because the producer handles retries for you, there is no point in handling retries within your own application logic. You will want to focus your efforts on handling non-retriable errors or cases where retry attempts were exhausted.

BATCH.SIZE

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes (not messages!) that will be used for each batch. When the batch is full, all the messages in the batch will be sent. However, this does not mean that the producer will wait for the batch to become full. The producer will send half-full batches and even batches with just a single message in them. Therefore setting the batch size too large will not cause delays in sending messages, it will just use more memory for the batches. Setting the batch size too small, will add some overhead since the producer will need to send messages more frequently.

LINGER.MS

`linger.ms` control the amount of time we wait for additional messages before sending the current batch. KafkaProducer sends a batch of messages either when the current batch is full or when `linger.ms` limit is reached. By default, the producer will send messages as soon as there is the sender thread is available to send them, even if there’s just one message in the batch. By setting `linger.ms` higher than 0, we instruct the producer to wait few milliseconds for additional messages to get added to the batch before sending it to the brokers. This increases latency, but also increases throughput (since we send more messages at once there is less overhead per message).

CLIENT.ID

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging, metrics and for quotas.

MAX.IN.FLIGHT.REQUESTS.PER.CONNECTION

This controls how many messages the producer will send to the server without receiving responses. Setting this high can increase memory usage while improving throughput, although setting it too high can reduce throughput as batching becomes less efficient. Setting this to 1 will guarantee that messages will be written to the broker in the order they were sent, even when retries occure.

TIMEOUT.MS, REQUEST.TIMEOUT.MS AND METADATA.FETCH.TIMEOUT.MS

These parameters control how long the producer will wait for reply from the server when sending data (`request.timeout.ms`) and when requesting metadata such as who are the current leaders for the partitions we are writing to (`metadata.fetch.timeout.ms`). If the timeout is reached without reply, the producer will either retry sending or respond with an error (either through exception or the send callback). `timeout.ms` controls the time the broker will wait for in-sync replicas to acknowledge the message in order to meet the `acks` configuration - the broker will return an error is the time elapsed without the necessary acknowledgements.

MAX.BLOCK.MS

This parameter controls how long the producer will block when calling `send()` and 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 sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For example, with a default maximum request size of 1MB, the largest message you can send is 1MB or the producer can batch 1000 messages of size 1K each into one request. In addition, the broker has its own limit on the size of the largest message it will accept (`message.max.bytes`), it is usually a good idea to have both these configurations match, so the producer will not attempt to send messages of a size that will be rejected by the broker.

RECEIVE.BUFFER.BYTES AND SEND.BUFFER.BYTES

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the operating system defaults will be used. It can be a good idea to increase those when producers or consumers communicate with brokers in a different data center - since those network links typically have higher latency and lower bandwidth.

ORDERING GUARANTEES

Apache Kafka preserves order of messages within a partition. This means that if messages were sent from the producer in a specific order, the broker will write them to a partition in this order and all consumers will read them in this order. For some use-cases, order is very important. There is a big difference between depositing 100$ in an account and later withdrawing them, and the other way around! However, some use cases are less sensitive.

Setting the `retries` parameter to non-zero and the `max.in.flights.requests.per.session` to more than one, mean that it is possible that the broker will fail to write the first batch of messages, succeed to write the second (which was already in flight) and then retry the first batch and succeed, thereby reversing the order.

If the order is critical, usually success is critical too so setting retries to zero is not an option, however you can set `in.flights.requests.per.session = 1` to make sure that no additional messages will be sent to the broker while the first batch is still retrying. This will severly limit the throughput of the producer, so only use this when order is important.

Serializers

As seen in previous examples, Producer configuration includes mandatory serializers. We've seen how to use the default String serializer. Kafka also includes Serializers for Integers and ByteArrays, but this does not cover most use-cases. Eventually you will want to be able to serialize more generic records.

We will start by showing how to write your own serializer, and then introduce the Avro serializer as a recommended alternative.

Custom Serializers

When the object you need to send to Kafka is not a simple String or Integer, you have a choice of either using a generic serialization library like Avro, Thrift or Protobuf to create records, or to create a custom serializer for objects you are already using. We highly recommend to use generic seriazation library. But in order to understand how the serializers work and why it is a good idea to use a serialization library, lets see what it takes to write your own custom serializer.

For example, suppose that instead of recording just the customer name, you created a simple class to represent customers:

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

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

Now suppose we want to create a custom serializer for this class. It will look something like this:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

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

  @Override
  /**
  We are serializing Customer as:
  4 byte int representing customerId
  4 byte int representing length of customerName in UTF-8 bytes (0 if name is Null)
  N bytes representing customerName in UTF-8
  */
  public byte[] serialize(String topic, Customer data) {
        try {
                  byte[] serializedName;
                  int stringSize;
      if (data == null)
        return null;
      else {
                              if (data.getName() != null) {
                serializeName = data.getName().getBytes("UTF-8");
                stringSize = serializedName.length;
                              } else {
                                      serializedName = new byte[0];
```

```
                                        stringSize = 0;
                                }
                        }

            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 Customer to byte[] " + e);
          }
        }

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

Configuring a Producer with this CustomerSerializer will allow you to define `ProducerRecord<String, Customer>` and send Customer data directly to the Producer. This example is pretty simple, but you can see how fragile the code is - If we ever have too many customers for example and need to change customerID to Long, or if we ever decide to add startDate field to Customer, we will have a serious issue in maintaining compatibility between old and new messages. Debugging compatibility issues between different versions of Serializers and Deserializers is fairly challenging - you need to compare arrays of raw bytes. To make matters even worse, if multiple teams in the same company end up writing Customer data to Kafka, they will all need to use the same Serializers and modify the code at the exact same time.

For these reasons, we recommend to never implement your own custom serializer, instead use an existing serializers and deserializers such as JSON, Apache Avro, Thrift or Protobuf. In the following section we will describe Apache Avro and then show how to serialize Avro records and send them to Kafka.

Serializing using Apache Avro

Apache Avro is a language neutral data serialization format. The project was created by Doug Cutting to provide a way to share data files with a large audience.

Avro data is described in a language independent `schema`. The schema is usually described in JSON and the serialization is usually to binary files although serializing to JSON is also supported. Avro assumes that the schema is present when reading and writing files, usually by embedding the schema in the files themselves.

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka is that when the application writing messages switches to a new schema, the applications reading the data can continue processing messages without requiring any change or update.

Suppose the original schema was:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string""},
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"} ❶
 ]
}
```

❶ id and name fields are mandatory, while fax number is optional and defaults to null

We used this schema for few month and generated few terabytes of data in this format. Now suppose that we decide that in the new version, we upgraded to the 21st century and we will no longer include a “faxNumer” field, instead we have “email” field.

The new schema will be:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

Now after upgrading to the new version, old records will contain "faxNumber" and new records will contain "email". Some of the applications reading the data were upgraded and how will this be handled?

The reading application will contain calls to methods similar to `getName()`, `getId()` and `getFaxNumber`. If it encounters a message written with the new schema, `getName()` and `getId()` will continue working with no modification. `getFaxNumber()` will return `null` since the message will not contain a fax number.

Now suppose we upgraded our reading application and it no longer has `getFaxNumber()` method but rather `getEmail()`. If it encounters a message written with the old schema, `getEmail()` will return `null` since the older messages do not contain an email address.

You can see here the main benefit of using Avro: Even though we changed the schema in the messages without changing all the applications reading the data, there will be no exceptions or breaking errors and no need for expensive updates of existing data.

There are two caveats to this ideal scenario:

- The schema used for writing the data and the schema expected by the reading application must be compatible. Avro documentation includes the compatibility rules.
- The deserializer will need access to the schema that was used when writing the data, even when it is different than the schema expected by the application that accesses the data. In Avro files the writing schema is included in the file itself, but there is a better way to handle this for Kafka messages. We will look at that next.

Using Avro records with Kafka

Unlike Avro files, where storing the entire schema in the data file is a fairly reasonable overhead, storing the entire schema in each record will usually more than double the record size. However, Avro still requires the entire schema to be present when reading the record, so we need to locate the schema elsewhere. To achieve this, we follow a common architecture pattern and use a `Schema Registry`. The Schema Registry is not part of Apache Kafka and there are several open source implementations of the schema registry concepts. We'll use the Confluent Schema Registry for this example. You can find the Schema Registry code in Github, or you can install it as part of the Confluent Platform (http://docs.confluent.io/current/installation.html). If you decide to use the Schema Registry, then we recommend checking the documentation (http://docs.confluent.io/current/schema-registry/docs/index.html).

The idea is to store all the schemas used to write data to Kafka in the registry. Then we simply store the identifier for the schema in the record we produce to Kafka. The readers can then use the identifier to pull the record out of the schema registry and deserialize the data. The key is that all this work - storing the schema in the registry and pulling it up when required is done in the serializers and deserializers. The code that produces data to Kafka simply uses the Avro serializers just like it would any other serializer.

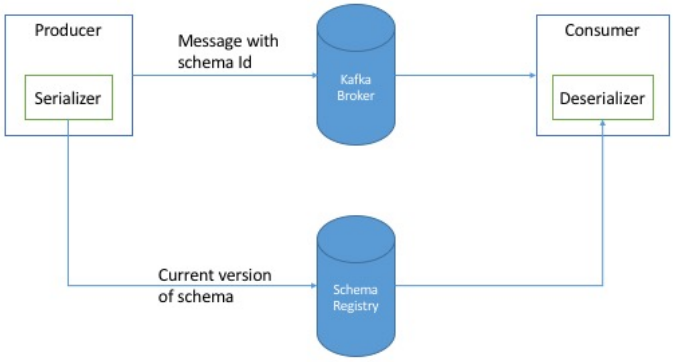

Figure 3-2. Flow diagram of serialization and deserializetion of Avro records

Here is an example of how to produce generated Avro objects to Kafka (See Avro Documentation (http://avro.apache.org/docs/current/) on how to use code generation with Avro):

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";
int wait = 500;

Producer<String, Customer> producer = new KafkaProducer<String, Customer>(props); ❸

// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer " + customer.toString());
    ProducerRecord<String, Customer> record =
                        new ProducerRecord<>(topic, customer.getId(), customer); ❹
    producer.send(record); ❺
}
```

❶ We use the KafkaAvroSerializer to serialize our objects with Avro. Note that the AvroSerializer can also handle primitives, which is why we can later use String as the record key and our Customer object as the value.

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas.

❸ `Customer` is our generated object. We tell the producer that our records will contain Customer as the value

❹ We also instantiate `ProducerRecord` with Customer as the value type, and pass a Customer object when creating the new record.

❺ That is it. We send the record with our Customer object and KafkaAvroSerializer will handle the rest.

What if you prefer to use generic Avro objects rather than the generated Avro objects? No worries. In this case you just need to provide the schema:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

    String schemaString = "{\"namespace\": \"customerManagement.avro\",
                                                                                                \"type
                          "\"name\": \"Customer\"," +
                          "\"fields\": [" +
                           "{\"name\": \"id\", \"type\": \"int\"}," +
                           "{\"name\": \"name\", \"type\": \"string\"}," +
                           "{\"name\": \"email\", \"type\": [\"null\",\"string\"], \"default\":\"null\" }" +
                          "]}";
    Producer<String, GenericRecord> producer = new KafkaProducer<String, GenericRecord>(props); ❹

    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse(schemaString);

    for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
      String name = "exampleCustomer" + nCustomers;
      String email = "example " + nCustomers + "@example.com"

      GenericRecord customer = new GenericData.Record(schema); ❺
      customer.put("id", nCustomer);
      customer.put("name", name);
      customer.put("email", email);

      ProducerRecord<String, GenericRecord> data =
                              new ProducerRecord<String, GenericRecord>("customerContacts", name, customer); ❻
      producer.send(data);
    }
  }
```

❶ We still use the same KafkaAvroSerializer

❷ And provide URI of the same Schema Registry

❸ But now we also need to provide the Avro schema, since it is not provided by the Avro generated object

❹ Our object type is an Avro GenericRecord, which we initialize with our schema and the data we want to write.

❺ Then the value of the ProducerRecord is simply a GenericRecord which countains our schema and data. The serializer will know how to get the schema from this record, store it in the schema registry and serialize the object data.

Partitions

In previous examples, the ProducerRecord objects we created included topic name, key and value. Kafka messages are key-value pairs and while it is possible to create a ProducerRecord with just topic and a value, with the key getting set to `null` by default, most applications produce records with keys. Keys serve two goals: They are additional information that gets stored with the message, and they are also used to decide to which one of the topic partitions the message will be written to. All messages with same key will go to the same partition. This means that if a process is reading only a subset of the partitions in a topic (more on that in Chapter 4), 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<>("CustomerCountry", "Laboratory Equipment", "USA");
```

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

```
ProducerRecord<Integer, String> record =
        new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ Here the key will simply be set to `null`, which may indicate that a customer name was missing on a form

When the key is `null` and the default partitioner is used, the record will be sent to one of the available partitions of the topic at random. Round-robin algorithm will be used to balance the messages between the partitions.

If a key exists and the default partitioner is used, Kafka will hash the key (using its own hash algorithm, so hash values will not change when Java is upgraded), and use the result to map the message to a specific partition. This time, it is important that a key will always get mapped to the same partition, so we use all the partitions in the topic to calculate the mapping and not just available partitions. This means that if a specific partition is unavailable when you write data to it, you may get an error. This is a fairly rare occurrence, as you will read in Chapter 6 when we discuss Kafka's replication and availability.

The mapping of keys to partitions is consistent only as long as the number of partitions in a topic does not change. So as long as the number of partitions is constant you can be sure that, for example, records regarding user 045189 will always get written to partition 34. This allows all kinds of optimizations when reading data from partitions. However, the moment you add new partitions to the topic, this is no longer guaranteed - the old records will stay in partition 34 while new records will get written to a different partition. When partitioning of the keys is important, the easiest solution is to create topics with sufficient partitions (Chapter 2 includes suggestions on how to determine a good number of partitions), and never add partitions.

IMPLEMENTING A CUSTOM PARTITIONING STRATEGY

So far we discussed the traits of the default partitioner, which is the one most commonly used. However, Kafka does not limit you to just hash partitions and sometimes there are good reasons to partition data differently. For example, suppose that you are a B2B vendor and your biggest customer is a company manufacturing hand-held devices called Banana. Suppose that Banana is so large that they comprise around 10% of your business. If you use default hash-partitioning, records regarding the Banana account will get allocated to the same partition as other accounts, resulting in one partition being about twice as large as the rest. This can cause servers to run out of space, processing to slow down, etc. What we really want is to give Banana its own partition and then use hash-partitioning to map the rest of the accounts to partitions.

Here is an example of a custom partitioner as described above:

```
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 BananaPartitioner implements Partitioner {

        public void configure(Map<String, ?> configs) {} ❶

        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))) ❷
                throw new InvalidRecordException("We expect all messages to have customer name as key")

        if (((String) key).equals("Banana"))
                return numPartitions; // Banana 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() {}
}
```

❶ Partitioner interface includes `configure`, `partition` and `close` methods. Here we only implement `partition`, although we really should have passed the special customer name through `configure` instead of hard-coding it in `partition`.

❷ We only expect String keys, so we throw an exception if that is not the case

Old Producer APIs

In this chapter we discussed the Java producer client that is part of `org.apache.kafka.clients` package. At the time of writing this chapter, Apache Kafka still has two older clients written in Scala that are part of `kafka.producer` package and part of the core Kafka module. These producers are called SyncProducer (which, depending on the value of `acks` parameter it may wait for the server to ack each message or batch of messages before sending additional messages) and AsyncProducer (Which batches messages in the background, sends them in a separate thread and does not provide feedback regarding success to the client).

Because the current producer supports both behaviors and give much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, please think twice and then refer to Apache Kafka documentation to learn more.

PREV
2. Installing Kafka

NEXT
4. Kafka Consumers - Reading Data from Kafka