Chapter 3. Kafka Producers: Writing Messages to Kafka

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by creating 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 was 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 `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 Chapter 4, we will look at Kafka's consumer client and reading data from Kafka.

THIRD-PARTY CLIENTS

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 languages, 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 the Apache Kafka project, but a list of non-Java clients 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 might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously 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 is critical to never lose a single message or duplicate any messages. Latency should be low, but latencies up to 500 ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

A different use case might be to store click information from a website. In that case, some message loss or a 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 a few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicks 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 use.

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

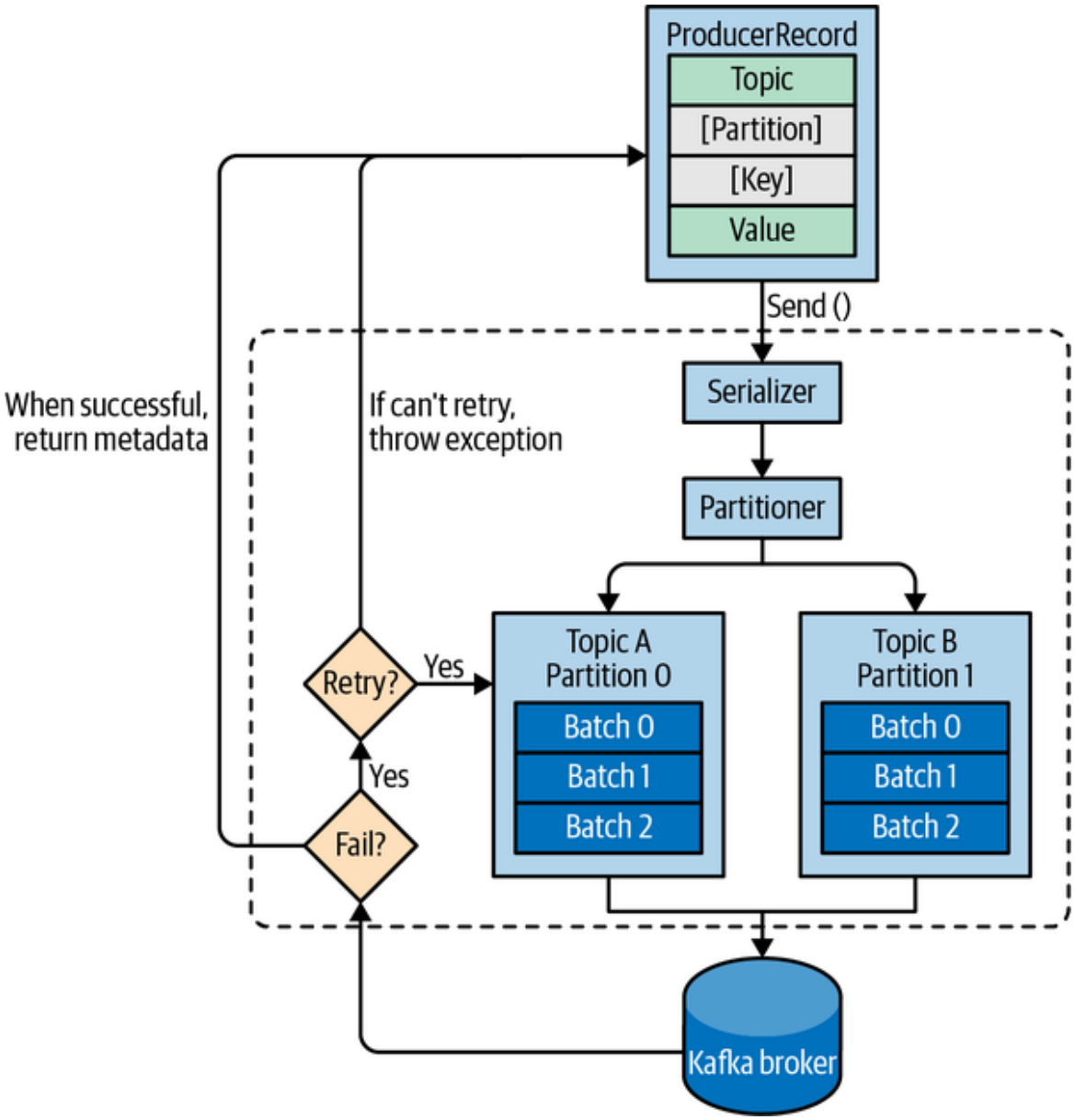

Figure 3-1. High-level overview of Kafka producer components

We start producing messages to Kafka by creating a `ProducerRecord`, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key, a partition, a timestamp, and/or a collection of headers. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to byte arrays so they can be sent over the network.

Next, if we didn't explicitly specify a partition, the data is sent to a partitioner. 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 a 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. A Kafka producer has three mandatory properties:

`bootstrap.servers`

> List of `host:port` pairs of brokers that the producer will use to establish initial connection to the Kafka cluster. This list doesn't need to include all brokers, since the producer will get more information after the initial connection. 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`

Name of a class that will be used to serialize the keys of the records we will produce to Kafka. Kafka brokers expect byte arrays as keys and values of messages. However, the producer interface allows, using parameterized types, any Java object to be sent as a 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 the `org.apache.kafka.common.serialization.Serializer` interface. The producer will use this class to serialize the key object to a byte array. The Kafka client package includes `ByteArraySerializer` (which doesn't do much), `StringSerializer`, `IntegerSerializer`, and much more, 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, but you can use the `Void` type for the key and the `VoidSerializer`.

`value.serializer`

Name of a class that will be used to serialize the values of the records we will produce to Kafka. 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 following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

```
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 plan 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, and we will go over the important ones later in this chapter.

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

Fire-and-forget

We send a message to the server and don't really care if it arrives successfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, in case of nonretriable errors or timeout, messages will get lost and the application will not get any information or exceptions about this.

Synchronous send

Technically, Kafka producer is always asynchronous—we send a message and the `send()` method returns a `Future` object. However, we use `get()` to wait on the `Future` and see if the `send()` was successful or not before sending the next record.

Asynchronous send

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

In the examples that follow, we will see how to send messages using these methods and how to handle the different types of errors that might occur.

While all the examples in this chapter are single threaded, a producer object can be used by multiple threads to send messages.

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 `key serializer` and `value serializer` objects.

❷ We use the producer object `send()` method to send the `ProducerRecord`. As we've seen in the producer architecture diagram in Figure 3-1, 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 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, for example, a `SerializationException` when it fails to serialize the message, a `BufferExhaustedException` or `TimeoutException` if the buffer is full, or an `InterruptException` if the sending thread was interrupted.

Sending a Message Synchronously

Sending a message synchronously is simple but still allows the producer to catch exceptions when Kafka responds to the produce request with an error, or when send retries were exhausted. The main trade-off involved is performance. Depending on how busy the Kafka cluster is, brokers can take anywhere from 2 ms to a few seconds to

respond to produce requests. If you send messages synchronously, the sending thread will spend this time waiting and doing nothing else, not even sending additional messages. This leads to very poor performance, and as a result, synchronous sends are usually not used in production applications (but are very common in code examples).

The simplest way to send a message synchronously is as follows:

```
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 for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a `RecordMetadata` object that we can use to retrieve the offset the message was written to and other metadata.

❷ If there were any errors before or while sending the record to Kafka, we will encounter an exception. In this case, 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, a connection error can be resolved because the connection may get reestablished. A “not leader for partition” error can be resolved when a new leader is elected for the partition and the client metadata is refreshed. `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 a Message Asynchronously

Suppose the network round-trip time between our application and the Kafka cluster is 10 ms. 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, which 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.

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 the `org.apache.kafka. clients.producer.Callback` interface, which has a single function—`onCompletion()`.

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

❸ The records are the same as before.

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

> **WARNING**
>
> The callbacks execute in the producer's main thread. This guarantees that when we send two messages to the same partition one after another, their callbacks will be executed in the same order that we sent them. But it also means that the callback should be reasonably fast to avoid delaying the producer and preventing other messages from being sent. It is not recommended to perform a blocking operation within the callback. Instead, you should use another thread to perform any blocking operation concurrently.

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 that are documented in Apache Kafka documentation, and many have reasonable defaults, so there is no reason to tinker with every single parameter.

However, some of the parameters have a significant impact on memory use, performance, and reliability of the producers. We will review those here.

client.id

`client.id` is a logical identifier for the client and the application it is used in. This can be any string and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics and for quotas. Choosing a good client name will make troubleshooting much easier—it is the difference between "We are seeing a high rate of authentication failures from IP 104.27.155.134" and "Looks like the Order Validation service is failing to authenticate—can you ask Laura to take a look?"

acks

The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful. By default, Kafka will respond that the record was written successfully after the leader received the record (release 3.0 of Apache Kafka is expected to change this default). This option has a significant impact on the durability of written messages, and depending on your use case, the default may not be the best choice. Chapter 7 discusses Kafka's reliability guarantees in depth, but for now let's review the three allowed values for the `acks` parameter:

`acks=0`

> The producer will not wait for a reply from the broker before assuming the message was sent successfully. This means that if something goes wrong and the broker does

not receive the message, the producer will not know about it, 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.

acks=1

The producer will receive a success response from the broker the moment the leader replica receives the message. If the message can't be written to the leader (e.g., 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 the latest messages were not yet replicated to the new leader.

acks=all

The producer will receive a success response from the broker once all in sync replicas receive the message. This is the safest mode since you can make sure more than one broker has the message and that the message will survive even in case of a crash (more information on this in Chapter 6). 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.

TIP

You will see that with lower and less reliable `acks` configuration, the producer will be able to send records faster. This means that you trade off reliability for *producer latency*. However, *end-to-end latency* is measured from the time a record was produced until it is available for consumers to read and is identical for all three options. The reason is that, in order to maintain consistency, Kafka will not allow consumers to read records until they are written to all in sync replicas. Therefore, if you care about end-to-end latency, rather than just the producer latency, there is no trade-off to make: you will get the same end-to-end latency if you choose the most reliable option.

Message Delivery Time

The producer has multiple configuration parameters that interact to control one of the behaviors that are of most interest to developers: how long will it take until a call to `send()` will succeed or fail. This is the time we are willing to spend until Kafka responds successfully, or until we are willing to give up and admit defeat.

The configurations and their behaviors were modified several times over the years. We will describe here the latest implementation, introduced in Apache Kafka 2.1.

Since Apache Kafka 2.1, we divide the time spent sending a `ProduceRecord` into two time intervals that are handled separately:

- Time until an async call to `send()` returns. During this interval, the thread that called `send()` will be blocked.
- From the time an async call to `send()` returned successfully until the callback is triggered (with success or failure). This is the same as from the point a `ProduceRecord` was placed in a batch for

sending until Kafka responds with success, nonretriable failure, or we run out of time allocated for sending.

> **NOTE**
>
> If you use `send()` synchronously, the sending thread will block for both time intervals continuously, and you won't be able to tell how much time was spent in each. We'll discuss the common and recommended case, where `send()` is used asynchronously, with a callback.

The flow of data within the producer and how the different configuration parameters affect each other can be summarized in Figure 3-2.[1]

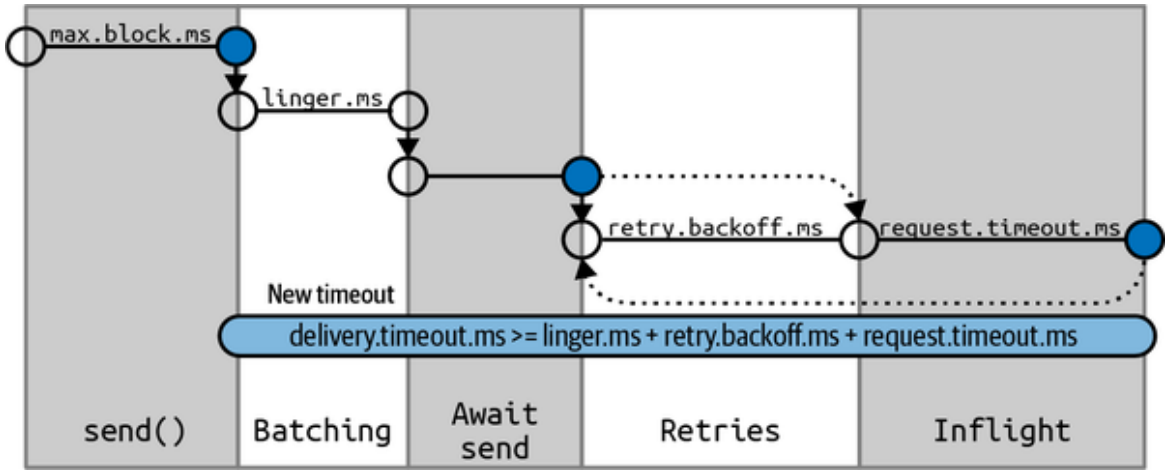

Figure 3-2. Sequence diagram of delivery time breakdown inside Kafka producer

We'll go through the different configuration parameters used to control the time spent waiting in these two intervals and how they interact.

max.block.ms

This parameter controls how long the producer may block when calling `send()` and when explicitly requesting

metadata via `partitionsFor()`. Those methods may 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.

delivery.timeout.ms

This configuration will limit the amount of time spent from the point a record is ready for sending (`send()` returned successfully and the record is placed in a batch) until either the broker responds or the client gives up, including time spent on retries. As you can see in Figure 3-2, this time should be greater than `linger.ms` and `request.timeout.ms`. If you try to create a producer with an inconsistent timeout configuration, you will get an exception. Messages can be successfully sent much faster than `delivery.timeout.ms`, and typically will.

If the producer exceeds `delivery.timeout.ms` while retrying, the callback will be called with the exception that corresponds to the error that the broker returned before retrying. If `delivery.timeout.ms` is exceeded while the record batch was still waiting to be sent, the callback will be called with a timeout exception.

TIP

You can configure the delivery timeout to the maximum time you'll want to wait for a message to be sent, typically a few minutes, and then leave the default number of retries (virtually infinite). With this configuration, the producer will keep retrying for as long as it has time to keep trying (or until it succeeds). This is a much more reasonable way to think about retries. Our normal process for tuning retries is: "In case of a broker crash, it typically takes leader election 30 seconds to complete, so let's keep retrying for 120 seconds just to be on the safe side." Instead of converting this mental dialog to number of retries and time between retries, you just configure `deliver.timeout.ms` to 120.

request.timeout.ms

This parameter controls how long the producer will wait for a reply from the server when sending data. Note that this is the time spent waiting on each producer request before giving up; it does not include retries, time spent before sending, and so on. If the timeout is reached without reply, the producer will either retry sending or complete the callback with a `TimeoutException`.

retries and retry.backoff.ms

When the producer receives an error message from the server, the error could be transient (e.g., a lack of leader for a partition). In this case, the value of the `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 100 ms between retries, but you can control this using the `retry.backoff.ms` parameter.

We recommend against using these parameters in the current version of Kafka. Instead, test how long it takes to recover from a crashed broker (i.e., how long until all partitions get new leaders), and set `delivery.timeout.ms` 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 (e.g., “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 nonretriable errors or cases where retry attempts were exhausted.

> **TIP**
>
> If you want to completely disable retries, setting `retries=0` is the only way to do so.

linger.ms

`linger.ms` controls the amount of time to wait for additional messages before sending the current batch. `KafkaProducer` sends a batch of messages either when the current batch is full or when the `linger.ms` limit is reached. By default, the producer will send messages as soon as there is a sender thread 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 a few milliseconds to add additional messages to the batch before sending it to the brokers. This increases latency a little and significantly increases throughput—the overhead per message is much lower, and compression, if enabled, is much better.

buffer.memory

This config 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, the producer may run out of space, and additional `send()` calls will block for `max.block.ms` and wait for space to free up before throwing an exception. Note that unlike most producer exceptions, this timeout is thrown by `send()` and not by the resulting `Future`.

compression.type

By default, messages are sent uncompressed. This parameter can be set to `snappy`, `gzip`, `lz4`, or `zstd`, 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 ratios 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 results in better compression ratios, so it is 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.

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 because the producer will need to send messages more frequently.

max.in.flight.requests.per.connection

This controls how many message batches the producer will send to the server without receiving responses. Higher settings can increase memory usage while improving throughput. Apache's wiki experiments show that in a single-DC environment, the throughput is maximized with only 2 in-flight requests; however, the default value is 5 and shows similar performance.

ORDERING GUARANTEES

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

Setting the `retries` parameter to nonzero and the `max.in.flight.requests.per.connection` to more than 1 means that it is possible that the broker will fail to write the first batch of messages, succeed in writing the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.

Since we want at least two in-flight requests for performance reasons, and a high number of retries for reliability reasons, the best solution is to set `enable.idempotence=true`. This guarantees message ordering with up to five in-flight requests and also guarantees that retries will not introduce duplicates. Chapter 8 discusses the idempotent producer in depth.

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 1 MB, the largest message you can send is 1 MB, or the producer can batch 1,024 messages of size 1 KB 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 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 OS defaults will be used. It is a good idea to increase these when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

enable.idempotence

Starting in version 0.11, Kafka supports *exactly once* semantics. Exactly once is a fairly large topic, and we'll dedicate an entire chapter to it, but idempotent producer is a simple and highly beneficial part of it.

Suppose you configure your producer to maximize reliability: `acks=all` and a decently large `delivery.timeout.ms` to allow sufficient retries. These make sure each message will be written to Kafka at least once. In some cases, this means that messages will be written to Kafka more than once. For example, imagine that a broker received a record from the producer, wrote it to local disk, and the record was successfully replicated to other brokers, but then the first broker crashed before sending a response to the producer. The producer will wait until it reaches `request.timeout.ms` and then retry. The retry will go to the new leader that already has a copy of this record since the

previous write was replicated successfully. You now have a duplicate record.

To avoid this, you can set `enable.idempotence=true`. When the idempotent producer is enabled, the producer will attach a sequence number to each record it sends. If the broker receives records with the same sequence number, it will reject the second copy and the producer will receive the harmless `DuplicateSequenceException`.

> **NOTE**
>
> Enabling idempotence requires `max.in.flight.requests.per.connection` to be less than or equal to 5, `retries` to be greater than 0, and `acks=all`. If incompatible values are set, a `ConfigException` will be thrown.

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, `ByteArrays`, and many more, 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 creating a custom serialization for

objects you are already using. We highly recommend using a generic serialization library. In order to understand how the serializers work and why it is a good idea to use a serialization library, let's see what it takes to write your own custom serializer.

Suppose that instead of recording just the customer name, you create 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) {
                    serializedName = 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 and pass `Customer` objects 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 a `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 using 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 that is writing messages switches to a new but compatible 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 `faxNumber` is optional and defaults to `null`.

We used this schema for a few months and generated a few terabytes of data in this format. Now suppose we decide that in the new version, we will upgrade to the 21st century and will no longer include a fax number field and will instead use an email field.

The new schema would 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`. In many organizations, upgrades are done slowly and over many months. So we need to consider how pre-upgrade

applications that still use the fax numbers and post-upgrade applications that use email will be able to handle all the events in Kafka.

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, but `getFax Number()` will return `null` because the message will not contain a fax number.

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

This example illustrates the 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.

However, there are two caveats to this scenario:

- The schema used for writing the data and the schema expected by the reading application must be compatible. The Avro documentation includes compatibility rules.
- The deserializer will need access to the schema that was used when writing the data, even when it is different from 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 associated with 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, but there are several open source options to choose from. We'll use the Confluent Schema Registry for this example. You can find the Schema Registry code on GitHub, or you can install it as part of the Confluent Platform. If you decide to use the Schema Registry, we recommend checking the documentation on Confluent.

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 consumers 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 serializer just like it would any other serializer. Figure 3-3 demonstrates this process.

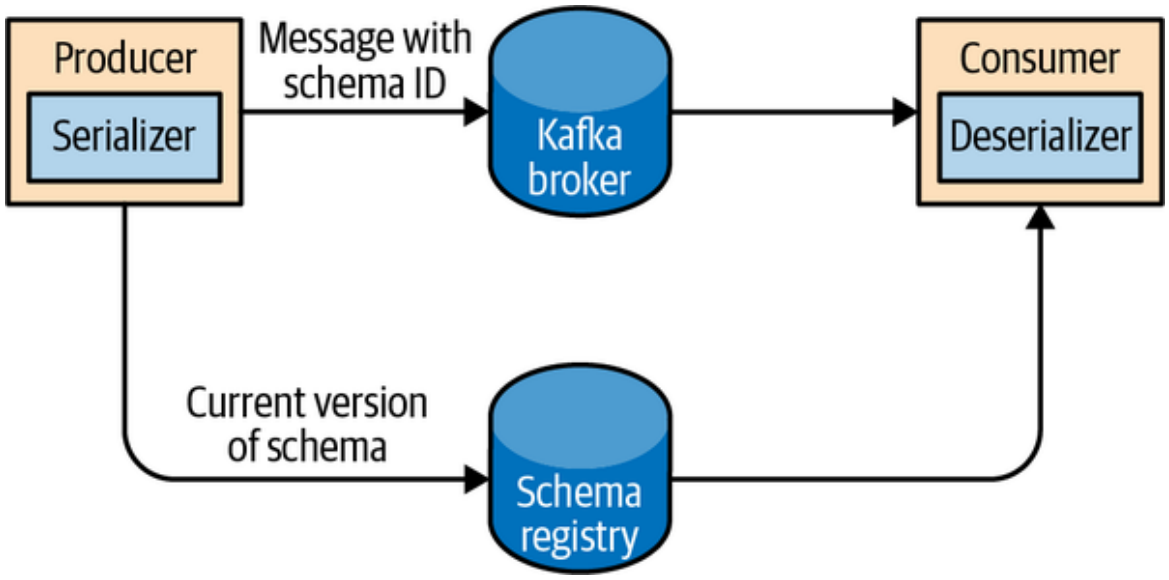

Figure 3-3. Flow diagram of serialization and deserialization of Avro records

Here is an example of how to produce generated Avro objects to Kafka (see the Avro documentation for how to generate objects from Avro schemas):

```
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";

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

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

❶ We use the `KafkaAvroSerializer` to serialize our objects with Avro. Note that the `KafkaAvroSerializer` 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 the configuration of the Avro serializer that will be passed to the serializer by the producer. It 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 `Customer` class is not a regular Java class (plain old Java object, or POJO) but rather a specialized Avro object, generated from a schema using Avro code generation. The Avro serializer can only serialize Avro objects, not POJO. Generating Avro classes can be done either using the *avro-tools.jar* or the Avro Maven plug-in, both part of Apache Avro. See the Apache Avro Getting Started (Java) guide for details on how to generate Avro classes.

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

❻ That's it. We send the record with our `Customer` object, and `KafkaAvroSerializer` will handle the rest.

Avro also allows you to use generic Avro objects, that are used as key-value maps, rather than generated Avro objects with getters and setters that match the schema that was used to generate them. To use generic Avro objects, 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\": \"record\", " + ❸
     "\"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", nCustomers);
    customer.put("name", name);
    customer.put("email", email);

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

❶ We still use the same `KafkaAvroSerializer`.

❷ And we provide the URI of the same Schema Registry.

❸ But now we also need to provide the Avro schema, since it is not provided by an 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` that contains 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 a topic name, key, and value. Kafka messages are key-value pairs, and while it is possible to create a `ProducerRecord` with just a topic and a value, with the key 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 typically also used to decide which one of the topic partitions the message will be written to (keys also play an important role in compacted topics—we'll discuss those in Chapter 6). All messages with the 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<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

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

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

❶

Here, the key will simply be set to `null`.

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. A round-robin algorithm will be used to balance the messages among the partitions. Starting in the Apache Kafka 2.4 producer, the round-robin algorithm used in the default partitioner when handling null keys is sticky. This means that it will fill a batch of messages sent to a single partition before switching to the next partition. This allows sending the same number of messages to Kafka in fewer requests, leading to lower latency and reduced CPU utilization on the broker.

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. Since it is important that a key is always mapped to the same partition, we use all the partitions in the topic to calculate the mapping—not just the available partitions. This means that if a specific partition is unavailable when you write data to it, you might get an error. This is fairly rare, as you will see in Chapter 7 when we discuss Kafka's replication and availability.

In addition to the default partitioner, Apache Kafka clients also provide `RoundRobinPartitioner` and `UniformStickyPartitioner`. These provide random partition assignment and sticky random partition assignment even when messages have keys. These are useful when keys are important for the consuming application (for example, there are ETL applications that use the key from Kafka records as the primary key when loading data from Kafka to a relational database), but the workload may be skewed,

so a single key may have a disproportionately large workload. Using the `UniformStickyPartitioner` will result in an even distribution of workload across all partitions.

When the default partitioner is used, 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 optimization 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 may get written to a different partition. When partitioning keys is important, the easiest solution is to create topics with sufficient partitions (the Confluent blog contains suggestions on how to choose the number of partitions) and never add partitions.

Implementing a custom partitioning strategy

So far, we have 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 that manufactures handheld devices called Bananas. Suppose that you do so much business with customer “Banana” that over 10% of your daily transactions are with this customer. If you use default hash partitioning, the Banana records will get allocated to the same partition as other accounts, resulting in one partition being much larger than 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 all other partitions.

Here is an example of a custom partitioner:

```
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 - 1; // 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 hardcoding it in `partition`.

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

Headers

Records can, in addition to key and value, also include headers. Record headers give you the ability to add some metadata about the Kafka record, without adding any extra information to the key/value pair of the record itself. Headers are often used for lineage to indicate the source of the data in the record, and for routing or tracing messages based on header information without having to parse the message itself (perhaps the message is encrypted and the router doesn't have permissions to access the data).

Headers are implemented as an ordered collection of key/value pairs. The keys are always a `String`, and the values can be any serialized object—just like the message value.

Here is a small example that shows how to add headers to a `ProduceRecord`:

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

record.headers().add("privacy-level","YOLO".getBytes(StandardCharsets.UTF_8));
```

Interceptors

There are times when you want to modify the behavior of your Kafka client application without modifying its code, perhaps because you want to add identical behavior to all applications in the organization. Or perhaps you don't have access to the original code.

Kafka's `ProducerInterceptor` interceptor includes two key methods:

`ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)`

This method will be called before the produced record is sent to Kafka, indeed before it is even serialized. When overriding this method, you can capture information about the sent record and even modify it. Just be sure to return a valid `ProducerRecord` from this method. The record that this method returns will be serialized and sent to Kafka.

`void onAcknowledgement(RecordMetadata metadata, Exception exception)`

This method will be called if and when Kafka responds with an acknowledgment for a send. The method does not allow modifying the response from Kafka, but you can capture information about the response.

Common use cases for producer interceptors include capturing monitoring and tracing information; enhancing the message with standard headers, especially for lineage tracking purposes; and redacting sensitive information.

Here is an example of a very simple producer interceptor. This one simply counts the messages sent and acks received within specific time windows:

```
public class CountingProducerInterceptor implements ProducerInterceptor {

  ScheduledExecutorService executorService =
          Executors.newSingleThreadScheduledExecutor();
  static AtomicLong numSent = new AtomicLong(0);
  static AtomicLong numAcked = new AtomicLong(0);

  public void configure(Map<String, ?> map) {
      Long windowSize = Long.valueOf(
              (String) map.get("counting.interceptor.window.size.ms")); ❶
      executorService.scheduleAtFixedRate(CountingProducerInterceptor::run,
              windowSize, windowSize, TimeUnit.MILLISECONDS);
  }
```

```
    public ProducerRecord onSend(ProducerRecord producerRecord) {
        numSent.incrementAndGet();
        return producerRecord; ❷
    }

    public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
        numAcked.incrementAndGet(); ❸
    }

    public void close() {
        executorService.shutdownNow(); ❹
    }

    public static void run() {
        System.out.println(numSent.getAndSet(0));
        System.out.println(numAcked.getAndSet(0));
    }

}
```

❶ `ProducerInterceptor` is a `Configurable` interface. You can override the `configure` method and setup before any other method is called. This method receives the entire producer configuration, and you can access any configuration parameter. In this case, we added a configuration of our own that we reference here.

❷ When a record is sent, we increment the record count and return the record without modifying it.

❸ When Kafka responds with an ack, we increment the acknowledgment count and don't need to return anything.

❹ This method is called when the producer closes, giving us a chance to clean up the interceptor state. In this case, we close the thread we created. If you opened file handles, connections to remote data stores, or similar, this is the place to close everything and avoid leaks.

As we mentioned earlier, producer interceptors can be applied without any changes to the client code. To use the preceding interceptor with `kafka-console-producer`, an example application that ships with Apache Kafka, follow these three simple steps:

1. Add your jar to the classpath:

   ```
   export
   CLASSPATH=$CLASSPATH:~./target/CountProducerIntercepto
   r-1.0-SNAPSHOT.jar
   ```

2. Create a config file that includes:

   ```
   interceptor.classes=com.shapira.examples.interceptors.
   CountProducerInterceptor
   counting.interceptor.window.size.ms=10000
   ```

3. Run the application as you normally would, but make sure to include the configuration that you created in the previous step:

   ```
   bin/kafka-console-producer.sh --broker-list
   localhost:9092 --topic interceptor-test --
   producer.config producer.config
   ```

Quotas and Throttling

Kafka brokers have the ability to limit the rate at which messages are produced and consumed. This is done via the quota mechanism. Kafka has three quota types: produce, consume, and request. Produce and consume quotas limit the rate at which clients can send and receive data, measured in bytes per second. Request quotas limit the percentage of time the broker spends processing client requests.

Quotas can be applied to all clients by setting default quotas, specific client-ids, specific users, or both. User-specific quotas are only meaningful in clusters where security is configured and clients authenticate.

The default produce and consume quotas that are applied to all clients are part of the Kafka broker configuration file. For example, to limit each producer to send no more than 2 MBps on average, add the following configuration to the broker configuration file: `quota.producer.default=2M`.

While not recommended, you can also configure specific quotas for certain clients that override the default quotas in the broker configuration file. To allow clientA to produce 4 MBps and clientB 10 MBps, you can use the following:
`quota.producer.override="clientA:4M,clientB:10M"`

Quotas that are specified in Kafka's configuration file are static, and you can only modify them by changing the configuration and then restarting all the brokers. Since new clients can arrive at any time, this is very inconvenient. Therefore the usual method of applying quotas to specific clients is through dynamic configuration that can be set using `kafka-config.sh` or the AdminClient API.

Let's look at few examples:

```
bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config
'producer_byte_rate=1024' --entity-name clientC --entity-type clients ❶

bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config
'producer_byte_rate=1024,consumer_byte_rate=2048' --entity-name user1 --
entity-type users ❷

bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config
'consumer_byte_rate=2048' --entity-type users ❸
```

❶

Limiting clientC (identified by client-id) to produce only 1024 bytes per second

❷ Limiting user1 (identified by authenticated principal) to produce only 1024 bytes per second and consume only 2048 bytes per second.

❸ Limiting all users to consume only 2048 bytes per second, except users with more specific override. This is the way to dynamically modify the default quota.

When a client reaches its quota, the broker will start throttling the client's requests to prevent it from exceeding the quota. This means that the broker will delay responses to client requests; in most clients this will automatically reduce the request rate (since the number of in-flight requests is limited) and bring the client traffic down to a level allowed by the quota. To protect the broker from misbehaved clients sending additional requests while being throttled, the broker will also mute the communication channel with the client for the period of time needed to achieve compliance with the quota.

The throttling behavior is exposed to clients via `produce-throttle-time-avg`, `produce-throttle-time-max`, `fetch-throttle-time-avg`, and `fetch-throttle-time-max`, the average and the maximum amount of time a produce request and fetch request was delayed due to throttling. Note that this time can represent throttling due to produce and consume throughput quotas, request time quotas, or both. Other types of client requests can only be throttled due to request time quotas, and those will also be exposed via similar metrics.

WARNING

If you use async `Producer.send()` and continue to send messages at a rate that is higher than the rate the broker can accept (whether due to quotas or just plain old capacity), the messages will first be queued in the client memory. If the rate of sending continues to be higher than the rate of accepting messages, the client will eventually run out of buffer space for storing the excess messages and will block the next `Producer.send()` call. If the timeout delay is insufficient to let the broker catch up to the producer and clear some space in the buffer, eventually `Producer.send()` will throw `TimeoutException`. Alternatively, some of the records that were already placed in batches will wait for longer than `delivery.timeout.ms` and expire, resulting in calling the `send()` callback with a `TimeoutException`. It is therefore important to plan and monitor to make sure that the broker capacity over time will match the rate at which producers are sending data.

Summary

We began this chapter with a simple example of a producer—just 10 lines of code that send events to Kafka. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behavior of the producers. We discussed serializers, which let us control the format of the events we write to Kafka. We looked in-depth at Avro, one of many ways to serialize events but one that is very commonly used with Kafka. We concluded the chapter with a discussion of partitioning in Kafka and an example of an advanced custom partitioning technique.

Now that we know how to write events to Kafka, in Chapter 4 we'll learn all about consuming events from Kafka.

1 Image contributed to the Apache Kafka project by Sumant Tambe under the ASLv2 license terms.

Chapter 4. Kafka Consumers: Reading Data from Kafka

Applications that need to read data from Kafka use a `KafkaConsumer` to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are a few unique concepts and ideas involved. It can be difficult to understand how to use the Consumer API without understanding these concepts first. We'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways Consumer APIs can be used to implement applications with varying requirements.

Kafka Consumer Concepts

To understand how to read data from Kafka, you first need to understand its consumers and consumer groups. The following sections cover those concepts.

Consumers and Consumer Groups

Suppose you have an application that needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store. In this case, your application will create a consumer object, subscribe to the appropriate topic, and start receiving messages, validating them, and writing the results. This

may work well for a while, but what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall further and further behind, unable to keep up with the rate of incoming messages. Obviously there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data among them.

Kafka consumers are typically part of a *consumer group*. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Let’s take topic T1 with four partitions. Now suppose we created a new consumer, C1, which is the only consumer in group G1, and use it to subscribe to topic T1. Consumer C1 will get all messages from all four T1 partitions. See Figure 4-1.

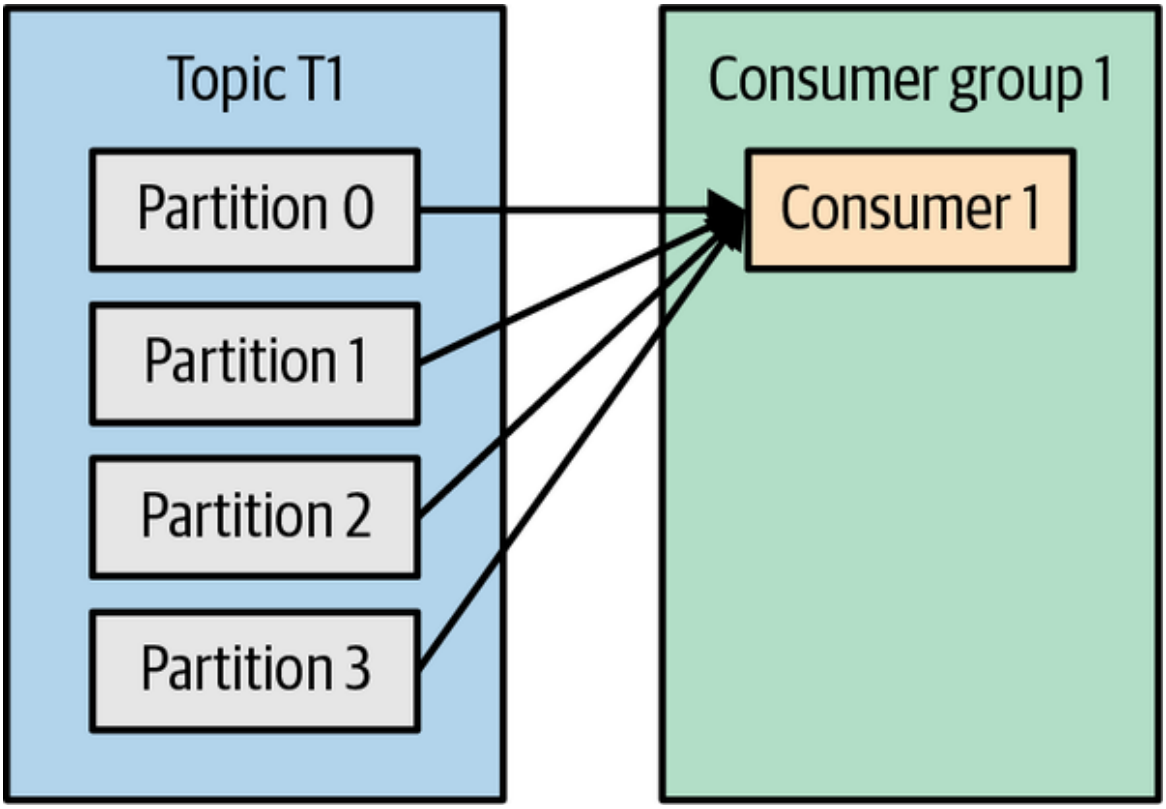

Figure 4-1. One consumer group with four partitions

If we add another consumer, C2, to group G1, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to C1, and messages from partitions 1 and 3 go to consumer C2. See Figure 4-2.

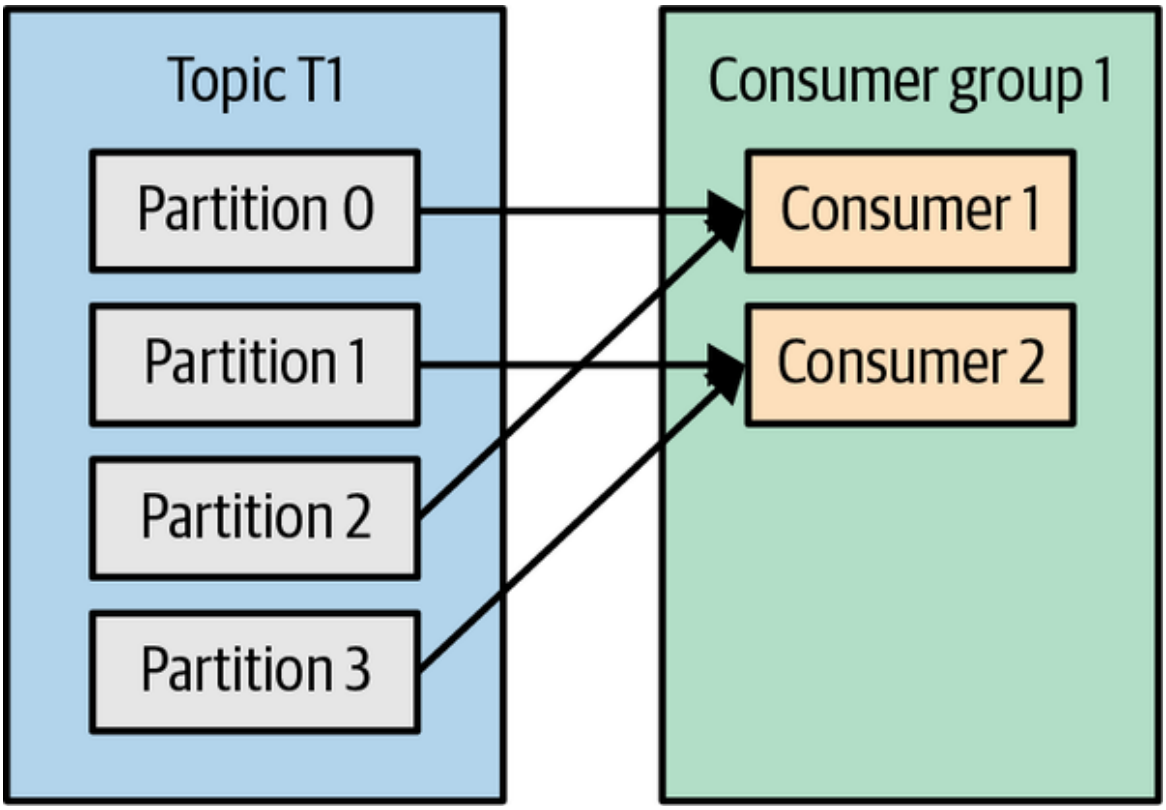

Figure 4-2. Four partitions split to two consumers in a group

If G1 has four consumers, then each will read messages from a single partition. See Figure 4-3.

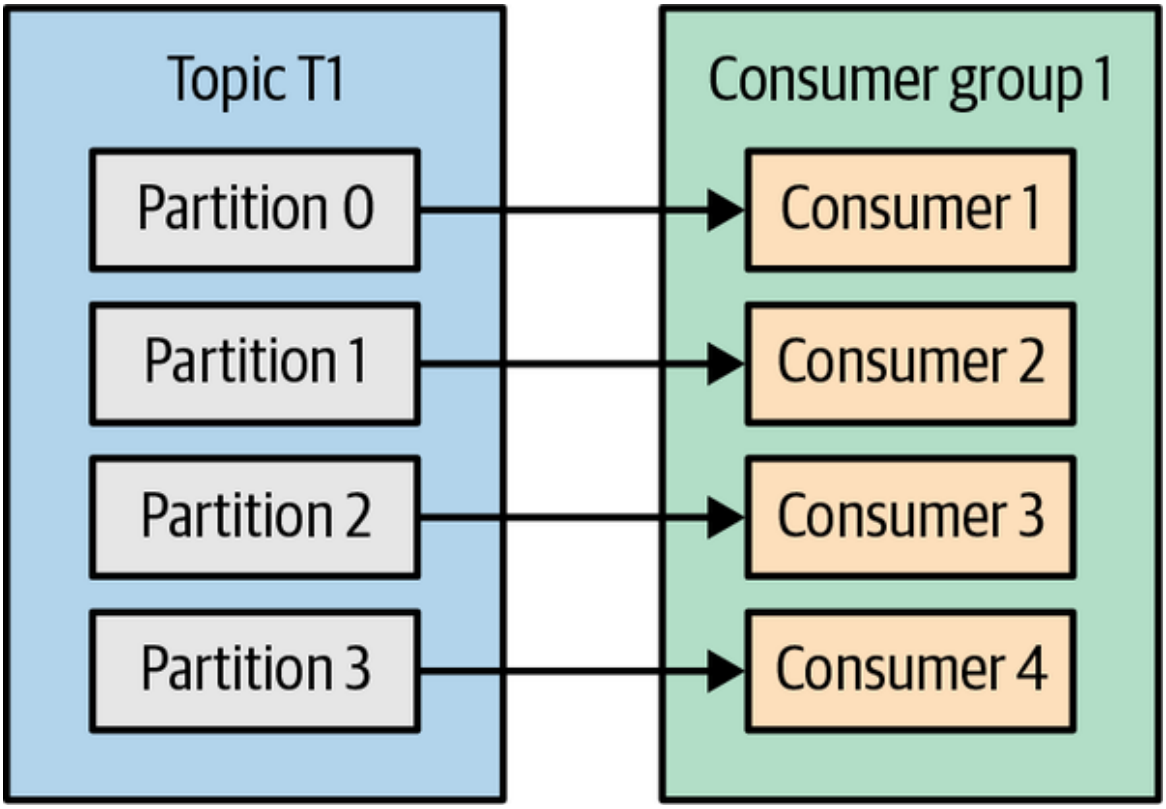

Figure 4-3. Four consumers in a group with one partition each

If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all. See Figure 4-4.

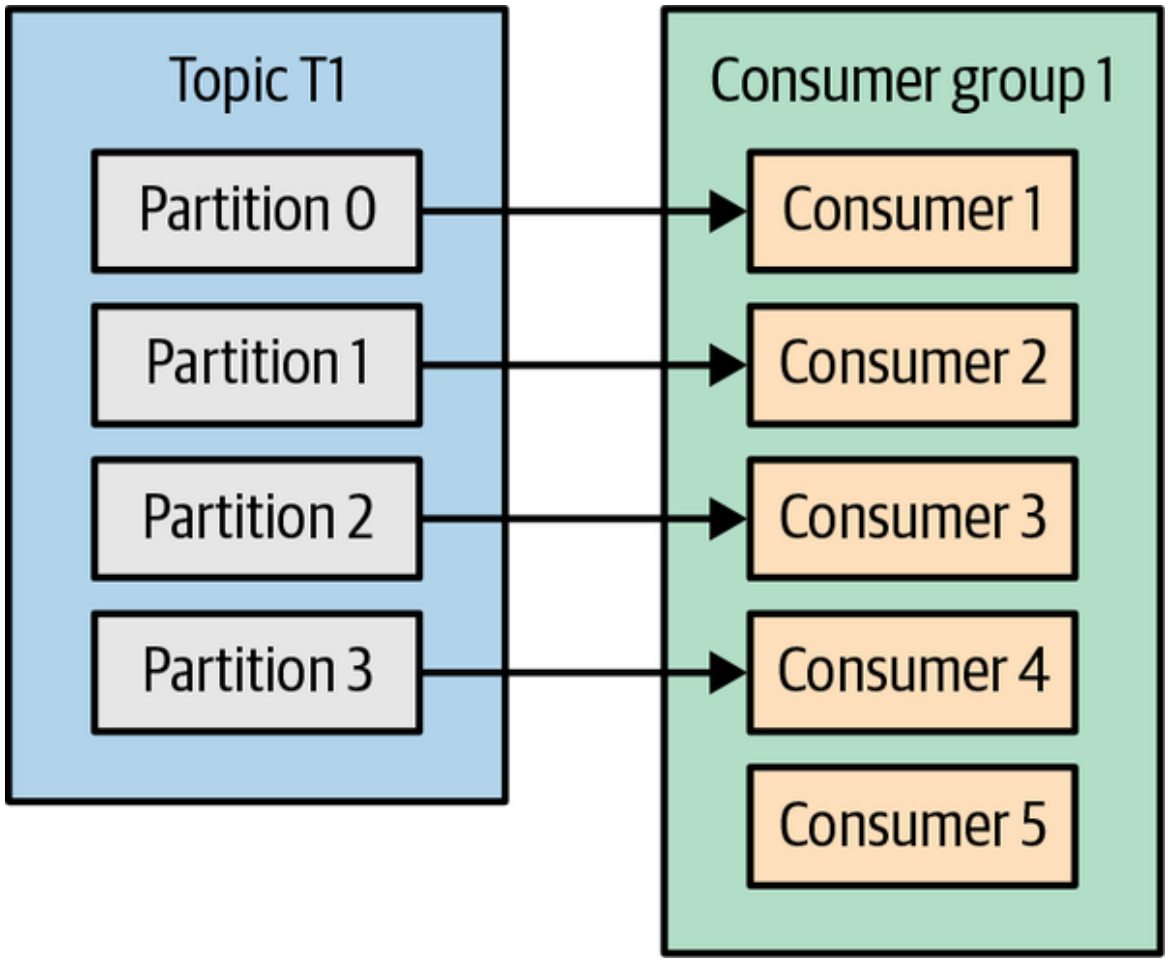

Figure 4-4. More consumers in a group than partitions means idle consumers

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data. In these cases, a single consumer can't possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic—some of the consumers will just be idle. Chapter 2 includes some

suggestions on how to choose the number of partitions in a topic.

In addition to adding consumers in order to scale a single application, it is very common to have multiple applications that need to read data from the same topic. In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics available for many use cases throughout the organization. In those cases, we want each application to get all of the messages, rather than just a subset. To make sure an application gets all the messages in a topic, ensure the application has its own consumer group. Unlike many traditional messaging systems, Kafka scales to a large number of consumers and consumer groups without reducing performance.

In the previous example, if we add a new consumer group (G2) with a single consumer, this consumer will get all the messages in topic T1 independent of what G1 is doing. G2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for G1, but G2 as a whole will still get all the messages regardless of other consumer groups. See Figure 4-5.

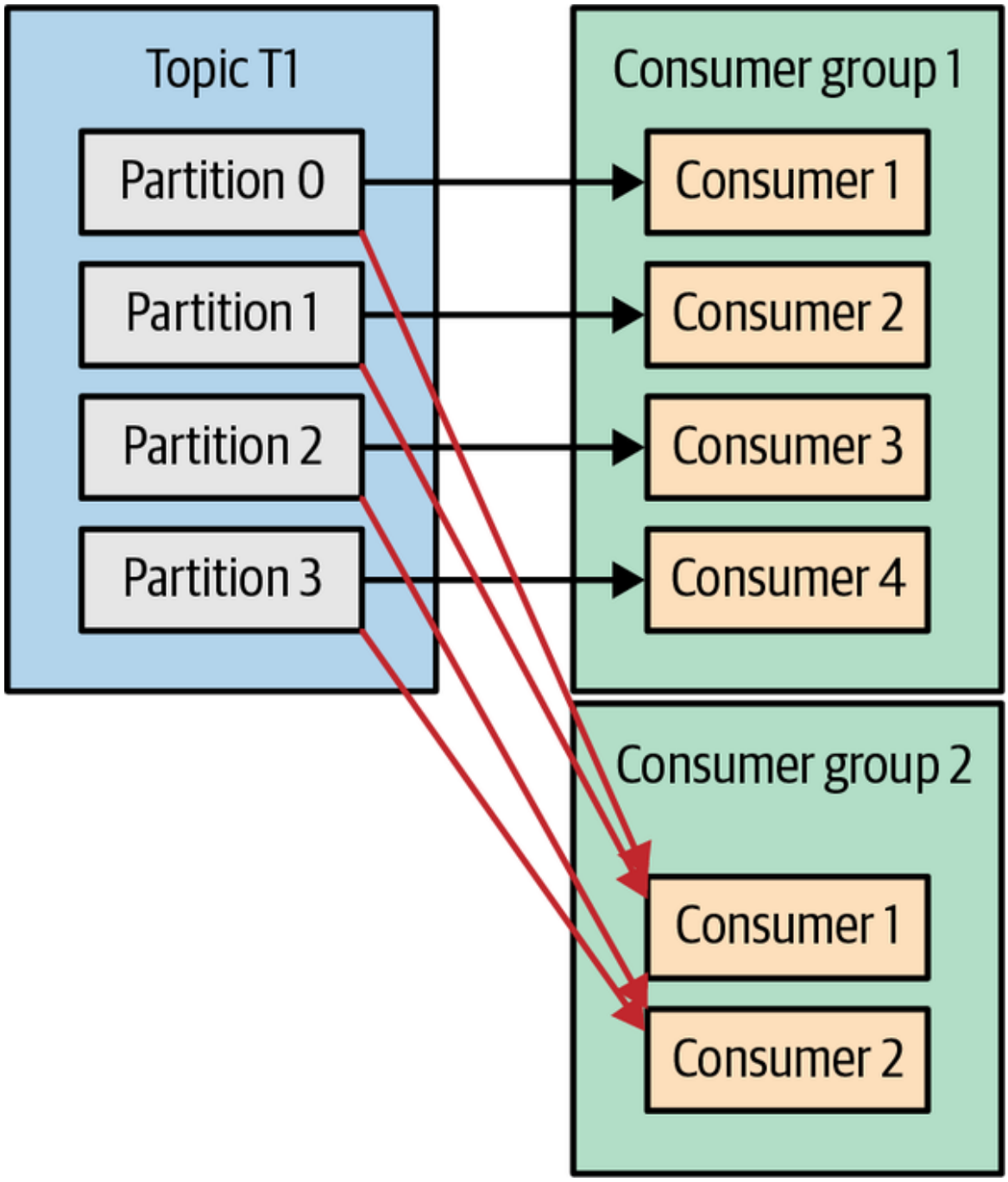

Figure 4-5. Adding a new consumer group, both groups receive all messages

To summarize, you create a new consumer group for each application that needs all the messages from one or more topics. You add consumers to an existing consumer group to scale the reading and processing of messages from the

topics, so each additional consumer in a group will only get a subset of the messages.

Consumer Groups and Partition Rebalance

As we saw in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happens when the topics the consumer group is consuming are modified (e.g., if an administrator adds new partitions).

Moving partition ownership from one consumer to another is called a *rebalance*. Rebalances are important because they provide the consumer group with high availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events they can be fairly undesirable.

There are two types of rebalances, depending on the partition assignment strategy that the consumer group uses:[1]

Eager rebalances

> During an eager rebalance, all consumers stop consuming, give up their ownership of all partitions, rejoin the consumer group, and get a brand-new partition assignment. This is essentially a short window of unavailability of the entire consumer group. The length of the window depends on the size of the consumer group as well as on several configuration

parameters. Figure 4-6 shows how eager rebalances have two distinct phases: first, all consumers give up their partition assigning, and second, after they all complete this and rejoin the group, they get new partition assignments and can resume consuming.

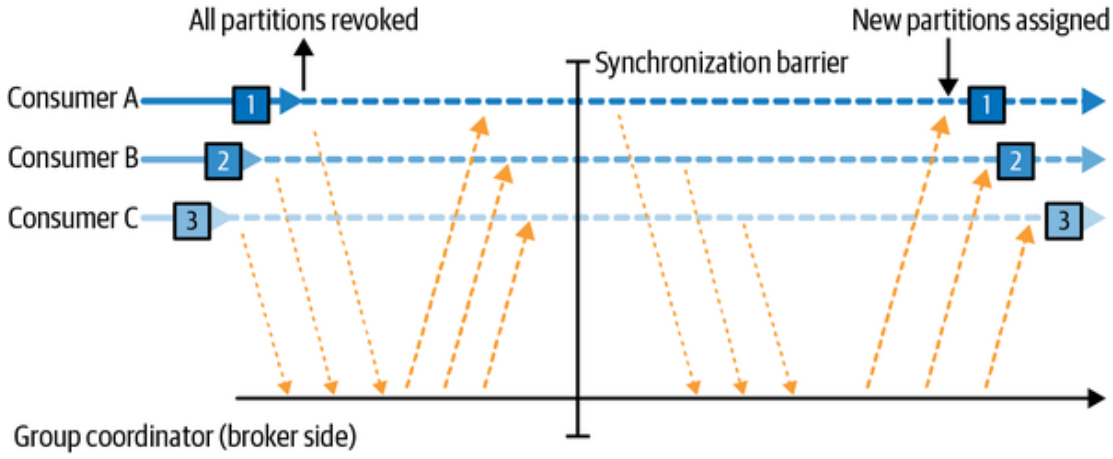

Figure 4-6. Eager rebalance revokes all partitions, pauses consumption, and reassigns them

Cooperative rebalances

Cooperative rebalances (also called *incremental rebalances*) typically involve reassigning only a small subset of the partitions from one consumer to another, and allowing consumers to continue processing records from all the partitions that are not reassigned. This is achieved by rebalancing in two or more phases. Initially, the consumer group leader informs all the consumers that they will lose ownership of a subset of their partitions, then the consumers stop consuming from these partitions and give up their ownership in them. In the second phase, the consumer group leader assigns these now orphaned partitions to their new owners. This incremental approach may take a few iterations until a stable partition assignment is achieved, but it avoids the

complete “stop the world” unavailability that occurs with the eager approach. This is especially important in large consumer groups where rebalances can take a significant amount of time. Figure 4-7 shows how cooperative rebalances are incremental and that only a subset of the consumers and partitions are involved.

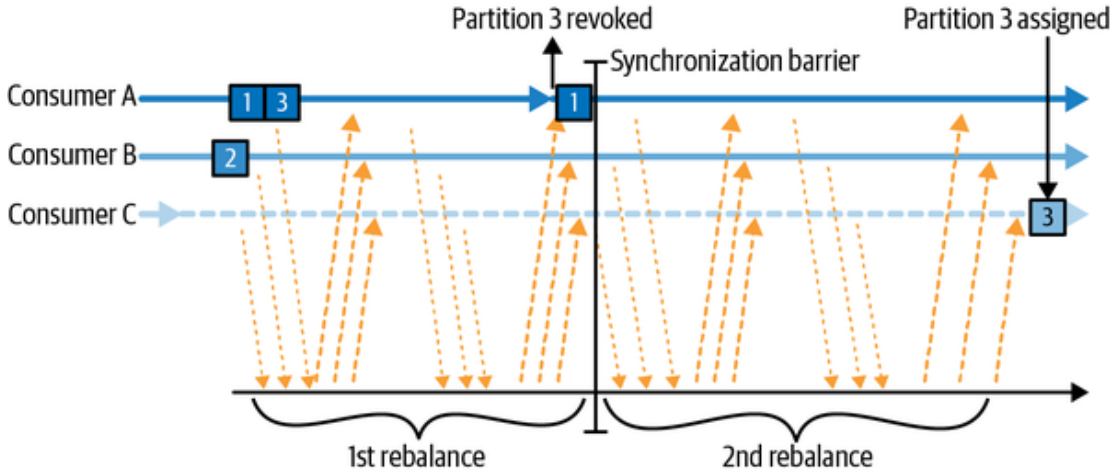

Figure 4-7. Cooperative rebalance only pauses consumption for the subset of partitions that will be reassigned

Consumers maintain membership in a consumer group and ownership of the partitions assigned to them by sending *heartbeats* to a Kafka broker designated as the *group coordinator* (this broker can be different for different consumer groups). The heartbeats are sent by a background thread of the consumer, and as long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive.

If the consumer stops sending heartbeats for long enough, its session will timeout and the group coordinator will consider it dead and trigger a rebalance. If a consumer crashed and stopped processing messages, it will take the group coordinator a few seconds without heartbeats to decide it is dead and trigger the rebalance. During those

seconds, no messages will be processed from the partitions owned by the dead consumer. When closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately, reducing the gap in processing. Later in this chapter, we will discuss configuration options that control heartbeat frequency, session timeouts, and other configuration parameters that can be used to fine-tune the consumer behavior.

HOW DOES THE PROCESS OF ASSIGNING PARTITIONS TO CONSUMERS WORK?

When a consumer wants to join a group, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and that are therefore considered alive) and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of `PartitionAssignor` to decide which partitions should be handled by which consumer.

Kafka has few built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer group leader sends the list of assignments to the `GroupCoordinator`, which sends this information to all the consumers. Each consumer only sees its own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Static Group Membership

By default, the identity of a consumer as a member of its consumer group is transient. When consumers leave a consumer group, the partitions that were assigned to the consumer are revoked, and when it rejoins, it is assigned a new member ID and a new set of partitions through the rebalance protocol.

All this is true unless you configure a consumer with a unique `group.instance.id`, which makes the consumer a *static* member of the group. When a consumer first joins a consumer group as a static member of the group, it is assigned a set of partitions according to the partition assignment strategy the group is using, as normal. However, when this consumer shuts down, it does not automatically leave the group—it remains a member of the group until its session times out. When the consumer rejoins the group, it is recognized with its static identity and is reassigned the same partitions it previously held without triggering a rebalance. The group coordinator that caches the assignment for each member of the group does not need to trigger a rebalance but can just send the cache assignment to the rejoining static member.

If two consumers join the same group with the same `group.instance.id`, the second consumer will get an error saying that a consumer with this ID already exists.

Static group membership is useful when your application maintains local state or cache that is populated by the partitions that are assigned to each consumer. When re-creating this cache is time-consuming, you don’t want this process to happen every time a consumer restarts. On the flip side, it is important to remember that the partitions owned by each consumer will not get reassigned when a consumer is restarted. For a certain duration, no consumer will consume messages from these partitions, and when the consumer finally starts back up, it will lag behind the latest messages in these partitions. You should be confident that the consumer that owns these partitions will be able to catch up with the lag after the restart.

It is important to note that static members of consumer groups do not leave the group proactively when they shut

down, and detecting when they are “really gone” depends on the `session.timeout.ms` configuration. You’ll want to set it high enough to avoid triggering rebalances on a simple application restart but low enough to allow automatic reassignment of their partitions when there is more significant downtime, to avoid large gaps in processing these partitions.

Creating a Kafka Consumer

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer`—you create a Java `Properties` instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start, we just need to use the three mandatory properties: `bootstrap.servers`, `key.deserializer`, and `value.deserializer`.

The first property, `bootstrap.servers`, is the connection string to a Kafka cluster. It is used the exact same way as in `KafkaProducer` (refer to Chapter 3 for details on how this is defined). The other two properties, `key.deserializer` and `value.deserializer`, are similar to the `serializers` defined for the producer, but rather than specifying classes that turn Java objects to byte arrays, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory but very commonly used. The property is `group.id`, and it specifies the consumer group the `KafkaConsumer` instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is uncommon, so

for most of the chapter we will assume the consumer is part of a group.

The following code snippet shows how to create a `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

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

Most of what you see here should be familiar if you've read Chapter 3 on creating producers. We assume that the records we consume will have `String` objects as both the key and the value of the record. The only new property here is `group.id`, which is the name of the consumer group this consumer belongs to.

Subscribing to Topics

Once we create a consumer, the next step is to subscribe to one or more topics. The `subscribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic

names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system or streams processing applications.

For example, to subscribe to all test topics, we can call:

```
consumer.subscribe(Pattern.compile("test.*"));
```

WARNING

If your Kafka cluster has large number of partitions, perhaps 30,000 or more, you should be aware that the filtering of topics for the subscription is done on the client side. This means that when you subscribe to a subset of topics via a regular expression rather than via an explicit list, the consumer will request the list of all topics and their partitions from the broker in regular intervals. The client will then use this list to detect new topics that it should include in its subscription and subscribe to them. When the topic list is large and there are many consumers, the size of the list of topics and partitions is significant, and the regular expression subscription has significant overhead on the broker, client, and network. There are cases where the bandwidth used by the topic metadata is larger than the bandwidth used to send data. This also means that in order to subscribe with a regular expression, the client needs permissions to describe all topics in the cluster—that is, a full `describe` grant on the entire cluster.

The Poll Loop

At the heart of the Consumer API is a simple loop for polling the server for more data. The main body of a consumer will look as follows:

```
Duration timeout = Duration.ofMillis(100);

while (true) { ❶
    ConsumerRecords<String, String> records = consumer.poll(timeout); ❷

    for (ConsumerRecord<String, String> record : records) { ❸
        System.out.printf("topic = %s, partition = %d, offset = %d, " +
                          "customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);

        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()); ❹
    }
}
```

❶ This is indeed an infinite loop. Consumers are usually long-running applications that continuously poll Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.

❷ This is the most important line in the chapter. The same way that sharks must keep moving or they die, 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 to continue consuming. The parameter we pass to `poll()` is a timeout interval and controls how long `poll()` will block if data is not available in the consumer buffer. If this is set to 0 or if there are records available already, `poll()` will return immediately; otherwise, it will wait for the specified number of milliseconds.

❸ `poll()` returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and, of course, the key

and the value of the record. Typically, we want to iterate over the list and process the records individually.

❹ Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each country, so we update a hash table and print the result as JSON. A more realistic example would store the updates result in a data store.

The `poll` loop does a lot more than just get data. The first time you call `poll()` with a new consumer, it is responsible for finding the `GroupCoordinator`, joining the consumer group, and receiving a partition assignment. If a rebalance is triggered, it will be handled inside the poll loop as well, including related callbacks. This means that almost everything that can go wrong with a consumer or in the callbacks used in its listeners is likely to show up as an exception thrown by `poll()`.

Keep in mind that if `poll()` is not invoked for longer than `max.poll.interval.ms`, the consumer will be considered dead and evicted from the consumer group, so avoid doing anything that can block for unpredictable intervals inside the poll loop.

Thread Safety

You can’t have multiple consumers that belong to the same group in one thread, and you can’t have multiple threads safely use the same consumer. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. It is useful to wrap the consumer logic in its own object and then use Java’s `ExecutorService` to start multiple threads,

each with its own consumer. The Confluent blog has a tutorial that shows how to do just that.

> **WARNING**
>
> In older versions of Kafka, the full method signature was `poll(long)`; this signature is now deprecated and the new API is `poll(Duration)`. In addition to the change of argument type, the semantics of how the method blocks subtly changed. The original method, `poll(long)`, will block as long as it takes to get the needed metadata from Kafka, even if this is longer than the timeout duration. The new method, `poll(Duration)`, will adhere to the timeout restrictions and not wait for metadata. If you have existing consumer code that uses `poll(0)` as a method to force Kafka to get the metadata without consuming any records (a rather common hack), you can't just change it to `poll(Duration.ofMillis(0))` and expect the same behavior. You'll need to figure out a new way to achieve your goals. Often the solution is placing the logic in the `rebalanceListener.onPartitionAssignment()` method, which is guaranteed to get called after you have metadata for the assigned partitions but before records start arriving. Another solution was documented by Jesse Anderson in his blog post "Kafka's Got a Brand-New Poll".

Another approach can be to have one consumer populate a queue of events and have multiple worker threads perform work from this queue. You can see an example of this pattern in a blog post from Igor Buzatović.

Configuring Consumers

So far we have focused on learning the Consumer API, but we've only looked at a few of the configuration properties—just the mandatory `bootstrap.servers`, `group.id`, `key.deserializer`, and `value.deserializer`. All of the consumer configuration is documented in the Apache Kafka documentation. Most of the parameters have reasonable defaults and do not require modification, but some have implications on the performance and availability of the

consumers. Let’s take a look at some of the more important properties.

fetch.min.bytes

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records, by default one byte. If a broker receives a request for records from a consumer but the new records amount to fewer bytes than `fetch.min.bytes`, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the consumer and the broker, as they have to handle fewer back-and-forth messages in cases where the topics don’t have much new activity (or for lower-activity hours of the day). You will want to set this parameter higher than the default if the consumer is using too much CPU when there isn’t much data available, or reduce load on the brokers when you have a large number of consumers—although keep in mind that increasing this value can increase latency for low-throughput cases.

fetch.max.wait.ms

By setting `fetch.min.bytes`, you tell Kafka to wait until it has enough data to send before responding to the consumer. `fetch.max.wait.ms` lets you control how long to wait. By default, Kafka will wait up to 500 ms. This results in up to 500 ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set `fetch.max.wait.ms` to a lower value. If you set `fetch.max.wait.ms` to 100 ms and

`fetch.min.bytes` to 1 MB, Kafka will receive a fetch request from the consumer and will respond with data either when it has 1 MB of data to return or after 100 ms, whichever happens first.

fetch.max.bytes

This property lets you specify the maximum bytes that Kafka will return whenever the consumer polls a broker (50 MB by default). It is used to limit the size of memory that the consumer will use to store data that was returned from the server, irrespective of how many partitions or messages were returned. Note that records are sent to the client in batches, and if the first record-batch that the broker has to send exceeds this size, the batch will be sent and the limit will be ignored. This guarantees that the consumer can continue making progress. It’s worth noting that there is a matching broker configuration that allows the Kafka administrator to limit the maximum fetch size as well. The broker configuration can be useful because requests for large amounts of data can result in large reads from disk and long sends over the network, which can cause contention and increase load on the broker.

max.poll.records

This property controls the maximum number of records that a single call to `poll()` will return. Use this to control the amount of data (but not the size of data) your application will need to process in one iteration of the poll loop.

max.partition.fetch.bytes

This property controls the maximum number of bytes the server will return per partition (1 MB by default). When `KafkaConsumer.poll()` returns `ConsumerRecords`, the record object will use at most `max.partition.fetch.bytes` per partition assigned to the consumer. Note that controlling memory usage using this configuration can be quite complex, as you have no control over how many partitions will be included in the broker response. Therefore, we highly recommend using `fetch.max.bytes` instead, unless you have special reasons to try and process similar amounts of data from each partition.

session.timeout.ms and heartbeat.interval.ms

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 10 seconds. If more than `session.timeout.ms` passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`, which controls how frequently the Kafka consumer will send a heartbeat to the group coordinator, whereas `session.timeout.ms` controls how long a consumer can go without sending a heartbeat. Therefore, those two properties are typically modified together—`heartbeat.interval.ms` must be lower than `session.timeout.ms` and is usually set to one-third of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than the default will allow consumer groups to detect and recover from failure sooner but may also cause unwanted rebalances. Setting `session.timeout.ms` higher will reduce

the chance of accidental rebalance but also means it will take longer to detect a real failure.

max.poll.interval.ms

This property lets you set the length of time during which the consumer can go without polling before it is considered dead. As mentioned earlier, heartbeats and session timeouts are the main mechanism by which Kafka detects dead consumers and takes their partitions away. However, we also mentioned that heartbeats are sent by a background thread. There is a possibility that the main thread consuming from Kafka is deadlocked, but the background thread is still sending heartbeats. This means that records from partitions owned by this consumer are not being processed. The easiest way to know whether the consumer is still processing records is to check whether it is asking for more records. However, the intervals between requests for more records are difficult to predict and depend on the amount of available data, the type of processing done by the consumer, and sometimes on the latency of additional services. In applications that need to do time-consuming processing on each record that is returned, `max.poll.records` is used to limit the amount of data returned and therefore limit the duration before the application is available to `poll()` again. Even with `max.poll.records` defined, the interval between calls to `poll()` is difficult to predict, and `max.poll.interval.ms` is used as a fail-safe or backstop. It has to be an interval large enough that it will very rarely be reached by a healthy consumer but low enough to avoid significant impact from a hanging consumer. The default value is 5 minutes. When the timeout is hit, the background thread will send a “leave group” request to let the broker know that the consumer is

dead and the group must rebalance, and then stop sending heartbeats.

default.api.timeout.ms

This is the timeout that will apply to (almost) all API calls made by the consumer when you don't specify an explicit timeout while calling the API. The default is 1 minute, and since it is higher than the request timeout default, it will include a retry when needed. The notable exception to APIs that use this default is the `poll()` method that always requires an explicit timeout.

request.timeout.ms

This is the maximum amount of time the consumer will wait for a response from the broker. If the broker does not respond within this time, the client will assume the broker will not respond at all, close the connection, and attempt to reconnect. This configuration defaults to 30 seconds, and it is recommended not to lower it. It is important to leave the broker with enough time to process the request before giving up—there is little to gain by resending requests to an already overloaded broker, and the act of disconnecting and reconnecting adds even more overhead.

auto.offset.reset

This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset, or if the committed offset it has is invalid (usually because the consumer was down for so long that the record with that offset was already aged out of the broker). The default is "latest," which means that lacking a valid offset, the consumer will start reading from the

newest records (records that were written after the consumer started running). The alternative is “earliest,” which means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning. Setting `auto.offset.reset` to `none` will cause an exception to be thrown when attempting to consume from an invalid offset.

enable.auto.commit

This parameter controls whether the consumer will commit offsets automatically, and defaults to `true`. Set it to `false` if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set `enable.auto.commit` to `true`, then you might also want to control how frequently offsets will be committed using `auto.commit.interval.ms`. We’ll discuss the different options for committing offsets in more depth later in this chapter.

partition.assignment.strategy

We learned that partitions are assigned to consumers in a consumer group. A `PartitionAssignor` is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default, Kafka has the following assignment strategies:

Range

: Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of

partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

RoundRobin

Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1, and partition 1 from topic T2. C2 would have partition 1 from topic T1, and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most one partition difference).

Sticky

The Sticky Assignor has two goals: the first is to have an assignment that is as balanced as possible, and the second is that in case of a rebalance, it will leave as many assignments as possible in place, minimizing the overhead associated with moving partition assignments from one consumer to another. In the common case where all consumers are subscribed to the same topic, the initial assignment from the Sticky Assignor will be as balanced as that of the RoundRobin Assignor. Subsequent assignments will be just as balanced but will reduce the number of partition movements. In cases where consumers in the same group subscribe to different topics, the assignment achieved by Sticky

Assignor is more balanced than that of the RoundRobin Assignor.

Cooperative Sticky

This assignment strategy is identical to that of the Sticky Assignor but supports cooperative rebalances in which consumers can continue consuming from the partitions that are not reassigned. See "Consumer Groups and Partition Rebalance" to read more about cooperative rebalancing, and note that if you are upgrading from a version older than 2.3, you'll need to follow a specific upgrade path in order to enable the cooperative sticky assignment strategy, so pay extra attention to the upgrade guide.

The `partition.assignment.strategy` allows you to choose a partition assignment strategy. The default is `org.apache.kafka.clients.consumer.RangeAssignor`, which implements the Range strategy described earlier. You can replace it with `org.apache.kafka.clients.consumer.RoundRobinAssignor`, `org.apache.kafka.clients.consumer.StickyAssignor`, or `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`. A more advanced option is to implement your own assignment strategy, in which case `partition.assignment.strategy` should point to the name of your class.

client.id

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

client.rack

By default, consumers will fetch messages from the leader replica of each partition. However, when the cluster spans multiple datacenters or multiple cloud availability zones, there are advantages both in performance and in cost to fetching messages from a replica that is located in the same zone as the consumer. To enable fetching from the closest replica, you need to set the `client.rack` configuration and identify the zone in which the client is located. Then you can configure the brokers to replace the default `replica.selector.class` with `org.apache.kafka.common.replica.RackAwareReplicaSelector`.

You can also implement your own `replica.selector.class` with custom logic for choosing the best replica to consume from, based on client metadata and partition metadata.

group.instance.id

This can be any unique string and is used to provide a consumer with static group membership.

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 OS defaults will be used. It can be a good idea to increase these when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

offsets.retention.minutes

This is a broker configuration, but it is important to be aware of it due to its impact on consumer behavior. As long as a consumer group has active members (i.e., members that are actively maintaining membership in the group by sending heartbeats), the last offset committed by the group for each partition will be retained by Kafka, so it can be retrieved in case of reassignment or restart. However, once a group becomes empty, Kafka will only retain its committed offsets to the duration set by this configuration —7 days by default. Once the offsets are deleted, if the group becomes active again it will behave like a brand-new consumer group with no memory of anything it consumed in the past. Note that this behavior changed a few times, so if you use versions older than 2.1.0, check the documentation for your version for the expected behavior.

Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group have not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As discussed before, one of Kafka's unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

We call the action of updating the current position in the partition an `offset commit`. Unlike traditional message queues, Kafka does not commit records individually. Instead, consumers commit the last message they've successfully processed from a partition and implicitly assume that every message before the last was also successfully processed.

How does a consumer commit an offset? It sends a message to Kafka, which updates a special `__consumer_offsets` topic with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

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. See Figure 4-8.

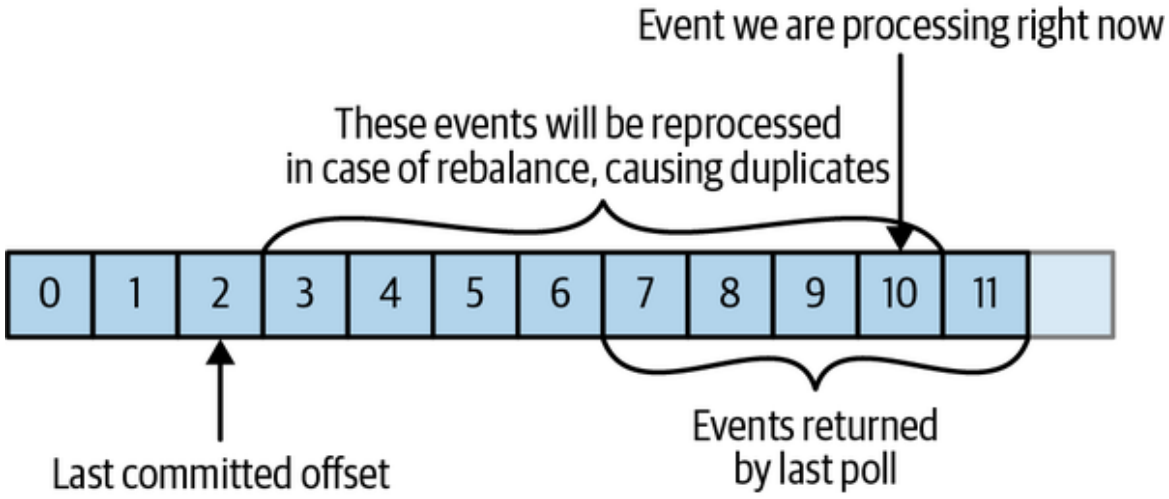

Figure 4-8. Reprocessed messages

If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group. See Figure 4-9.

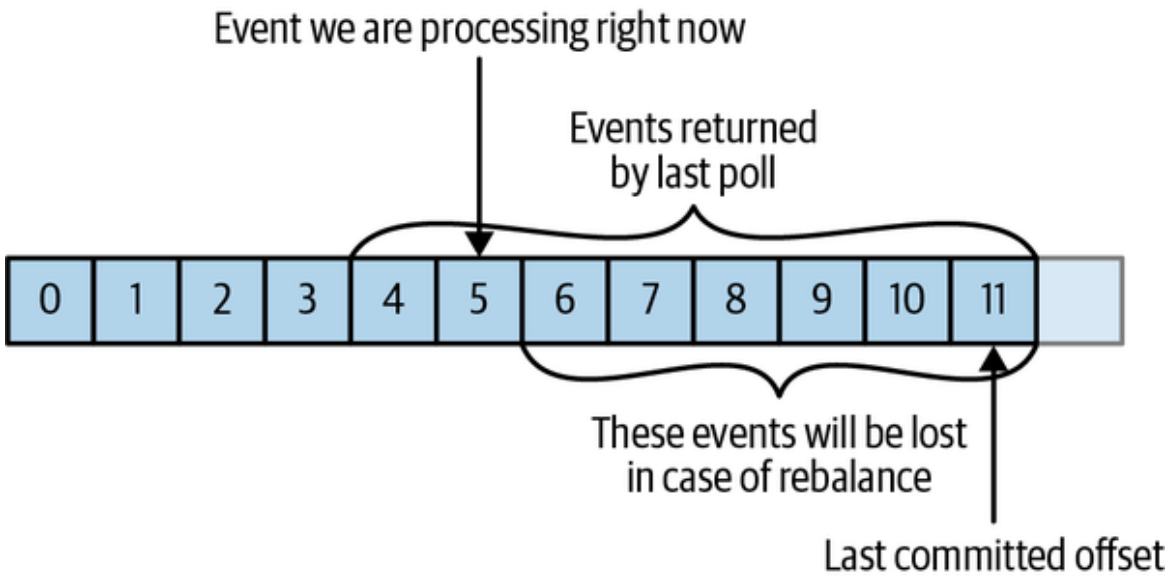

Figure 4-9. Missed messages between offsets

Clearly, managing offsets has a big impact on the client application. The `KafkaConsumer` API provides multiple ways of committing offsets.

> **WHICH OFFSET IS COMMITTED?**
>
> When committing offsets either automatically or without specifying the intended offsets, the default behavior is to commit the offset after the last offset that was returned by `poll()`. This is important to keep in mind when attempting to manually commit specific offsets or seek to commit specific offsets. However, it is also tedious to repeatedly read "Commit the offset that is one larger than the last offset the client received from `poll()`," and 99% of the time it does not matter. So, we are going to write "Commit the last offset" when we refer to the default behavior, and if you need to manually manipulate offsets, please keep this note in mind.

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit=true`, then every five seconds the consumer will commit the latest offset that your client received from `poll()`. The five-second interval is the default and is controlled by setting

`auto.commit.interval.ms`. Just like everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll.

Before using this convenient option, however, it is important to understand the consequences.

Consider that, by default, automatic commits occur every five seconds. Suppose that we are three seconds after the most recent commit our consumer crashed. After the rebalancing, the surviving consumers will start consuming the partitions that were previously owned by the crashed broker. But they will start from the last offset committed. In this case, the offset is three seconds old, so all the events that arrived in those three seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them.

With autocommit enabled, when it is time to commit offsets, the next poll will commit the last offset returned by the previous poll. It doesn't know which events were actually processed, so it is critical to always process all the events returned by `poll()` before calling `poll()` again. (Just like `poll()`, `close()` also commits offsets automatically.) This is usually not an issue, but pay attention when you handle exceptions or exit the poll loop prematurely.

Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commit Current Offset

Most developers exercise more control over the time at which offsets are committed—both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The Consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `enable.auto.commit=false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if the commit fails for some reason.

It is important to remember that `commitSync()` will commit the latest offset returned by `poll()`, so if you call `commitSync()` before you are done processing all the records in the collection, you risk missing the messages that were committed but not processed, in case the application crashes. If the application crashes while it is still processing records in the collection, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice—this may or may not be preferable to missing messages.

Here is how we would use `commitSync` to commit offsets after we finished processing the latest batch of messages:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %d, offset =
            %d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
```

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

❶ Let's assume that by printing the contents of a record, we are done processing it. Your application will likely do a lot more with the records—modify them, enrich them, aggregate them, display them on a dashboard, or notify users of important events. You should determine when you are "done" with a record according to your use case.

❷ Once we are done "processing" all the records in the current batch, we call `commitSync` to commit the last offset in the batch, before polling for additional messages.

❸ `commitSync` retries committing as long as there is no error that can't be recovered. If this happens, there is not much we can do except log an error.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance may create.

Another option is the asynchronous commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

```
Duration timeout = Duration.ofMillis(100);
```

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
            offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a nonretriable failure, `commitAsync()` will not retry. The reason it does not retry is that by the time `commitAsync()` receives a response from the server, there may have been a later commit that was already successful. Imagine that we sent a request to commit offset 2000. There is a temporary communication problem, so the broker never gets the request and therefore never responds. Meanwhile, we processed another batch and successfully committed offset 3000. If `commitAsync()` now retries the previously failed commit, it might succeed in committing offset 2000 *after* offset 3000 was already processed and committed. In the case of a rebalance, this will cause more duplicates.

We mention this complication and the importance of correct order of commits because `commitAsync()` also gives you an option to pass in a callback that will be triggered when the broker responds. It is common to use the callback to log commit errors or to count them in a metric, but if you want to use the callback for retries, you need to be aware of the problem with commit order:

```
Duration timeout = Duration.ofMillis(100);
```

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    }); ❶
}
```

❶ We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged.

RETRYING ASYNC COMMITS

A simple pattern to get the commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit, and add the sequence number at the time of the commit to the `commitAsync` callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable; if it is, there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry because a newer commit was already sent.

Combining Synchronous and Asynchronous Commits

Normally, occasional failures to commit without retrying are not a huge problem because if the problem is temporary, the following commit will be successful. But if we know that this is the last commit before we close the consumer, or before a rebalance, we want to make extra sure that the commit succeeds.

Therefore, a common pattern is to combine `commitAsync()` with `commitSync()` just before shutdown. Here is how it works (we will discuss how to commit just before rebalance when we get to the section about rebalance listeners):

```
Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(),
                record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
    consumer.commitSync(); ❷
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
        consumer.close();
}
```

❶ While everything is fine, we use `commitAsync`. It is faster, and if one commit fails, the next commit will serve as a retry.

❷ But if we are closing, there is no "next commit." We call `commitSync()`, because it will retry until it succeeds or suffers unrecoverable failure.

Committing a Specified Offset

Committing the latest offset only allows you to commit as often as you finish processing batches. But what if you want to commit more frequently than that? What if `poll()` returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those

rows again if a rebalance occurs? You can't just call `commitSync()` or `commitAsync()`—this will commit the last offset returned, which you didn't get to process yet.

Fortunately, the Consumer API allows you to call `commitSync()` and `commitAsync()` and pass a map of partitions and offsets that you wish to commit. If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic "customers" has offset 5000, you can call `commitSync()` to commit offset 5001 for partition 3 in topic "customers." Since your consumer may be consuming more than a single partition, you will need to track offsets on all of them, which adds complexity to your code.

Here is what a commit of specific offsets looks like:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value()); ❷
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0)   ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ This is the map we will use to manually track offsets.

❷ Remember, `println` is a stand-in for whatever processing you do for the records you consume.

❸ After reading each record, we update the offsets map with the offset of the next message we expect to process. The committed offset should always be the offset of the next message that your application will read. This is where we'll start reading next time we start.

❹ Here, we decide to commit current offsets every 1,000 records. In your application, you can commit based on time or perhaps content of the records.

❺ I chose to call `commitAsync()` (without a callback, therefore the second parameter is `null`), but `commitSync()` is also completely valid here. Of course, when committing specific offsets you still need to perform all the error handling we've seen in previous sections.

Rebalance Listeners

As we mentioned in the previous section about committing offsets, a consumer will want to do some cleanup work before exiting and also before partition rebalancing.

If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed. Perhaps you also need to close file handles, database connections, and such.

The Consumer API allows you to run your own code when partitions are added or removed from the consumer. You do this by passing a `ConsumerRebalanceListener` when calling the `subscribe()` method we discussed previously. `ConsumerRebalanceListener` has three methods you can implement:

`public void onPartitionsAssigned(Collection<TopicPartition> partitions)`

Called after partitions have been reassigned to the consumer but before the consumer starts consuming messages. This is where you prepare or load any state that you want to use with the partition, seek to the correct offsets if needed, or similar. Any preparation done here should be guaranteed to return within `max.poll.timeout.ms` so the consumer can successfully join the group.

`public void onPartitionsRevoked(Collection<TopicPartition> partitions)`

Called when the consumer has to give up partitions that it previously owned—either as a result of a rebalance or when the consumer is being closed. In the common case, when an eager rebalancing algorithm is used, this method is invoked before the rebalancing starts and after the consumer stopped consuming messages. If a cooperative rebalancing algorithm is used, this method is invoked at the end of the rebalance, with just the subset of partitions that the consumer has to give up. This is where you want to commit offsets, so whoever gets this partition next will know where to start.

`public void onPartitionsLost(Collection<TopicPartition> partitions)`

Only called when a cooperative rebalancing algorithm is used, and only in exceptional cases where the partitions were assigned to other consumers without first being revoked by the rebalance algorithm (in normal cases, `onPartitionsRevoked()` will be called). This is where you clean up any state or resources that are used with these

partitions. Note that this has to be done carefully—the new owner of the partitions may have already saved its own state, and you'll need to avoid conflicts. Note that if you don't implement this method, `onPartitionsRevoked()` will be called instead.

TIP

If you use a cooperative rebalancing algorithm, note that:

- `onPartitionsAssigned()` will be invoked on every rebalance, as a way of notifying the consumer that a rebalance happened. However, if there are no new partitions assigned to the consumer, it will be called with an empty collection.
- `onPartitionsRevoked()` will be invoked in normal rebalancing conditions, but only if the consumer gave up the ownership of partitions. It will not be called with an empty collection.
- `onPartitionsLost()` will be invoked in exceptional rebalancing conditions, and the partitions in the collection will already have new owners by the time the method is invoked.

If you implemented all three methods, you are guaranteed that during a normal rebalance, `onPartitionsAssigned()` will be called by the new owner of the partitions that are reassigned only after the previous owner completed `onPartitionsRevoked()` and gave up its ownership.

This example will show how to use `onPartitionsRevoked()` to commit offsets before losing ownership of a partition:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>();
Duration timeout = Duration.ofMillis(100);

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Lost partitions in rebalance. " +
```

```
            "Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                 customer = %s, country = %s\n",
                 record.topic(), record.partition(), record.offset(),
                 record.key(), record.value());
             currentOffsets.put(
                 new TopicPartition(record.topic(), record.partition()),
                 new OffsetAndMetadata(record.offset()+1, null));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // ignore, we're closing
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}
```

❶ We start by implementing a `ConsumerRebalanceListener`.

❷ In this example we don’t need to do anything when we get a new partition; we’ll just start consuming messages.

❸ However, when we are about to lose a partition due to rebalancing, we need to commit offsets. We are committing offsets for all partitions, not just the partitions we are about to lose—because the offsets are for events that were already processed, there is no harm

in that. And we are using `commitSync()` to make sure the offsets are committed before the rebalance proceeds.

❹ The most important part: pass the `ConsumerRebalanceListener` to the `subscribe()` method so it will get invoked by the consumer.

Consuming Records with Specific Offsets

So far we've seen how to use `poll()` to start consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence. However, sometimes you want to start reading at a different offset. Kafka offers a variety of methods that cause the next `poll()` to start consuming in a different offset.

If you want to start reading all messages from the beginning of the partition, or you want to skip all the way to the end of the partition and start consuming only new messages, there are APIs specifically for that: `seekToBeginning(Collection<TopicPartition> tp)` and `seekToEnd(Collection<TopicPartition> tp)`.

The Kafka API also lets you seek a specific offset. This ability can be used in a variety of ways; for example, a time-sensitive application could skip ahead a few records when falling behind, or a consumer that writes data to a file could be reset back to a specific point in time in order to recover data if the file was lost.

Here's a quick example of how to set the current offset on all partitions to records that were produced at a specific point in time:

```
Long oneHourEarlier = Instant.now().atZone(ZoneId.systemDefault())
          .minusHours(1).toEpochSecond();
Map<TopicPartition, Long> partitionTimestampMap = consumer.assignment()
        .stream()
        .collect(Collectors.toMap(tp -> tp, tp -> oneHourEarlier)); ❶
Map<TopicPartition, OffsetAndTimestamp> offsetMap
        = consumer.offsetsForTimes(partitionTimestampMap); ❷

for(Map.Entry<TopicPartition,OffsetAndTimestamp> entry: offsetMap.entrySet())
{
    consumer.seek(entry.getKey(), entry.getValue().offset()); ❸
}
```

❶ We create a map from all the partitions assigned to this consumer (via `consumer.assignment()`) to the timestamp we wanted to revert the consumers to.

❷ Then we get the offsets that were current at these timestamps. This method sends a request to the broker where a timestamp index is used to return the relevant offsets.

❸ Finally, we reset the offset on each partition to the offset that was returned in the previous step.

But How Do We Exit?

Earlier in this chapter, when we discussed the poll loop, we told you not to worry about the fact that the consumer polls in an infinite loop, and that we would discuss how to exit the loop cleanly. So, let's discuss how to exit cleanly.

When you decide to shut down the consumer, and you want to exit immediately even though the consumer may be waiting on a long `poll()`, you will need another thread to call `consumer.wakeup()`. If you are running the consumer loop in the main thread, this can be done from `ShutdownHook`. Note that `consumer.wakeup()` is the only consumer method that is safe to call from a different thread. Calling `wakeup` will cause

`poll()` to exit with `WakeupException`, or if `consumer.wakeup()` was called while the thread was not waiting on poll, the exception will be thrown on the next iteration when `poll()` is called. The `WakeupException` doesn't need to be handled, but before exiting the thread, you must call `consumer.close()`. Closing the consumer will commit offsets if needed and will send the group coordinator a message that the consumer is leaving the group. The consumer coordinator will trigger rebalancing immediately, and you won't need to wait for the session to timeout before partitions from the consumer you are closing will be assigned to another consumer in the group.

Here is what the exit code will look like if the consumer is running in the main application thread. This example is a bit truncated, but you can view the full example on GitHub:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...
Duration timeout = Duration.ofMillis(10000); ❷

try {
    // looping until ctrl-c, the shutdown hook will cleanup on exit
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "--  waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value = %s\n",
```

```
                    record.offset(), record.key(), record.value());
            }
            for (TopicPartition tp: consumer.assignment())
                System.out.println("Committing offset at position:" +
                    consumer.position(tp));
            movingAvg.consumer.commitSync();
        }
    } catch (WakeupException e) {
        // ignore for shutdown ❸
    } finally {
        consumer.close(); ❹
        System.out.println("Closed consumer and we are done");
    }
```

❶ `ShutdownHook` runs in a separate thread, so the only safe action you can take is to call `wakeup` to break out of the `poll` loop.

❷ A particularly long poll timeout. If the poll loop is short enough and you don't mind waiting a bit before exiting, you don't need to call `wakeup`—just checking an atomic boolean in each iteration would be enough. Long poll timeouts are useful when consuming low-throughput topics; this way, the client uses less CPU for constantly looping while the broker has no new data to return.

❸ Another thread calling `wakeup` will cause poll to throw a `WakeupException`. You'll want to catch the exception to make sure your application doesn't exit unexpectedly, but there is no need to do anything with it.

❹ Before exiting the consumer, make sure you close it cleanly.

Deserializers

As discussed in the previous chapter, Kafka producers require *serializers* to convert objects into byte arrays that are then sent to Kafka. Similarly, Kafka consumers require

deserializers to convert byte arrays received from Kafka into Java objects. In previous examples, we just assumed that both the key and the value of each message are strings, and we used the default `StringDeserializer` in the consumer configuration.

In Chapter 3 about the Kafka producer, we saw how to serialize custom types and how to use Avro and `AvroSerializers` to generate Avro objects from schema definitions and then serialize them when producing messages to Kafka. We will now look at how to create custom deserializers for your own objects and how to use Avro and its deserializers.

It should be obvious that the serializer used to produce events to Kafka must match the deserializer that will be used when consuming events. Serializing with `IntSerializer` and then deserializing with `StringDeserializer` will not end well. This means that, as a developer, you need to keep track of which serializers were used to write into each topic and make sure each topic only contains data that the deserializers you use can interpret. This is one of the benefits of using Avro and the Schema Registry for serializing and deserializing—the `AvroSerializer` can make sure that all the data written to a specific topic is compatible with the schema of the topic, which means it can be deserialized with the matching deserializer and schema. Any errors in compatibility—on the producer or the consumer side—will be caught easily with an appropriate error message, which means you will not need to try to debug byte arrays for serialization errors.

We will start by quickly showing how to write a custom deserializer, even though this is the less common method,

and then we will move on to an example of how to use Avro to deserialize message keys and values.

Custom Deserializers

Let's take the same custom object we serialized in Chapter 3 and write a deserializer for it:

```
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;
    }
}
```

The custom deserializer will look as follows:

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

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

public class CustomerDeserializer implements Deserializer<Customer> { ❶

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

    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
```

```
        String name;

        try {
            if (data == null)
                return null;
            if (data.length < 8)
                throw new SerializationException("Size of data received " +
                    "by deserializer is shorter than expected");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, "UTF-8");

            return new Customer(id, name); ❷

        } catch (Exception e) {
                throw new SerializationException("Error when deserializing " +
"byte[] to Customer " + e);
        }
    }

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

❶ The consumer also needs the implementation of the `Customer` class, and both the class and the serializer need to match on the producing and consuming applications. In a large organization with many consumers and producers sharing access to the data, this can become challenging.

❷ We are just reversing the logic of the serializer here—we get the customer ID and name out of the byte array and use them to construct the object we need.

The consumer code that uses this deserializer will look similar to this example:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    CustomerDeserializer.class.getName());


KafkaConsumer<String, Customer> consumer =
    new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout);
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("current customer Id: " +
            record.value().getID() + " and
            current customer name: " +  record.value().getName());
    }
    consumer.commitSync();
}
```

Again, it is important to note that implementing a custom serializer and deserializer is not recommended. It tightly couples producers and consumers and is fragile and error prone. A better solution would be to use a standard message format, such as JSON, Thrift, Protobuf, or Avro. We'll now see how to use Avro deserializers with the Kafka consumer. For background on Apache Avro, its schemas, and schema-compatibility capabilities, refer back to Chapter 3.

Using Avro Deserialization with Kafka Consumer

Let's assume we are using the implementation of the `Customer` class in Avro that was shown in Chapter 3. In order to consume those objects from Kafka, you want to implement a consuming application similar to this:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader","true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
            record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ We use `KafkaAvroDeserializer` to deserialize the Avro messages.

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas. This way, the consumer can use the schema that was registered by the producer to deserialize the message.

❸ We specify the generated class, `Customer`, as the type for the record value.

❹ `record.value()` is a `Customer` instance, and we can use it accordingly.

Standalone Consumer: Why and How to Use a Consumer Without a Group

So far, we have discussed consumer groups, which are where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case, there is no reason for groups or rebalances—just assign the consumer-specific topic and/or partitions, consume messages, and commit offsets on occasion (although you still need to configure `group.id` to commit offsets, without calling *subscribe* the consumer won't join any group).

When you know exactly which partitions the consumer should read, you don't *subscribe* to a topic—instead, you *assign* yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group) or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```
Duration timeout = Duration.ofMillis(100);
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
```

```
            partitions.add(new TopicPartition(partition.topic(),
                partition.partition()));
        consumer.assign(partitions); ❷

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

            for (ConsumerRecord<String, String> record: records) {
                System.out.printf("topic = %s, partition = %s, offset = %d,
                    customer = %s, country = %s\n",
                    record.topic(), record.partition(), record.offset(),
                    record.key(), record.value());
            }
            consumer.commitSync();
        }
    }
```

❶ We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.

❷ Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply by bouncing the application whenever partitions are added.

Summary

We started this chapter with an in-depth explanation of Kafka's consumer groups and the way they allow multiple consumers to share the work of reading events from topics. We followed the theoretical discussion with a practical example of a consumer subscribing to a topic and

continuously reading events. We then looked into the most important consumer configuration parameters and how they affect consumer behavior. We dedicated a large part of the chapter to discussing offsets and how consumers keep track of them. Understanding how consumers commit offsets is critical when writing reliable consumers, so we took time to explain the different ways this can be done. We then discussed additional parts of the Consumer APIs, handling rebalances, and closing the consumer.

We concluded by discussing the deserializers used by consumers to turn bytes stored in Kafka into Java objects that the applications can process. We discussed Avro deserializers in some detail, even though they are just one type of deserializer you can use, because these are most commonly used with Kafka.

---

1 Diagrams by Sophie Blee-Goldman, from her May 2020 blog post, "From Eager to Smarter in Apache Kafka Consumer Rebalances".