4 Kafka clients

This chapter covers

- Producing records with the KafkaProducer
- Understanding message delivery semantics
- Consuming records with the KafkaConsumer
- Learning about Kafka's exactly-once streaming
- Using the Admin API for programmatic topic management
- Handling multiple event types in a single topic

This chapter is where the "rubber hits the road" and we take what you've learned over the previous two chapters and apply it here to start building event streaming applications. We'll start out by working with the producer and consumer clients individually to gain a deep understanding how each one works.

In their simplest form, clients operate like this: producers send records (in a produce request) to a broker and the broker stores them in a topic and consumers send a fetch request and the broker retrieves records from the topic to fulfill that request. When we talk about the Kafka event streaming platform, it's common to mention both producers and consumers. After all, it's a safe assumption that you are producing data for someone else to consume. But it's very important to understand that the producers and consumers are unaware of each other, there's no synchronization between these two clients.

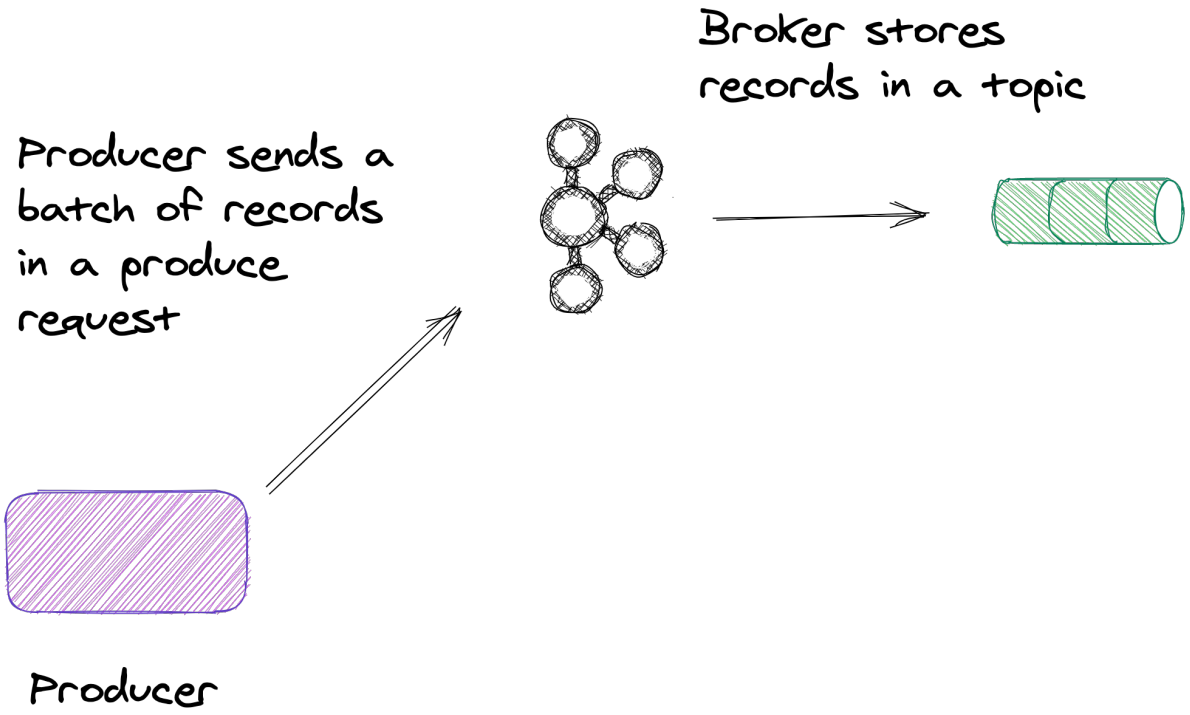

Figure 4.1 Producers send batches of records to Kafka in a produce request

The `KafkaProducer` has just one task, sending records to the broker. The records themselves contain all the information the broker needs to store them.

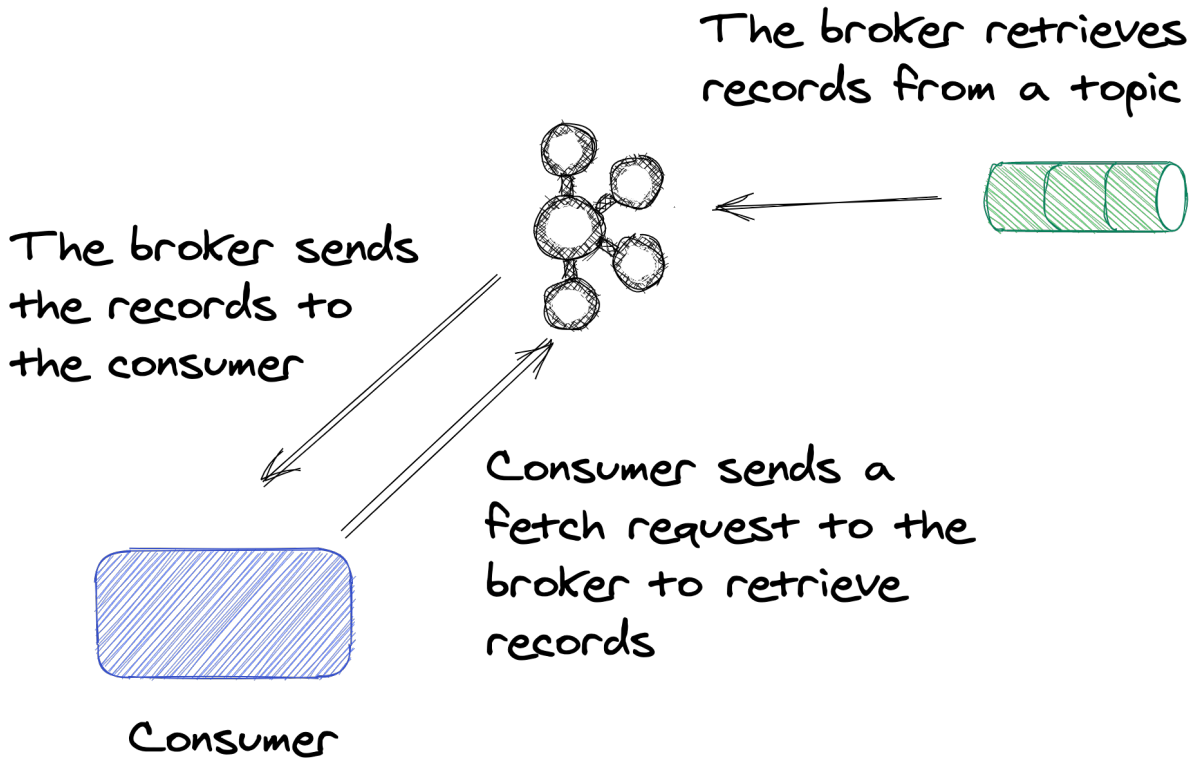

Figure 4.2 Consumers send fetch requests to consume records from a topic, the broker retrieves those records to fulfill the request

The `KafkaConsumer` on the other hand only reads or consumes records from a topic. Also, as we mentioned in the chapter covering the Kafka broker, the broker handles the storage of the records. The act of consuming records has no impact on how long the broker retains them.

In this chapter you'll take a `KafkaProducer` and dive into the essential configurations and walk through examples of producing records to the Kafka broker. Learning how the `KafkaProducer` works is important because that's the crucial starting point for building event streaming applications; getting the records into Kafka.

Next you'll move on to learning how to use the `KafkaConsumer`. Again we'll cover the vital configuration settings and from working with some examples, you'll see how an event streaming application works by continually consuming records from the Kafka broker. You've started your event streaming journey by getting your data into Kafka, but it's when you start consuming the data that you start building useful applications.

Then we'll go into working with the `Admin` interface. As the name implies, it's a client that allows you to perform administrative functions programmatically.

From there you'll get into more advanced subject matter such as the idempotent producer configuration which guarantees you per partition, exactly-once message delivery and the Kafka transnational API for exactly once delivery across multiple partitions.

When you're done with this chapter you'll know how to build event streaming applications using the `KafkaProducer` and `KafkaConsumer` clients. Additionally, you'll have a good understanding how they work so you can recognize when you have a good use-case for including them in your application. You should also come away with a good sense of how to configure the clients to make sure your applications are robust and can handle situations when things don't go as expected.

So with this overview in mind, we are going to embark on a guided tour of how the clients do their jobs. First we'll discuss the producer, then we'll cover the consumer. Along the way we'll take some time going into deeper details, then we'll come back up and continue along with the tour.

4.1 Producing records with the KafkaProducer

You've seen the `KafkaProducer` some in chapter three when we covered Schema Registry, but I didn't go into the details of how the producer works. Let's do that now.

Say you work on the data ingest team for a medium sized wholesale company. You get transaction data delivered via a point of sale service and several different departments within the company want access to the data for things such as reporting, inventory control, detecting trends

etc.

You've been tasked with providing a reliable and fast way of making that information available to anyone within the company that wants access. The company, Vandelay Industries, uses Kafka to handle all of its event streaming needs and you realize this is your opportunity to get involved. The sales data contains the following fields:

1. Product name
2. Per-unit price
3. Quantity of the order
4. The timestamp of the order
5. Customer name

At this point in your data pipeline, you don't need to do anything with the sales data other than to send it into a Kafka topic, which makes it available for anyone in the company to consume

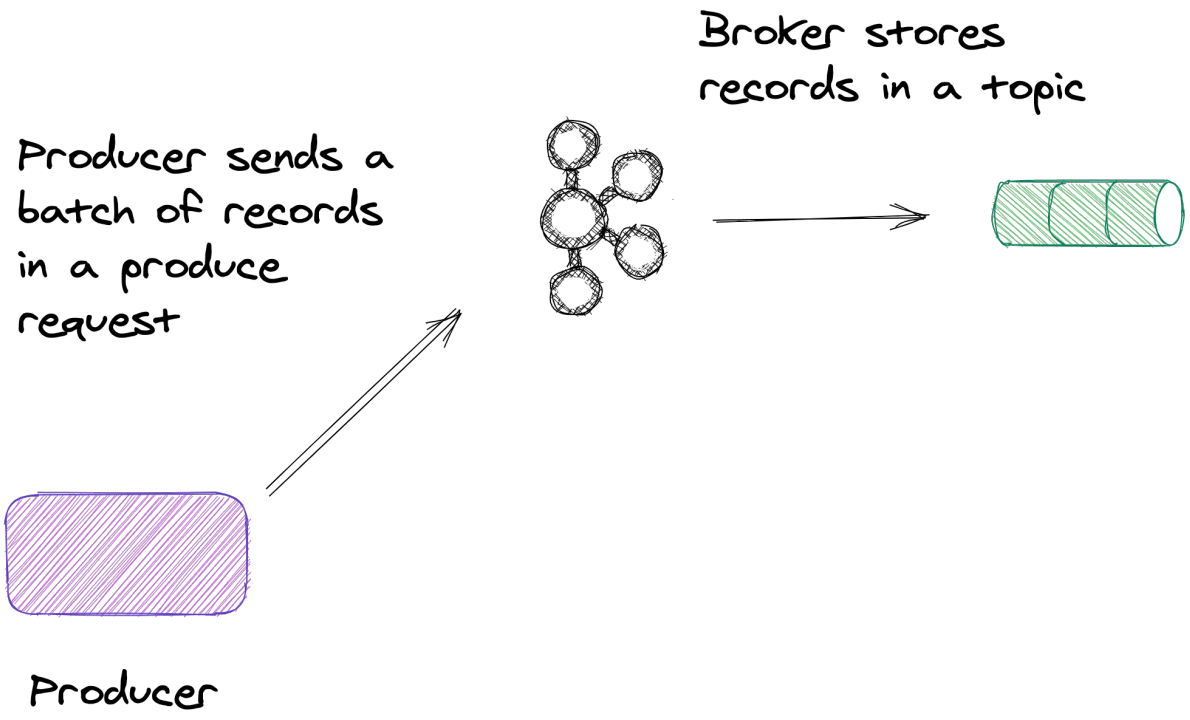

Figure 4.3 Sending the data into a Kafka Topic

To make sure everyone is on the same page with the structure of the data, you've modeled the records with a schema and published it to Schema Registry. All that's left is for you to do write the `KafkaProducer` code to take the sales records and send them into Kafka. Here's what your code looks like

Listing 4.1 A KafkaProducer the source code can be found at bbejeck.chapter_4.sales.SalesProducerClient

```
// There are some details left out for clarity here in the text
try (
Producer<String, ProductTransaction> producer = new KafkaProducer<>(
  producerConfigs)) {   ❶
   while(keepProducing) {
    Collection<ProductTransaction> purchases = salesDataSource.fetch();   ❷
     purchases.forEach(purchase -> {
        ProducerRecord<String, ProductTransaction> producerRecord =
             new ProducerRecord<>(topicName, purchase.getCustomerName(),
              purchase);   ❸
        producer.send(producerRecord,
          (RecordMetadata metadata, Exception exception) -> {      ❹
              if (exception != null) {     ❺
                  LOG.error("Error producing records ", exception);
            } else {
              LOG.info("Produced record at offset {} with timestamp {}",   ❻
                          metadata.offset(), metadata.timestamp());
              }
          });
     });
```

❶ Creating the KafkaProducer instance using a try-with-resources statement so the producer closes automatically when the code exits

❷ The data source providing the sales transaction records

❸ Creating the ProducerRecord from the incoming data

❹ Sending the record to the Kafka broker and providing a lambda for the Callback instance

❺ Logging if an exception occured with the produce request

❻ In the success case logging the offset and timestamp of the record stored in the topic

Notice at annotation one the `KafkaProducer` takes a `Map` of configuration items (In a section following this example we'll dicusss some of the more important `KafkaProducer` configurations). At annotation number 2, we're going to use a data generator to simulate the delivery of sales records. You take the list of `ProductTransaction` objects and use the Java stream API to map each object in the list into a `ProducerRecord` object.

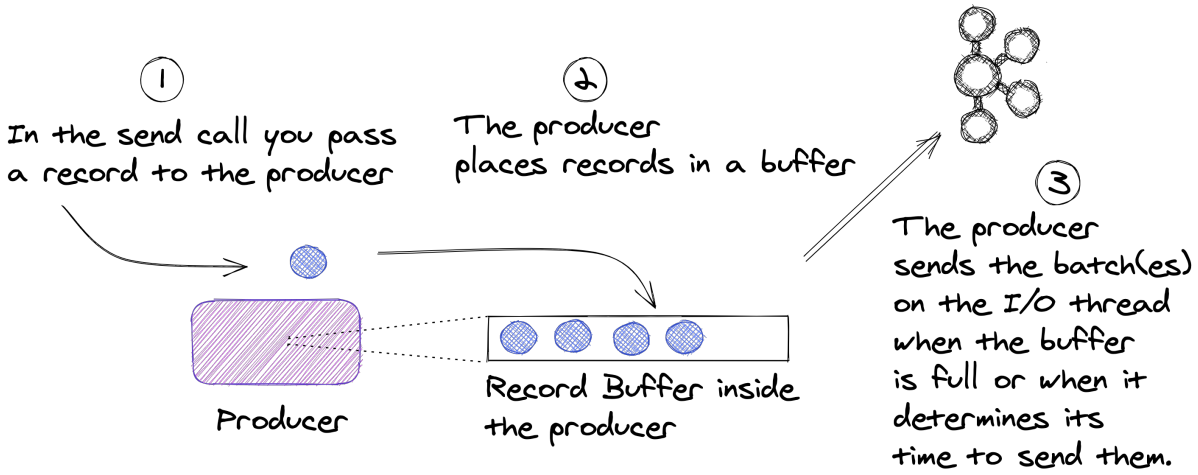

Figure 4.4 The Producer batches records and sends them to the broker when the buffer is full or it's time to send them

For each `ProducerRecord` created you pass it as a parameter to the `KafkaProducer.send()` method. However, the producer does not immediately send the record to the broker, instead it attempts to batch up records. By using batches the producer makes fewer requests which helps with performance on both the broker and the producer client. The `KafkaProducer.send()` call is asynchronous to allow for continually adding records to a batch. The producer has a separate a thread (the I/O thread) that can send records when the batch is full or when it decides it's time so transmit the batch.

There are two signatures for the `send` method. The version you are using in the code here accepts a `ProducerRecord` and `Callback` object as parameters. But since the `Callback` interface only contains one method, also known as functional interface, we can use a lambda expression instead of a concrete implementation. The producer I/O thread executes the `Callback` when the broker acknowledges the record as persisted.

The `Callback.onCompletion` method, again represented here as a lambda, accepts two parameters `RecordMetadata` and `Exception`. The `RecordMetadata` object contains metadata of the record the broker has acknowledged. Referring back to our discussion on the `acks` configuration, the `RecordMetadata.offset` field is `-1` if you have `acks=0`. The offset is `-1` because the producer doesn't wait for acknowledgment from the broker, so it can't report the offset assigned to the record. The exception parameter is non-null if an error occurred.

Since the producer I/O thread executes the callback, it's best if you don't do any heavy processing as that would hold up sending of records. The other overloaded `KafkaProducer.send()` method only accepts a `ProducerRecord` parameter and returns a `Future<RecordMetadata>`. Calling the `Future.get()` method blocks until the broker acknowledges the record (request completion). Note that if an error occurs during the send then executing the `get` method throws an exception.

Generally speaking it's better to use the `send` method with the `Callback` parameter as it's a bit cleaner to have the I/O thread handle the results of the send asynchronously vs. having to keep track of every `Future` resulting from the send calls.

At this point we've covered the fundamental behavior for a `KafkaProducer`, but before we move onto consuming records, we should take a moment to discuss other important subjects involving the producer: configurations, delivery semantics, partition assignment, and timestamps.

4.1.1 Producer configurations

- `bootstrap.servers` - One or more host:port configurations specifying a broker for the producer to connect to. Here we have a single value because this code runs against a single broker in development. In a production setting, you could list every broker in your cluster in a comma separated list.
- `key.serializer` - The serializer for converting the key into bytes. In this example, the key is a `String` so we can use the `StringSerializer` class provided with the Kafka clients. The `org.apache.kafka.common.serialization` package contains serializers for `String`, `Integer`, `Double` etc. You could also use Avro, Protobuf, or JSON Schema for the key and use the appropriate serializer.
- `value.serializer` - The serializer for the value. Here we're using object generated from an Avro schema. Since we're using Schema Registry, we'll use the `KafkaAvroSerializer` we saw from chapter 3. But the value could also be a String, Integer etc and you would use one of the serializers from the `org.apache.kafka.common.serialization` package.
- `acks` - The number of acknowledgments required to consider the produce request successful. The valid values are "0", "1", "all" the `acks` configuration is one of the most important to understand as it has a direct impact on data durability. Let's go through the different settings here.
    - Zero (`acks=0`) Using a value of 0 means the producer will not wait for any acknowledgment from the broker about persisting the records. The producer considers the send successfully immediately after transmitting it to the broker. You could think using `acks=0` as "fire and forget". Using this setting has the highest throughput, but has the lowest guarantee on data durability.
    - One (`acks=1`) A setting of one means the producer waits for notification from the lead broker for the topic-partition that it successfully persisted the record to its log. But the producer doesn't wait for acknowledgment from the leader that any of the followers persisted the record. While you have a little more assurance on the durability of the record in this case, should the lead broker fail before the followers replicate the record, it will be lost.
    - All (`acks=all`) This setting gives the highest guarantee of data durability. In this case, the producer waits for acknowledgment from the lead broker that it successfully persisted the record to its own log ***and*** the following in-sync brokers were able to persist the record as well. This setting has the lowest throughput, but the highest durability guarantees. When using the `acks=all` setting it's advised to set the `min.insync.replicas` configuration for your topics to a value higher than the default of `1`. For example with a replication factor of 3, setting `min.insyc.replicas=2` means the producer will raise an exception if there are not enough replicas available for persisting a record. We'll go into more detail on this scenario later in this chapter.
- `delivery.timeout.ms` - This is an upper bound on the amount of time you want to wait for a response after calling `KafkaProducer.send()`. Since Kafka is a distributed system, failures delivering records to the broker are going to occur. But in many cases these errors are temporary and hence re-tryable. For example the producer may encounter trouble connecting due to a network partition. But network connectivity can be a temporary issue, so the producer will re-try sending the batch and in a lot cases the re-sending of records succeeds. But after a certain point, you'll want the producer to stop

trying and throw an error as prolonged connectivity problems mean there's an issue that needs attention. Note that if the producer encounters what's considered a fatal error, then the producer will throw an exception before this timeout expires.

- `retries` - When the producer encounters an non-fatal error, it will retry sending the record batch. The producer will continue to retry until the `delivery.timeout.ms` timeout expires. The default value for `retries` is `INTEGER.MAX_VALUE`. Generally you should leave the retried configuration at the default value. If you want to limit the amount of retries a producer makes, you should reduce the amount of time for the `delivery.timeout.ms` configuration. With errors and retries it's possible that records could arrive out of order to the same partition. Consider the producer sends a batch of records but there is an error forcing a retry. But in the intervening time the producer sends a second batch that did not encounter any errors. The first batch succeeds in the subsequent retry, but now it's appended to the topic ***after*** the second batch. To avoid this issue you can set the configuration `max.in.flight.requests.per.connection=1`. Another approach to avoid the possibility of out of order batches is to use the `idempotent producer` which we'll discuss in 4.3.1 in this chapter.

Now that you have learned about the concept of retries and record acknowledgments, let's look at message delivery semantics now.

4.1.2 Kafka delivery semantics

Kafka provides three different delivery semantic types: at least once, at most once, and exactly once. Let's discuss each of them here.

- At least once - With "at least once" a records are never lost, but may be delivered more than once. From the producer's perspective this can happen when a producer sends a batch of records to the broker. The broker appends the records to the topic-partition, but the producer does not receive the acknowledgment in time. In this case the producer re-sends the batch of records. From the consumer point of view, you have processed incoming records, but before the consumer can commit, an error occurs. Your application reprocesses data from the last committed offset which includes records already processed, so there are duplicates as a result. Records are never lost, but may be delivered more than once. Kafka provides at least once delivery by default.
- At most once - records are successfully delivered, but may be lost in the event of an error. From the producer standpoint enabling `acks=0` would be an example of at most once semantics. Since the producer does not wait for any acknowledgment as soon as it sends the records it has no notion if the broker either received them or appended them to the topic. From the consumer perspective, it commits the offsets before confirming a write so in the event of an error, it will not start processing from the missed records as the consumer already committed the offsets. To achieve at "at most once" producers set `acks=0` and consumers commit offsets before doing any processing.
- Exactly once - With "exactly once" semantics records are neither delivered more than once or lost. Kafka uses transactions to achieve exactly once semantics. If a transaction is aborted, the consumers internal position gets reset to the offset prior to the start of the transaction and the stored offsets aren't visible to any consumer configured with `read_committed`.

Both of these concepts are critical elements of Kafka's design. Partitions determine the level of parallelism and allow Kafka to distribute the load of a topic's records to multiple brokers in a

cluster. The broker uses timestamps to determine which log segments it will delete. In Kafka Streams, they drive progress of records through a topology (we'll come back to timestamps in the Kafka Streams chapter).

4.1.3 Partition assignment

When it comes to assigning a partition to a record, there are three possibilities:

1. If you provide a valid partition number, then it's used when sending the record
2. If you don't give the partition number, but there is a key, then the producer sets the partition number by taking the hash of the key modulo the number of partitions.
3. Without providing a partition number or key, the `KafkaProducer` sets the partition by alternating the partition numbers for the topic. The approach to assigning partitions without keys has changed some as of the 2.4 release of Kafka and we'll discuss that change now.

Prior to Kafka 2.4, the default partitioner assigned partitions on a round-robin basis. That meant the producer assigned a partition to a record, it would increment the partition number for the next record. Following this round-robin approach, results in sending multiple, smaller batches to the broker. The following illustration will help clarify what is going on:

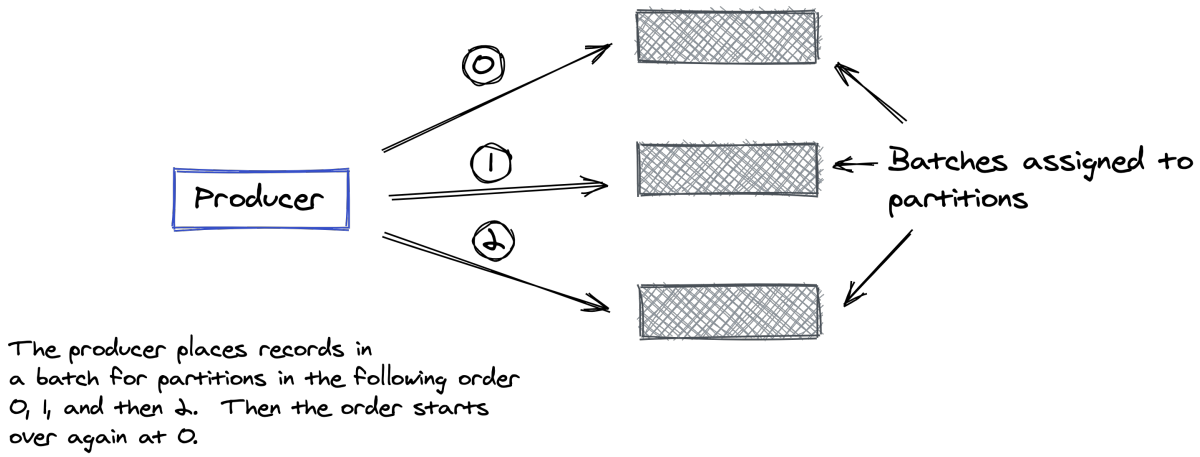

Figure 4.5 Round robin partition assignment

This approach also led to more load on the broker due to a higher number of requests.

But now when you don't provide a key or partition for the record, the partitioner assigns a partition for the record per batch. This means when the producer flushes its buffer and sends records to the broker, the batch is for single partition resulting in a single request. Let's take a look at an illustration to visualize how this works:

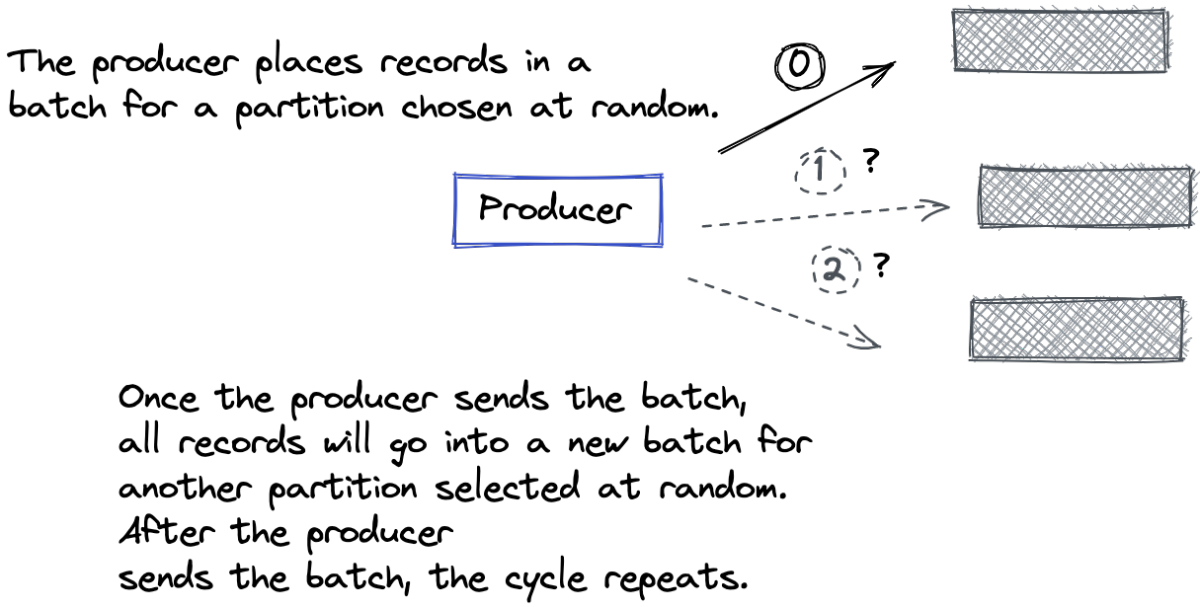

Figure 4.6 Sticky partition assignment

After sending the batch, the partitioner selects a partition at random and assigns it to the next batch. In time, there's still an even distribution of records across all partitions, but it's done one batch at a time.

Sometimes the provided partitioners may not suit your requirements and you'll need finer grained control over partition assignment. For those cases you can write your own custom partitioner.

4.1.4 Writing a custom partitioner

Let's revisit the producer application from the 4.1 section above. The key is the name of the customer, but you have some orders that don't follow the typical process and end up with a customer name of "CUSTOM" and you'd prefer to restrict those orders to a single partition 0, and have all other orders on partition 1 or higher.

So in this case, you'll need to write a custom partitioner that can look at the key and return the appropriate partition number.

The following example custom partitioner does just that. The `CustomOrderPartitioner` (from src/main/java/bbejeck/chapter_4/sales/CustomOrderPartitioner.java) examines the key to determine which partition to use.

Listing 4.2 `CustomOrderPartitioner` custom partitioner

```
public class CustomOrderPartitioner implements Partitioner {

// Some details omitted for clarity

@Override
public int partition(String topic,
                     Object key,
                     byte[] keyBytes,
                     Object value,
                     byte[] valueBytes,
                     Cluster cluster) {

    Objects.requireNonNull(key, "Key can't be null");
    int numPartitions = cluster.partitionCountForTopic(topic);   ❶
    String strKey = (String) key;
    int partition;

    if (strKey.equals("CUSTOM")) {
        partition = 0;    ❷
    } else {
        byte[] bytes = strKey.getBytes(StandardCharsets.UTF_8);
        partition = Utils.toPositive(Utils.murmur2(bytes)) %
                                     (numPartitions - 1) + 1;   ❸
    }
    return partition;
  }
}
```

❶ Retrieve the number of partitions for the topic

❷ If the name of the customer is "CUSTOM" return 0

❸ Determine the partition to use in the non-custom order case

To create your own partitioner you implement the `Partitioner` interface which has 3 methods, `partition`, `configure`, and `close`. I'm only showing the `partition` method here as the other two are no-ops in this particular case. The logic is straight forward; if the customer name equates to "CUSTOM", return zero for the partition. Otherwise you determine the partition as usual, but with a small twist. First we subtract one from the number of candidate partitions since the 0 partition is reserved. Then we shift the partiton number by 1 which ensures we always return 1 or greater for the non-custom order case.

> **NOTE** **This example does not represent a typical use case and is presented only for the purpose of demonstrating how to you can provide a custom partitioner. In most cases it's best to go with one of the provided ones.**

You've just seen how to construct a custom partitioner and next we'll wire it up with our producer.

4.1.5 Specifying a custom partitioner

Now that you've written a custom partitioner, you need to tell the producer you want to use it instead of the default partitioner. You specify a different partitioner when configuring the Kafka producer:

```
producerConfigs.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
 CustomOrderPartitioner.class);
```

The bbejeck.chapter_4.sales.SalesProducerClient is configured to use the `CustomOrderPartitioner`, but you can simply comment out the line if you don't want to use it. You should note that since the partitioner config is a producer setting, it must be done on each one you want to use the custom partitioner.

4.1.6 Timestamps

The `ProducerRecord` object contains a timestamp field of type `Long`. If you don't provide a timestamp, the `KafkaProducer` adds one to the record, which is simply the current time of the system the producer is running on. Timestamps are an important concept in Kafka. The broker uses them to determine when to delete records, by taking the oldest timestamp in a segment and comparing it to the current time. If the difference exceeds the configured retention time, the broker removes the segment. Kafka Streams and ksqlDB also rely heavily on timestamps, but I'll defer those discussions until we get to their respective chapters.

There are two possible timestamps that Kafka may use depending on the configuration of the topic.

In Kafka topics have a configuration, `message.timestamp.type` which can either be `CreateTime` or `LogAppendTime`. A configuration of `CreatTime` means the broker stores the record with the timestamp provided by the producer. If you configure your topic with `LogAppendTime`, then the broker overwrites the timestamp in the record with its current wall-clock (i.e, system) time when the broker appends the record in the topic. In practice, the difference between these timestamps should be close.

Another consideration is that you can embed the timestamp of the event in payload of the record value when you are creating it.

This wraps up our discussion on the producer related issues. Next we'll move on to the mirror image of producing records to Kafka, consuming records.

4.2 Consuming records with the KafkaConsumer

So you're back on the job at Vandelay Industries and you now have a new task. Your producer application is up an running happily pushing sales records into a topic. But now you're asked to develop a `KafkaConsumer` application to serve as a model for consuming records from a Kafka topic.

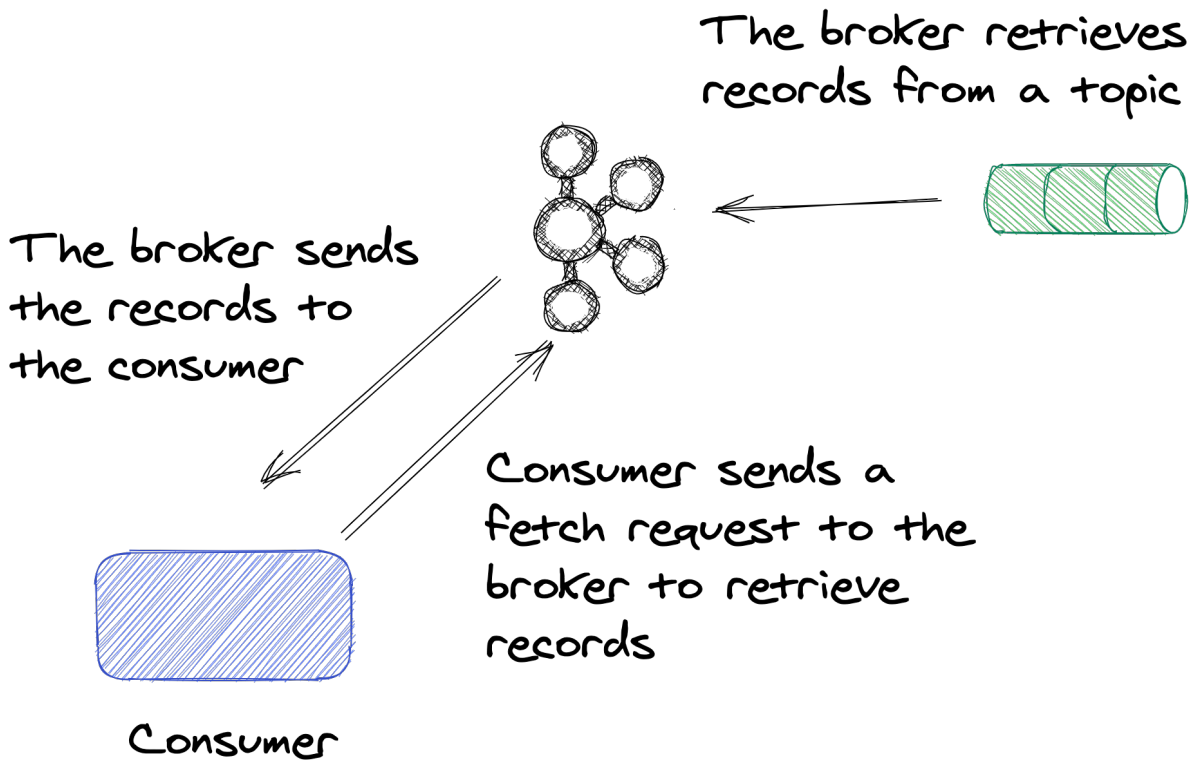

Figure 4.7 Consumers send fetch requests to consume records from a topic, the broker retrieves those records to fulfill the request

The `KafkaConsumer` sends a fetch request to the broker to retrieve records from topics it's subscribed to. The consumer makes what known as a `poll` call to get the records. But each time the consumer polls, it doesn't necessarily result in the broker fetching records. Instead it could be retrieving records cached by a previous call.

> **NOTE** There are producer and consumer clients available in other programming languages, but in this book we'll focus on the clients available in the Apache Kafka distribution, which are written in Java. To see a list of clients available in other languages checkout take a look at this resource docs.confluent.io/platform/current/clients/index.html#ak-clients

Let's get started by looking at the code for creating a `KafkaConsumer` instance:

Listing 4.3 KafkaConsumer code found in bbejeck.chapter_4.sales.SalesConsumerClient

```
// Details left out for clarity
 try (
  final Consumer<String, ProductTransaction> consumer = new KafkaConsumer<>(
    consumerConfigs)) {   ❶
    consumer.subscribe(topicNames);     ❷
    while (keepConsuming) {
        ConsumerRecords<String, ProductTransaction> consumerRecords =
          consumer.poll(Duration.ofSeconds(5));   ❸
        consumerRecords.forEach(record -> {       ❹
            ProductTransaction pt = record.value();
            LOG.info("Sale for {} with product {} for a total sale of {}",
                    record.key(),
                    pt.getProductName(),
                    pt.getQuantity() * pt.getPrice());
        });
    }
}
```

❶ Creating the new consumer instance

❷ Subscribing to topic(s)

❸ Polling for records

❹ Doing some processing with each of the returned records

In this code example, you're creating a `KafkaConsumer`, again using the try-with-resources statement. After subscribing to a topic or topics, you begin processing records returned by the `KafkaConsumer.poll` method. When the `poll` call returns records, you start doing some processing with them. In this example case we're simply logging out the details of the sales transactions.

TIP **Whenever you create either a `KafkaProducer` or `KafkaConsumer` you need to close them when your done to make sure you clean up all of the threads and socket connections. The try-with-resources (docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) in Java ensures that resources created in the `try` portion are closed at the end of the statement. It's a good practice to always use the try-with-resources statement as it's easy to overlook adding a `close` call on either a producer or a consumer.**

You'll notice that just like with the producer, you create a `Map` of configurations and pass them as a parameter to the constructor. Here I'm going to show you some of the more prominent ones.

- `bootstrap.servers` - One or more host:port configurations specifying a broker for the consumer to connect to. Here we have a single value, but this could be a comma separated list.
- `max.poll.interval.ms` - The maximum amount of time a consumer can take between calls to `KafkaConsumer.poll()` otherwise the consumer group coordinator considers

the individual consumer non-active and triggers a rebalance. We'll talk more about the consumer group coordinator and relabances in this section.

- `group.id` - An arbitrary string value used to associate individual consumers as part of the same consumer group. Kafka uses the concept of a consumer group to logically map multiple consumers as one consumer.
- `enable.auto.commit` - A boolean flag that sets whether the consumer will automatically commit offsets. If you set this to false, your application code must manually commit the offsets of records you considered successfully processed.
- `auto.commit.interval.ms` - The time interval at which offsets are automatically committed.
- `auto.offset.reset` - When a consumer starts it will resume consuming from the last committed offset. If offsets aren't available for the consumer then this configuration specifies where to start consuming records, either the earliest available offset or the latest which means the offset of the next record that arrives after the consumer started.
- `key.deserializer.class` - The classname of the deserializer the consumer uses to convert record key bytes into the expected object type for the key.
- `value.deserializer.class` - The classname of the deserializer the consumer uses to convert record value bytes into the expected object type for the value. Here we're using the provided `KafkaAvroDeserializer` for the value which requires the `schema.registry.url` configuration which we have in our configuration.

The code we use in our first consumer application is fairly simple, but that's not the main point. Your business logic, what you do when you consume the records is always going to be different on a case-by-case basis.

It's more important to grasp how the `KafkaConsumer` works and the implications of the different configurations. By having this understanding you'll be in a better position to know how to best write the code for performing the desired operations on the consumed records. So just as we did in the producer example, we're going to take a detour from our narrative and go a little deeper on the implications of these different consumer configurations.

4.2.1 The poll interval

Let's first discuss the roll of `max.poll.interval.ms`. It will be helpful to look at an illustration of what the poll interval configuration in action to get a full understanding:

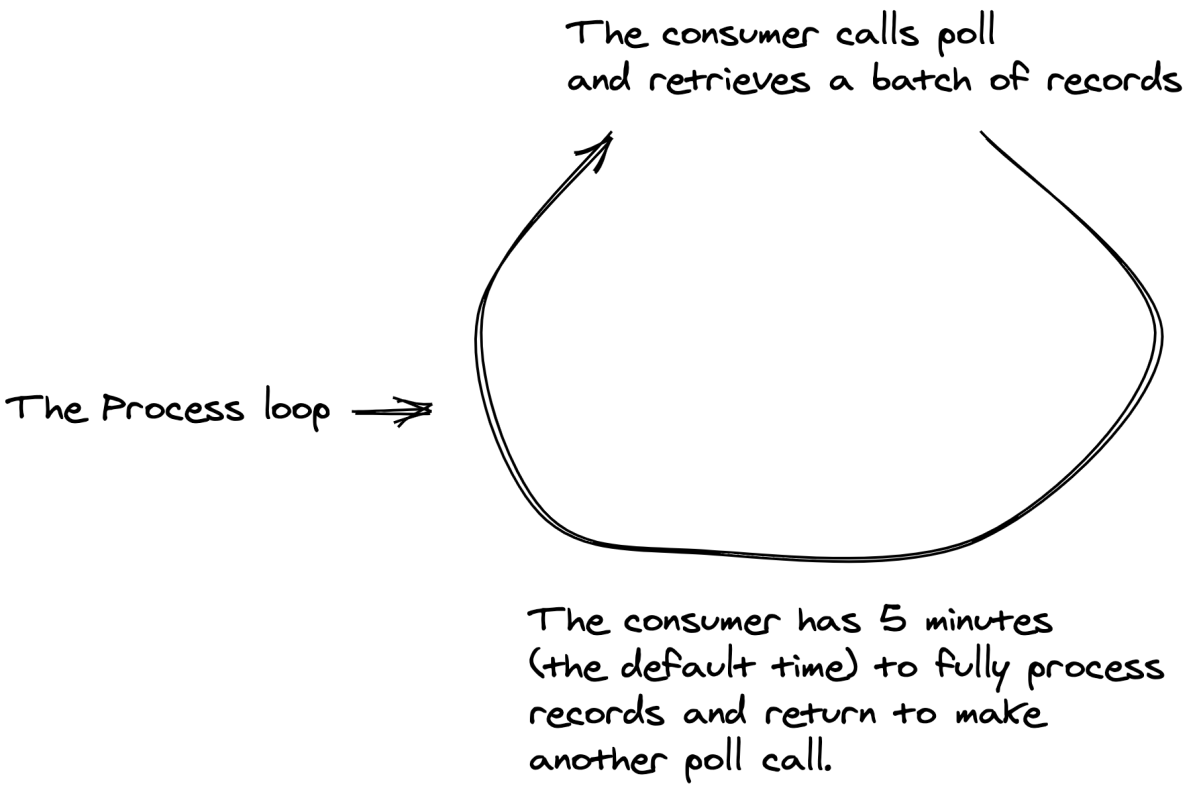

Figure 4.8 The max.poll.interval.ms configuration specifies how long a consumer may take between calls to `KafkaConsumer.poll()` before the consumer is considered inactive and removed from the consumer group

In the illustration here, the consumer processing loop starts with a call to `KafkaConsumer.poll(Duration.ofSeconds(5))`, the time passed to the `poll(Duration)` call is the maximum time the consumer waits for new records, in this case five seconds. When the `poll(Duration)` call returns, if there are any records present, the `for` loop over the `ConsumerRecords` executes your code over each one. Had there been no records returned, the outer `while` loop simply goes back to the top for another `poll(Duration)` call.

Going through this illustration, iterating over all the records and execution for each record must complete before the `max.poll.interval.ms` time elapses. By default this value is five minutes, so if your processing of returned records takes longer, then that individual consumer is considered dead and the group coordinator removes the consumer from the group and triggers a rebalance. I know that I've mentioned a few new terms in group coordinator and rebalancing, we'll cover them in the next section when we cover the `group.id` configuration.

If you find that your processing takes longer than the `max.poll.interval.ms` there are a couple of things you can do. The first approach would be to validate what you're doing when processing the records and look for ways to speed up the processing. If you find there's no changes to make to your code, the next step could be to to reduce the maximum number of records the consumer

retrieves from a `poll` call. You can do this by setting the `max.poll.records` configuration to a setting less than the default of 500. I don't have any recommendations, you'll have to experiment some to come up with a good number.

4.2.2 Group id

The `group.id` configuration takes into a deeper conversation about consumer groups in Kafka. Kafka consumers use a `group.id` configuration which Kafka uses to map all consumers with the same `group.id` into the same consumer group. A consumer group is a way to logically treat all members of the group as one consumer. Here's an illustration to demonstrating how group membership works:

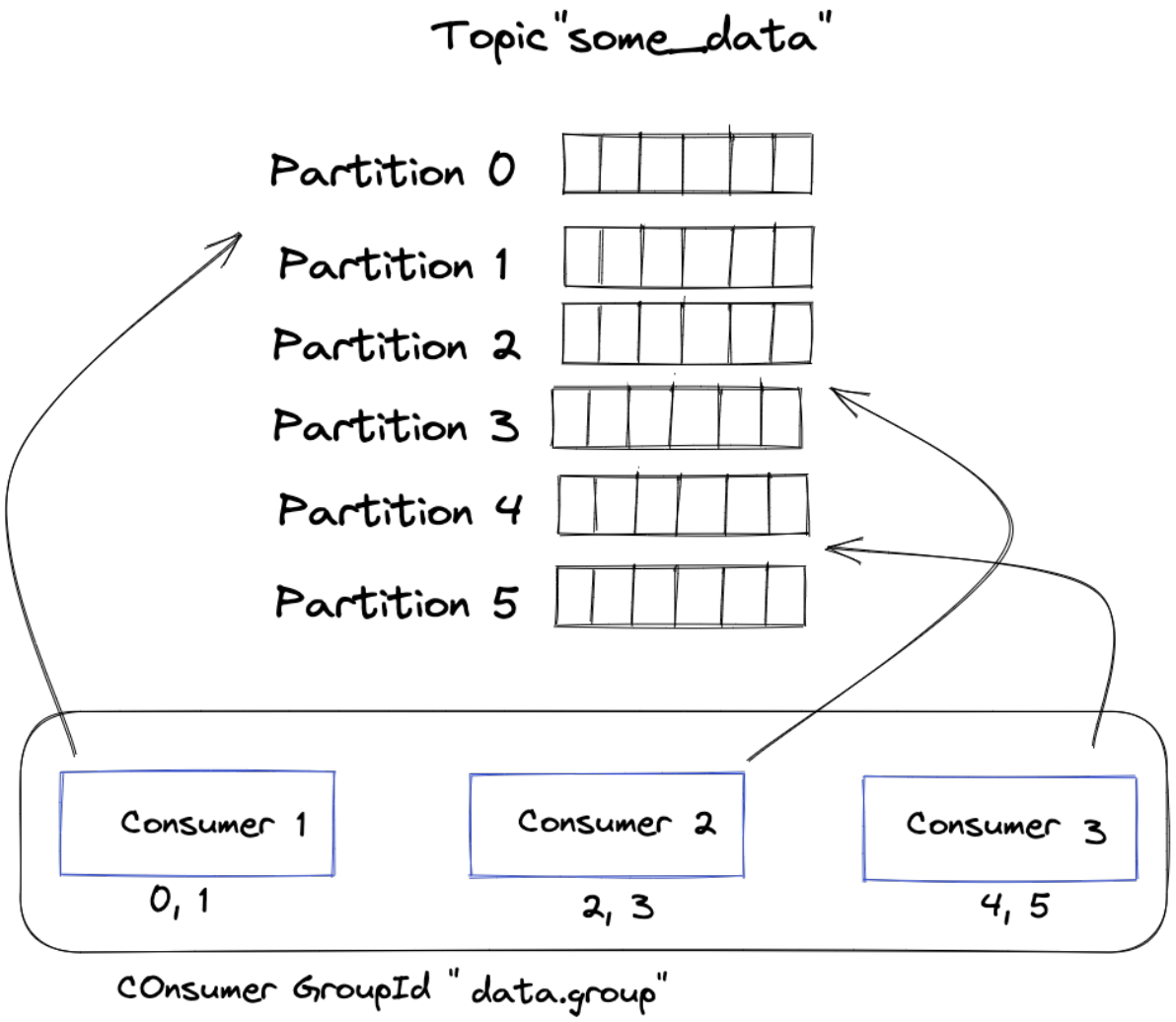

Figure 4.9 Consumer groups allow for assigning topic-partitions across multiple consumers

So going off the image above, there is one topic with six partitions. There are three consumers in the group, so each consumer has an assignment of two partitions. Kafka guarantees that only a single consumer maintains an assignment for a given topic-partition. To have more than one consumer assigned to a single topic-partition would lead to undefined behavior.

Life with distributed systems means that failures aren't to be avoided, but embraced with sound practices to deal with them as they occur. So what happens with our scenario here if one of the consumers in the group fails whether from an exception or missing a required timeout like we described above with the `max.poll.interval.ms` timeout? The answer is the Kafka rebalance protocol, depicted below:

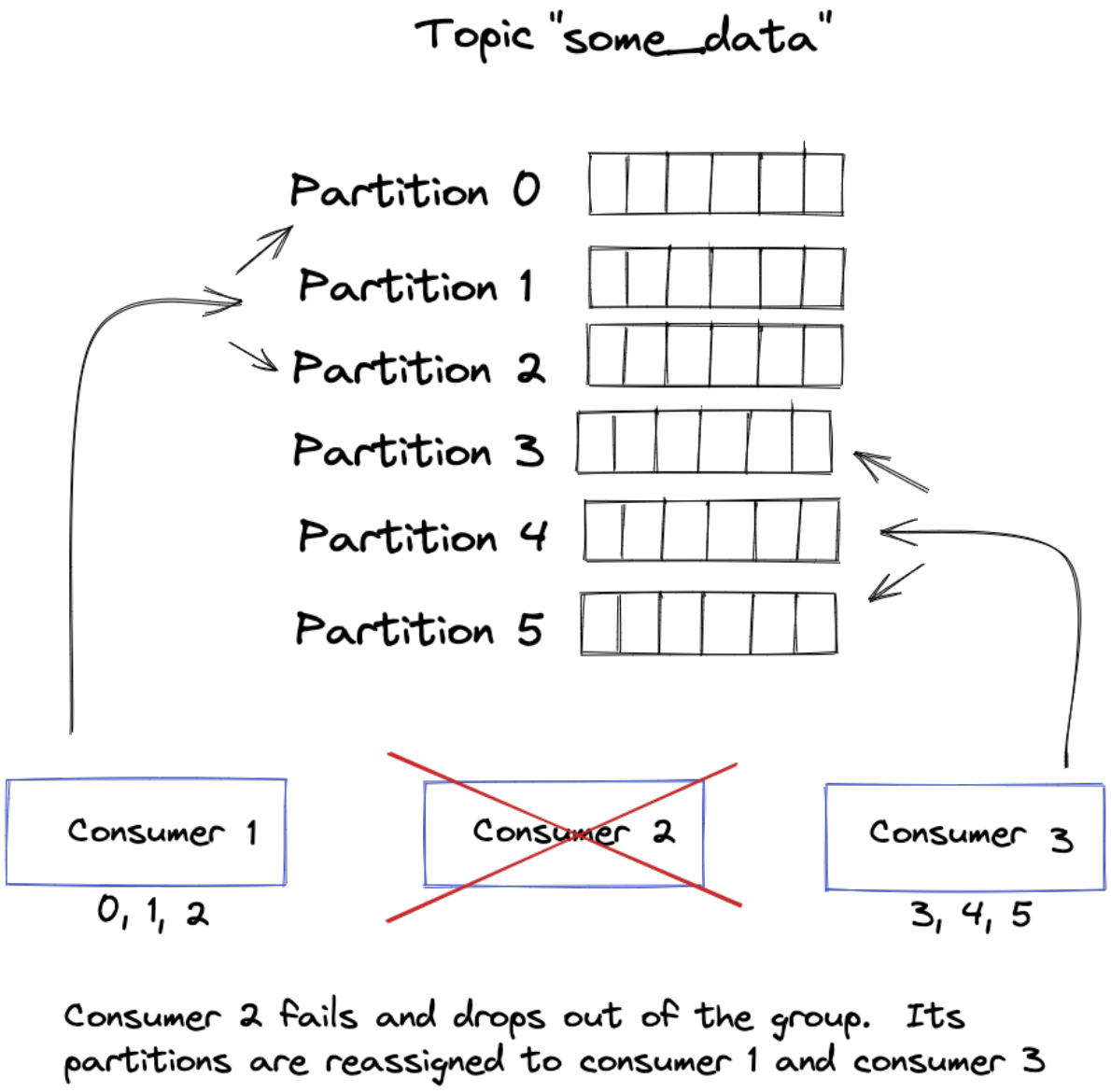

Figure 4.10 The Kafka rebalance protocol re-assigns topic-partitions from failed consumers to still alive ones

What we see here is that consumer-two fails and can longer function. So rebalancing takes the topic-partitions owned by consumer-two and reassigns one topic-partition each to other active consumers in the group. Should consumer-two become active again (or another consumer join the group), then another rebalance occurs and reassigns topic-partitions from the active members and each group member will be responsible for two topic-partitions each again.

NOTE The number of active consumers that you can have is bounded by the number of partitions. From our example here, you can start up to six consumers in the group, but any more beyond six will be idle. Also note that different groups don't affect each other, each one is treated independently.

So far, I've discussed how not making a `poll()` call within the specified timeout will cause a consumer to drop out of the group triggering a rebalance and assigning its topic-partition assignments to other consumers in the group. But if you recall the default setting for `max.poll.interval.ms` is five minutes. Does this mean it takes up to five minutes for potentially dead consumer to get removed from the group and its topic-partitions reassigned? The answer is no and let's look at the poll interval illustration again but we'll update it to reflect session timeouts:

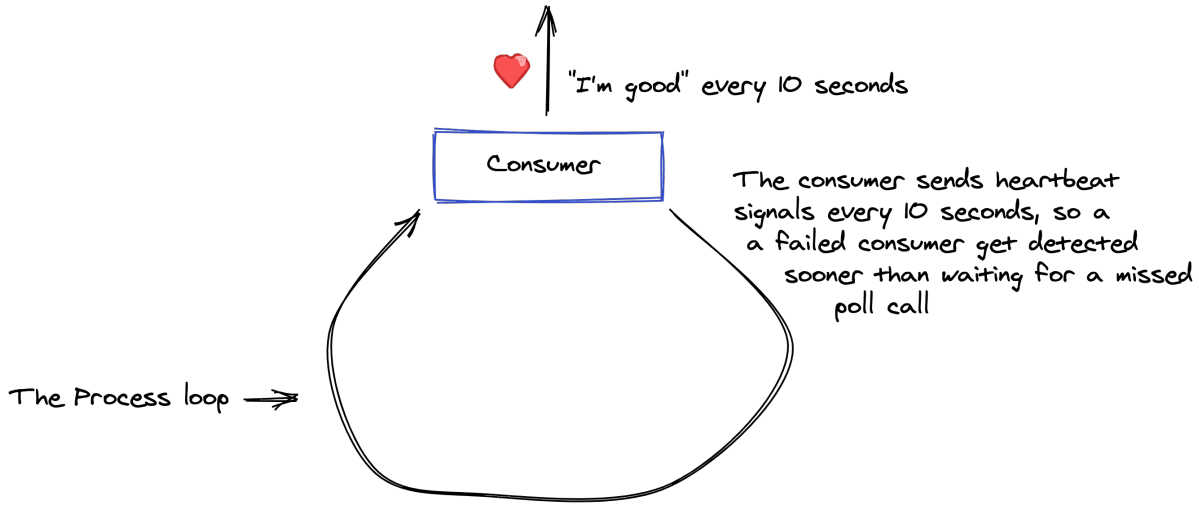

Figure 4.11 In addition to needing to call poll within the timeout, a consumer must send a heartbeat every ten seconds

There is another configuration timeout the `session.timeout.ms` which is set at ten seconds for default value. Each `KafkaConsumer` runs a separate thread for sending heartbeats indicating its still alive. Should a consumer fail to send a heartbeat within ten seconds, it's marked as dead and removed from the group, triggering a rebalance. This two level approach for ensuring consumer liveliness is essential to make sure all consumers are functioning and allows for reassigning their topic-partition assignments to other members of the group to ensure continued processing should one of them fail.

To give you a clear picture of how group membership works, let's discuss a the new terms group coordinator, rebalancing, and the group leader I just spoke about. Let's start with a visual representation of how these parts are tied together:

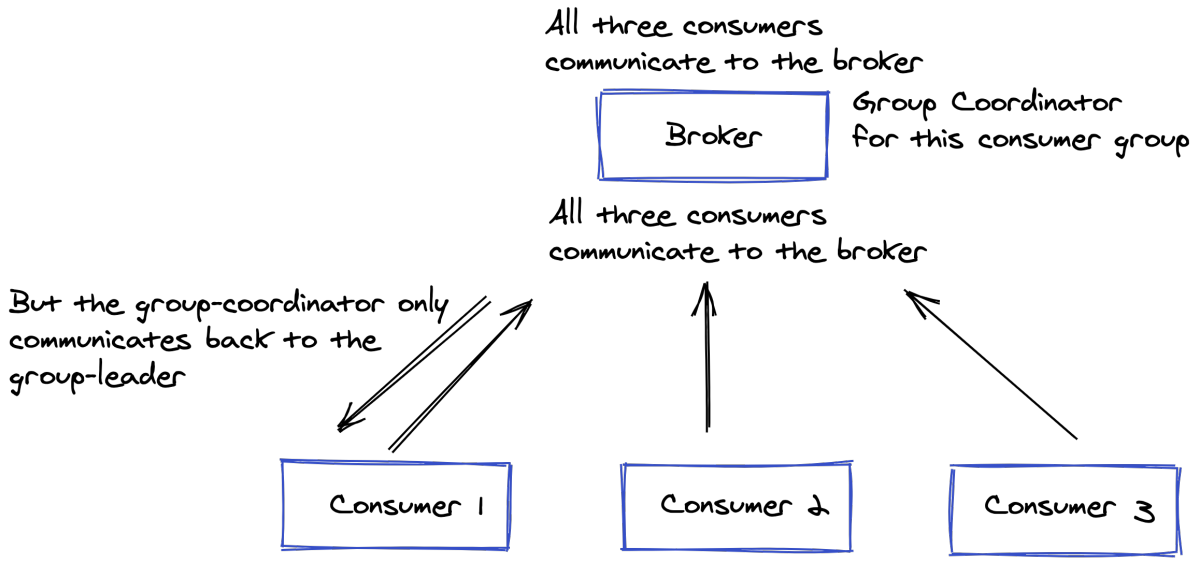

Figure 4.12 Group coordinator is a broker assigned to track a subset of consumer groups and the group leader is a consumer that communicates with the group coordinator

The group coordinator is a broker that handles membership for subset of all available consumer groups. Not one single broker will act as the group coordinator, the responsibility for that is spread around the different brokers. The group coordinator monitors the membership of a consumer group via requests to join a group or when a member is considered dead when it fails to communicate (either a poll or heartbeat) within the given timeouts.

When the group coordinator detects a membership change it triggers a rebalance for the existing members.

A rebalance is the process of having all members of the group rejoin so that group resources (topic-partitions) can be evenly (as possible) distributed to the other members. When a new member joins, then some topic partitions are removed from some or all members of the existing group and are assigned to the new member. When an existing member leaves, the opposite process occurs, its topic-partitions are reassigned to the other active members.

The rebalance process is fairly straight forward, but it comes at a cost of time lost processing waiting for the rebalance process to complete, known as a "stop-the-world" or a eager rebalance. But with the release of Kafka 2.4, there’s new rebalance protocol you can use called cooperative rebalancing.

Let’s take a quick look at both of these protocols, first with the eager rebalancing

EAGER REBALANCING

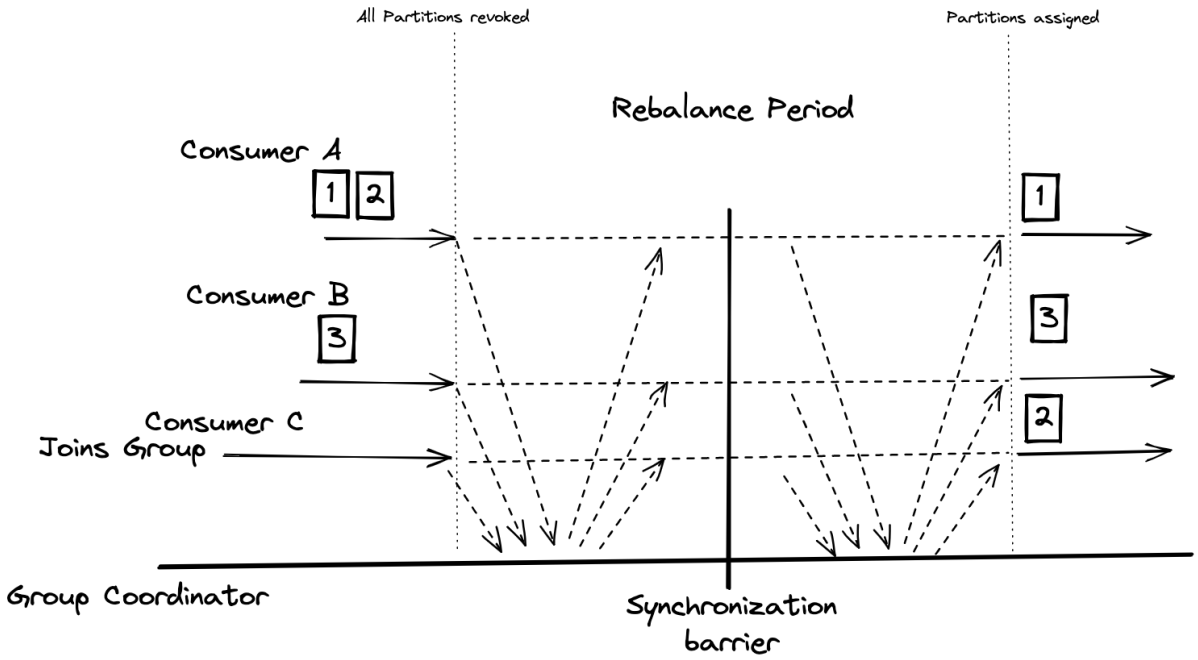

Figure 4.13 Rebalancing with the eager or "stop-the-world" approach processing on all partitions stops until reassigned but most of the partitions end up on with the original consumer

When the group coordinator detects a change in membership it triggers a rebalance. This is true of both rebalance protocols we're going to discuss.

Once the rebalance process initiates, each member of the group first gives up ownership of all its assigned topic-partitions. Then they send a `JoinGroup` request to the controller. Part of the request includes the topic-partitions that consumer is interested in, the ones they just relinquished control of. As a consequence of the consumers giving up their topic partitions is that processing now stops.

The controller collects all of the topic-partition information from the group and sends out the `JoinGroup` response, but the group leader receives all of included topic-partition information.

> **NOTE** **Remember from chapter two in our discussion of the broker all actions are executed in a request/response process.**

The group leader takes this information and creates topic-partition assignments for all members of the group. Then the group leader sends assignment information to the coordinator in a `SyncGroup` request. Note that the other members of the group also send `SyncGroup` requests, but don't include any assignment information. After the group controller receives the assignment information from the leader, all members of the group get their new assignment via the `SyncGroup` response.

Now with their topic-partition assignments, all members of the group begin processing again. Take note again that no processing occurred from the time group members sent the `JoinGroup` request until the `SyncGroup` response arrived with their assignments. This gap in processing is known as a synchronization barrier, and is required as it's important to ensure that each topic-partition only has one consumer owner. If a topic-partition had multiple owners, undefined behavior would result.

> **NOTE** **During this entire process, consumer clients don't communicate with each other. All the consumer group members communicate only with the group coordinator. Additionally only one member of the group, the leader, sets the topic-partition assignments and sends it to the coordinator.**

While the eager rebalance protocol gets the job done of redistributing resources and ensuring only one consumer owns a given topic-partition, it comes at a cost of downtime as each consumer is idle during the period from the initial `JoinGroup` request and the `SyncGroup` response. For smaller applications this cost might be negligible, but for applications with several consumers and a large number of topic-partitions, the cost of down time increases. Fortunately there's another rebalancing approach that aims to remedy this situation.

INCREMENTAL COOPERATIVE REBALANCING

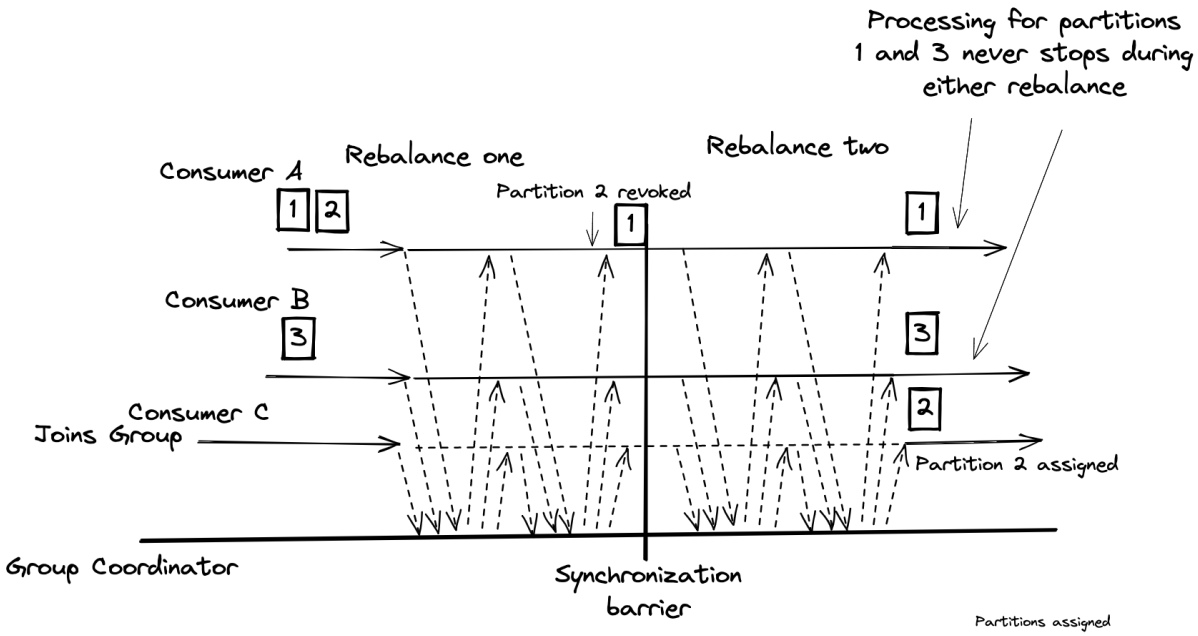

Figure 4.14 Rebalancing with cooperative approach processing continues and only stops for partitions that will be reassigned

Introduced in the 2.4 Kafka release the incremental cooperative rebalance protocol takes the approach that relabances don't need to be so expensive. The incremental cooperative rebalancing

approach takes a different view of rebalancing that we can summarize below:

1. Consumers don't automatically give up ownership of all their topic-partitions
2. The group leader identifies specific topic-partitions requiring new ownership
3. Processing continues for topic-partitions that ***are not changing ownership***

The third bullet point here is the big win (in my opinion) with the cooperative rebalancing approach. Instead of the "stop the world" approach, only those topic-partitions which are moving will experience a pause in processing. In other words, the synchronization barrier is much smaller.

I'm skipping over some of some details, so let's walk through the process of the incremental cooperative rebalancing protocol.

Just like before when the group controller detects a change in group membership, it triggers a rebalance. Each member of the group encodes their current topic-partition subscriptions in a `JoinGroup` request, but each member retains ownership for the time being.

The group coordinator assembles all the subscription information and in the `JoinGroup` response the group leader looks at the assignments and determines which topic-partitions, if any, need to migrate to new ownership. The leader removes any topic-partitions requiring new ownership from the assignments and sends the updated subscriptions to the coordinator via a `SyncGroup` request. Again, each member of the group sends a `SyncGroup` request, but only the leaders` request contains the subscription information.

> **NOTE** **All members of the group receive a `JoinGroup` response, but only the response to the group leader contains the assignment information. Likewise, each member of the group issues a `SyncGroup` group request, but only the leader encodes a new assignment. In the `SyncGroup` response, all members receive their respective, possible updated assignment.**

The members of group take the `SyncGroup` response and potentially calculate a new assignment. Either revoking topic-partitions that are not included or adding ones in the new assignment but not the previous one. Topic-partitions that are included in both the old and new assignment require no action.

Members then trigger a second rebalance, but only topic-partitions changing ownership are included. This second rebalance acts as the synchronization barrier as in the eager approach, but since it only includes topic partitions receiving new owners, it is much smaller. Additionally, topic-partitions that are not moving, continue to process records!

After this discussion of the different rebalance approaches, we should cover some broader information about partition assignment strategies available and how you apply them.

APPLYING PARTITION ASSIGNMENT STRATEGIES

We've already discussed that a broker serves as a group coordinator for some subset of consumer groups. Since two different consumer groups could have different ideas of how to distribute resources (topic-partitions), the responsibility for which approach to use is entirely on the client side.

To choose the partition strategy you want your the `KafkaConsumer` instances in a group to use, you set the the `partition.assignment.strategy` by providing a list of supported partition assignment strategies. All of the available petitioners implement the `ConsumerPartitionAssignor` interface. Here's a list of the available assignors with a brief description of the functionality each one provides.

NOTE **For Kafka Connect and Kafka Streams, which are abstractions built on top of Kafka producers and consumers, use cooperative rebalance protocols and I'd generally recommend to stay with the default settings. This discussion about partitioners is to inform you of what's available for applications directly using a `KafkaConsumer`.**

- RangeAssignor - This is the default setting. The `RangeAssignor` uses an algorithm of sorting the partitions in numerical order and assigning them to consumers by dividing the number of available partitions by number of available consumers. This strategy assigns partition to consumers in lexicographical order.
- RoundRobinAssignor - The `RoundRobinAssignor` takes all available partitions and assigns a partition to each available member of the group in a round-robin manner.
- StickyAssignor - The `StickyAssignor` attempts to assign partitions in a balanced manner as possible. Additionally, the `StickyAssignor` attempts to always preserve existing assignments during a rebalance as much as possible. The `StickyAssignor` follows the eager rebalancing protocol.
- CooperativeStickyAssignor - The `CooperativeStickyAssignor` follows the same assignment algorithm as the `StickyAssignor`. The difference lies in fact that the `CooperativeStickyAssignor` uses the cooperative rebalance protocol.

While it's difficult to provide concrete advice as each use case requires careful analysis of its unique needs, in general for newer applications one should favor using the `CooperativeStickyAssignor` for the reasons outlined in the section on incremental cooperative rebalancing.

TIP **If you are upgrading from a version of Kafka 2.3 or earlier you need to follow a specific upgrade path found in the 2.4 upgrade documentation (kafka.apache.org/documentation/{hash}upgrade_240_notable) to safely use the cooperative rebalance protocol.**

We've concluded our coverage of consumer groups and how the rebalance protocol works. Next

we'll cover a different configuration - static membership, that when a consumer leaves the group, there's no initial rebalance.

4.2.3 Static membership

In the previous section you learned that when a consumer instance shuts down it sends a leave group request to the group controller. Or if it's considered unresponsive by the controller, it gets removed from the consumer group. Either way the end result is the same, the controller triggers a rebalance to re-assign resources (topic-partitions) to the remaining members of the group.

While this protocol is exactly what you want to keep your applications robust, there are some situations where you'd prefer slightly different behavior. For example, let's say you have several consumer applications deployed. Any time you need to update the applications, you might do what's called a rolling upgrade or restart.

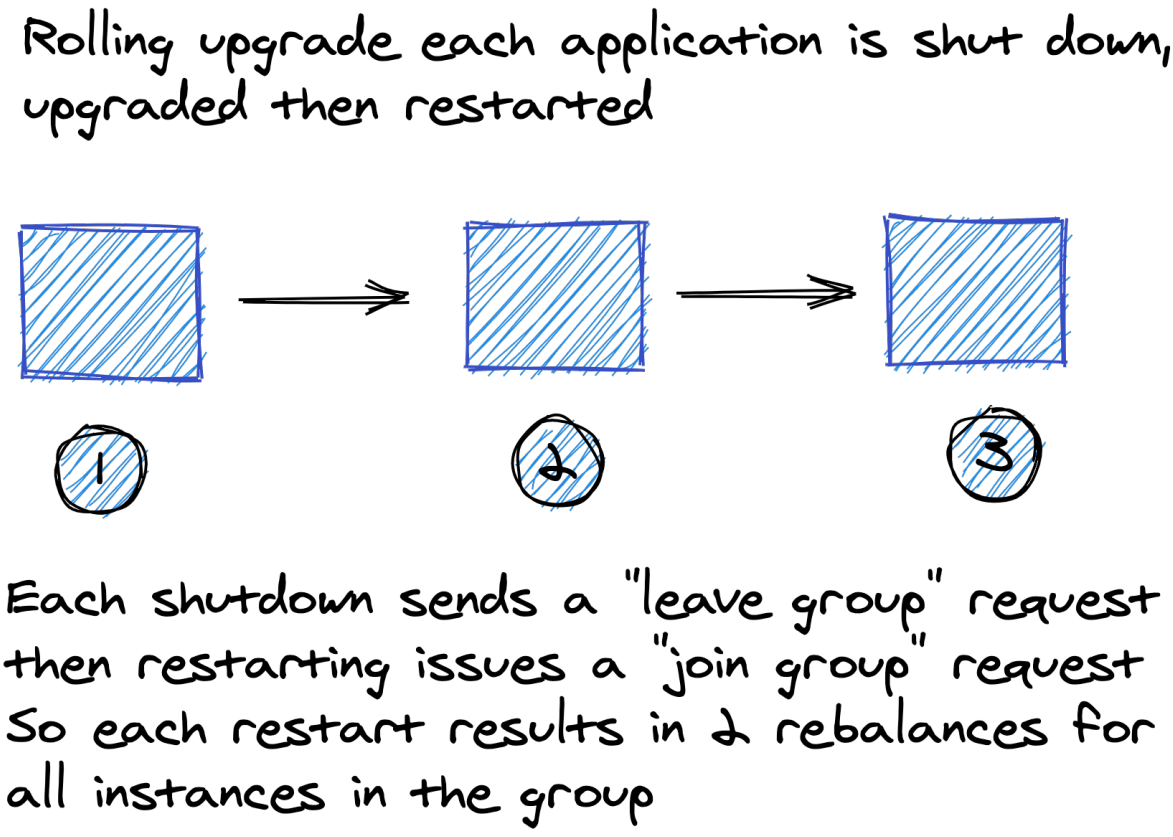

Figure 4.15 Rolling upgrades trigger multiple relabances

You'll stop instance 1, upgrade and restart it, then move on to instance number 2 and so it continues until you've updated every application. By doing a rolling upgrade, you don't lose nearly as much processing time if you shut down every application at the same time. But what happens is this "rolling upgrade", triggers two rebalances for every instance, one when the

application shuts down and another when it starts back up. Or consider a cloud environment where an application node can drop off at any moment only to have it back up an running once its failure is detected.

Even with the improvements brought by cooperative rebalancing, it would be advantageous in these situations to not have a rebalance triggered automatically for these transient actions. The concept of "static membership" was introduced in the 2.3 version of Apache Kafka. We'll use the following illustration to help with our discussion of how static membership works

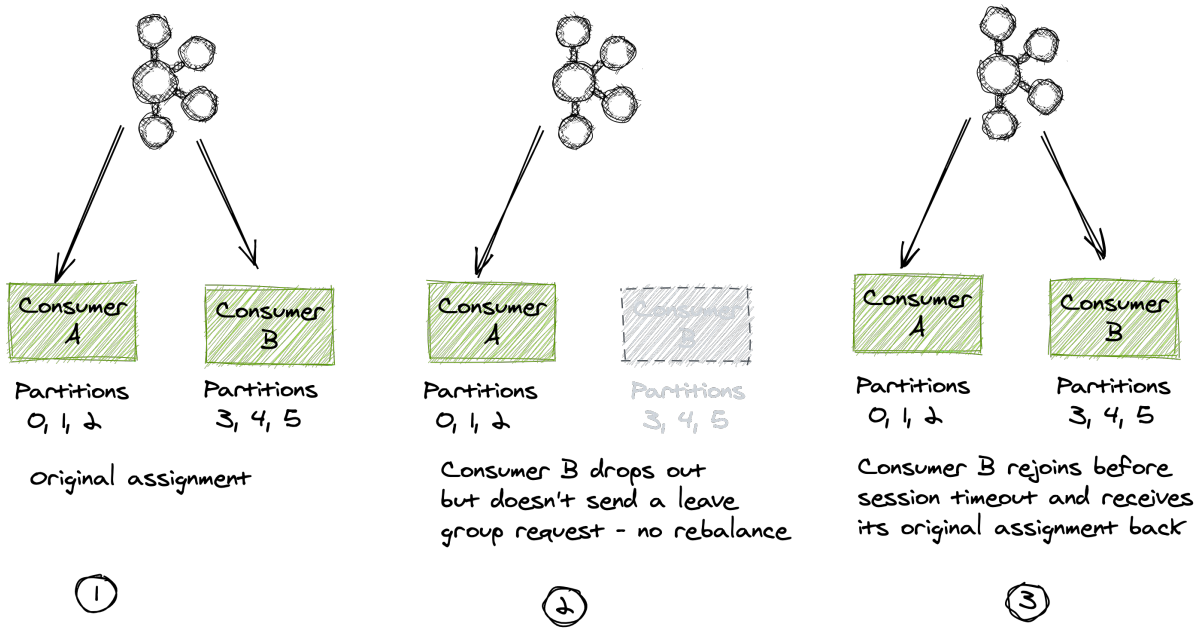

Figure 4.16 Static members don't issue leave group requests when dropping out of a group and a static id allows the controller to remember them

At a high-level with static membership you set a unique id in the consumer configuration, `group.instance.id`. The consumer provides this id to the controller when it joins a group and the controller stores this unique group-id. When a consumer leaves the group, it does not send a leave group request. When it rejoins it presents this unique membership id to the controller. The controller looks it up and can then give back the original assignment to this consumer with no rebalancing involved at all! The trade-off for using static membership is that you'll need to increase the `session.timeout.ms` configuration to a value higher than the default of 10 seconds, as once a session timeout occurs, then the controller kicks the consumer out of the group and triggers a rebalance.

The value you choose should be long enough to account for transient unavailability and not triggering a rebalance but not so long that a true failure gets handled correctly with a rebalance. So if you can sustain ten minutes of partial unavailability then maybe set the session timeout to eight minutes. While static membership can be a good option for those running KafkaConsumer applications in a cloud environment, it's important to take into account the performance

implications before opting to use it. Note that to take advantage of static membership, you must have Kafka brokers and clients on version 2.3.0 or higher.

Next, we'll cover a subject that is very important when using a `KafkaConsumer`, commit the offsets of messages.

4.2.4 Committing offsets

In chapter two, we talked about how the broker assigns a number to incoming records called an offset. The broker increments the offset by one for each incoming record. Offsets are important because they serve to identify the logical position of a record in a topic. A `KafkaConsumer` uses offsets to know where it last consumed a record. For example if a consumer retrieves a batch of records with offsets from 10 to 20, the starting offset of the next batch of records the consumer wants to read starts at offset 21.

To make sure the consumer continues to make progress across restarts for failures, it needs to periodically commit the offset of the last record it has successfully processed. Kafka consumers provide a mechanism for automatic offset commits. You enable automatic offset commits by setting the `enable.auto.commit` configuration to `true`. By default this configuration is turned on, but I've listed it here so we can talk about how automatic commits work. Also, we'll want to discuss the concept of a consumers' position vs. its latest committed offset. There is also a related configuration, `auto.commit.interval.ms` that specifies how much time needs to elapse before the consumer should commit offsets and is based on the system time of the consumer.

But first, lets show how automatic commits work.

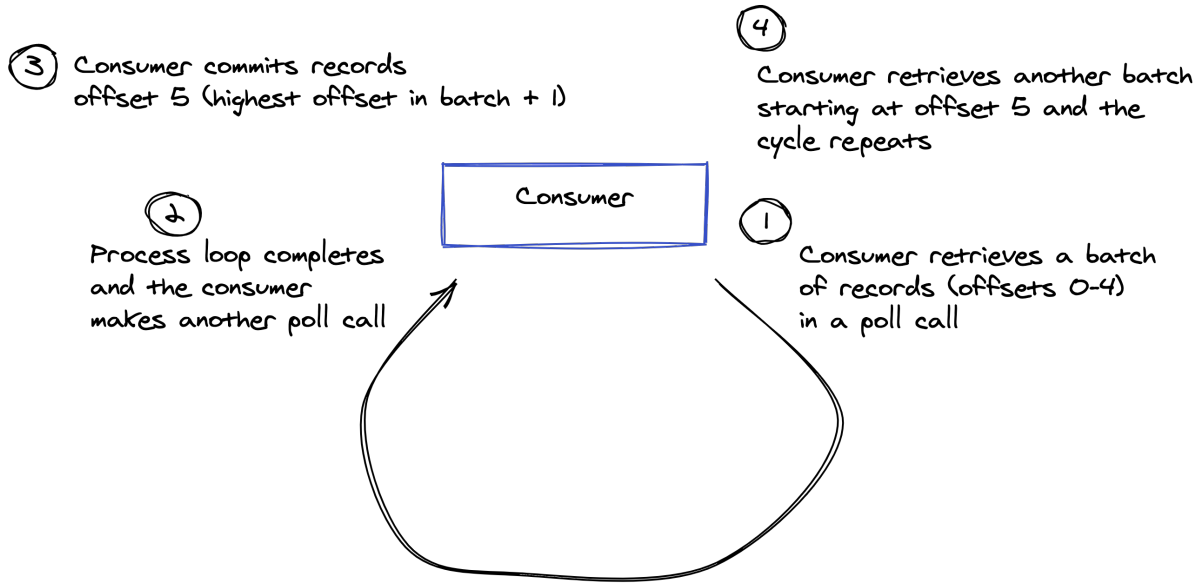

Figure 4.17 With automatic commits enabled when returning to the top of the poll loop the highest offset +1 of the previous batch could be committed if the auto commit interval has passed

Following from the graphic above, the consumer retrieves a batch of records from the `poll(Duration)` call. Next the code takes the `ConsumerRecords` and iterates over them and does some processing of the records. After that the code returns to top of the `poll` loop and attempts to retrieve more records. But before retrieving records, if the consumer has auto-commit enabled and the amount of time elapsed since the last auto-commit check is greater than the `auto.commit.interval.ms` interval, the consumer commits the offsets of the records from the previous batch. By committing the offsets, we are marking these records as consumed, and under normal conditions the consumer won't process these records again. I'll describe what I mean about this statement a little bit later.

What does it mean to commit offsets? Kafka maintains an internal topic named `_offsets` where it stores the committed offsets for consumers. When we say a consumer commits it's not storing the offsets for each record it consumes, it's the highest offset, per partition, plus one that the consumer has consumed so far that's committed.

For example, in the illustration above, let's say the records returned in the batch contained offsets from 0-4. So when the consumer commits, it will be offset 5.

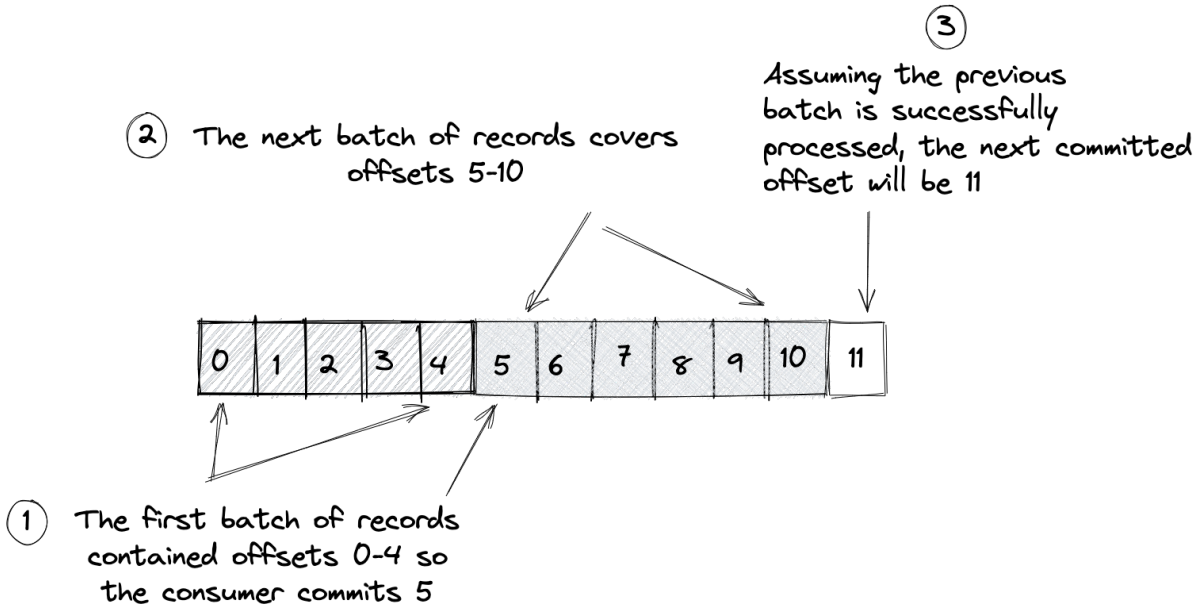

Figure 4.18 A consumers committed position is the largest offset it has consumed so far plus one

So the committed position is offset that has been sucessfully stored, and it indicates the starting record for the next batch it will retrieve. In this illustration it's 5. Should the consumer in this example fail or you restarted the application the consumer would consume records starting at offset 5 again since it wasn't able to commit prior to the failure or restart.

Consuming from the last committed offset means that you are guaranteed to not miss processing a record due to errors or application restarts. But it also means that you may process a record

more than once.

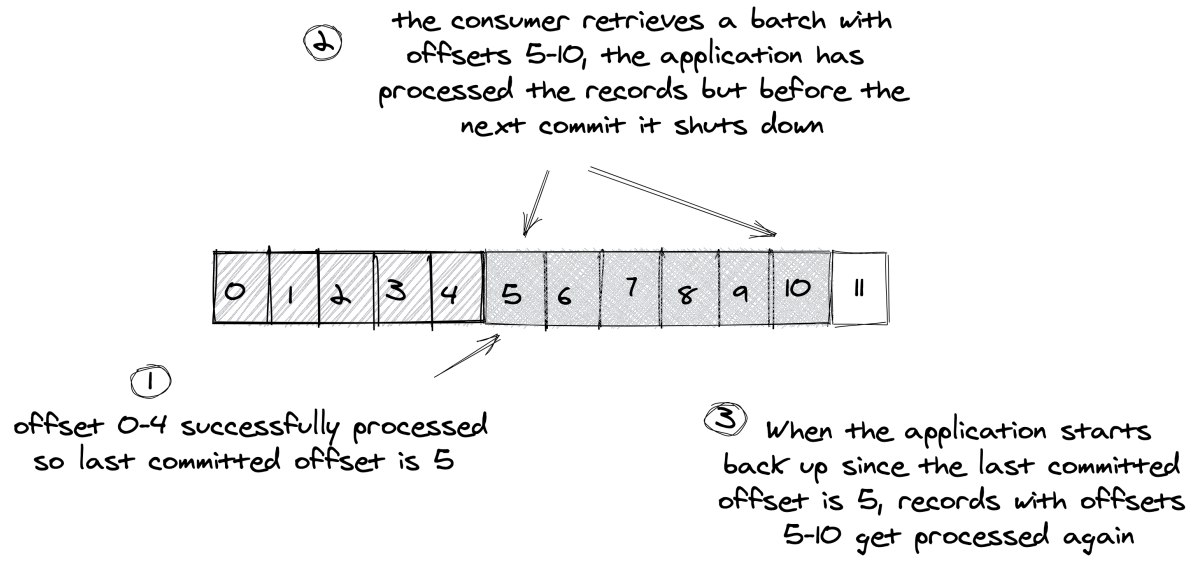

Figure 4.19 Restarting a consumer after processing without a commit means reprocessing some records

If you processed some of the records with offsets larger than the latest one committed, but your consumer failed to commit for whatever reason, this means when you resume processing, you start with records from the committed offset, so you'll reprocess some of the records. This potential for reprocessing is known as at-least-once. We covered at-least-once delivery in the delivery semantics earlier in the chapter.

To avoid reprocessing records you could manually commit offsets immediately after retrieving a batch of records, giving you at-most-once delivery. But you run the risk of losing some records if your consumer should encounter an error after committing and before it's able to process the records. Another option (probably the best), to avoid reprocessing is to use the Kafka transactional API which guarantees exactly-once delivery.

COMMITTING CONSIDERATIONS

When enabling auto-commit with a Kafka consumer, you need to make sure you've fully processed all the retrieved records before the code returns to the top of the poll loop. In practice, this should present no issue assuming you are working with your records synchronously meaning your code waits for the completion of processing of each record. However, if you were to hand off records to another thread for asynchronous processing or set the records aside for later processing, you also run the risk of potentially not processing all consumed records before you commit. Let me explain how this could happen.

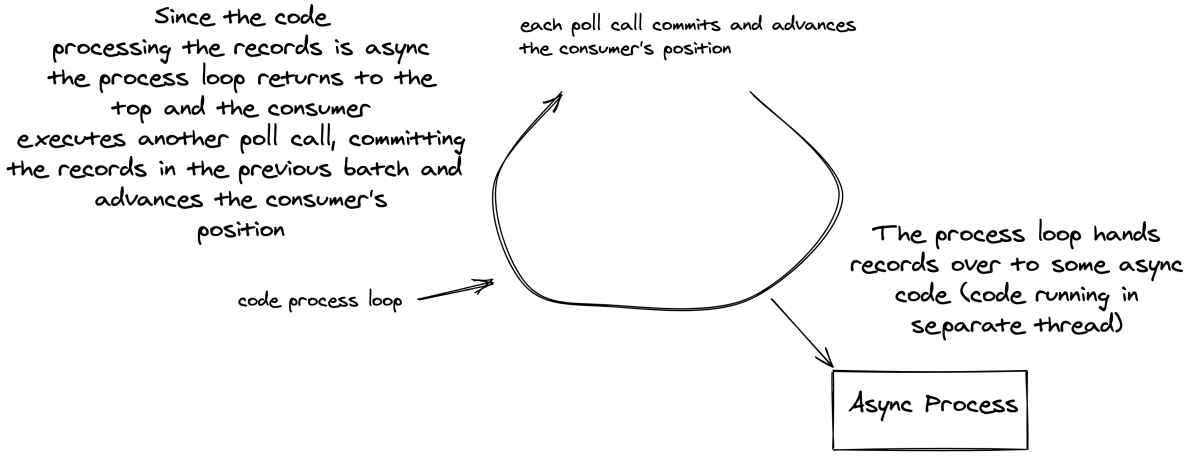

Figure 4.20 Asynchronous processing with auto committing can lead to potentially lost records

When you hand the records off to an asynchronous process, the code in your poll loop won't wait for the successful processing of each record. When your application calls the `poll()` method again, it commits the current position i.e the highest offset + 1 from the for each topic-partition consumed in the previous batch. But your async process may not completed working with all the records up to the highest offset at the time of the commit. If your consumer application experienced a failure or a shutdown for any reason, when it resumes processing, it will start from the last committed offset, which skips over the un-processed records in the last run of your application.

To avoid prematurely-maturely committing records before you consider them fully processed, then you'll want to disable auto-commits by setting `enable.auto.commit` to `false`.

But why would you need to use asynchronous processing requiring manually committing? Let's say when you consume records, you do some processing that takes long time (up to 1 second) to process each record. The topic you consume from has a high volume of traffic, so you don't want to fall behind. So you decide that as soon as you consume a batch of records, you'll hand them off to an async process so the consumer can immediately return to the poll call to retrieve the next batch.

Using an approach like this is called pipeling. But you'll need make sure you're only committing the offsets for records that have been successfully processed, which means turning off auto-committing and coming up with a way to commit only records that your application considers fully processed. The following example code shows one example approach you could take. Note that I'm only showing the key details here and you should consult the source code to see the entire example

Listing 4.4 Consumer code found in bbejeck.chapter_4.pipelining.PipliningConsumerClient

```
// Details left out for clarity
ConsumerRecords<String, ProductTransaction> consumerRecords = consumer.poll(
  Duration.ofSeconds(5));
if (!consumerRecords.isEmpty()) {
    recordProcessor.processRecords(consumerRecords);       ❶
    Map<TopicPartition, OffsetAndMetadata> offsetsAndMetadata =
      recordProcessor.getOffsets();   ❷
    if (offsetsAndMetadata != null) {
        consumer.commitSync(offsetsAndMetadata);   ❸
    }
```

❶ After you've retrieved a batch of records you hand off the batch of records to the async processor.

❷ Checking for offsets of completed records

❸ If the `Map` is not empty, you commit the offsets of the records processed so far.

The key point with this consumer code is that the `RecordProcessor.processRecords()` call returns immediately, so the next call to `RecordProcessor.getOffsets()` returns offsets from a previous batch of records that are fully processed. What I want to emphasize here is how the code hands over new records for processing then collects the offsets of records already fully processed for committing. Let's take a look at the processor code to see this is done:

Listing 4.5 Asynchronous processor code found in bbejeck.chapter_4.piplining.ConcurrentRecordProcessor

```
Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();   ❶
consumerRecords.partitions().forEach(topicPartition -> {    ❷
   List<ConsumerRecord<String,ProductTransaction>> topicPartitionRecords =
  consumerRecords.records(topicPartition);   ❸
   topicPartitionRecords.forEach(this::doProcessRecord);   ❹
   long lastOffset = topicPartitionRecords.get(
  topicPartitionRecords.size() - 1).offset();   ❺
   offsets.put(topicPartition, new OffsetAndMetadata(lastOffset + 1));   ❻
  });
 ....
 offsetQueue.offer(offsets);   ❼
```

❶ Creating the `Map` for collecting the offset for committing

❷ Iterating over the `TopicPartition` objects

❸ Getting records by `TopicPartition` for processing

❹ Doing the actual work on the consumed records

❺ Getting the last offset for all records of a given `TopicPartition`

❻ Storing the offset to commit for the `TopicPartition`

❼ Putting the entire `Map` of offsets in a queue.

The the takeaway with the code here is that by iterating over records by `TopicPartition` it's

easy to create the map entry for the offsets to commit. Once you've iterated over all the records in the list, you only need to get the last offset. You, the observant read might be asking yourself "Why does the code add1 to the last offset?" When committing offsets it's always the offset of the ***next*** record you'll retrieve. For example if the last offset is 5, you want to commit 6. Since you've already consumed 0-5 you're only interested in consuming records from offset 6 forward.

Then you simply use the `TopicPartition` from the top of the loop as the key and the `OffsetAndMetadata` object as the value. When the consumer retrieves the offsets from the queue, it's safe to commit those offsets as the records have been fully processed. The main point to this example is how you can ensure that you only commit records you consider "complete" if you need to asynchronously process records outside of the `Consumer.poll` loop. It's important to note that this approach only uses a **single thread** and consumer for the record processing which means the code still processes the records in order, so it's safe to commit the offsets as they are handed back.

NOTE	**For a fuller example of threading and the `KafkaConsumer` you should consult www.confluent.io/blog/introducing-confluent-parallel-message-processing-client/ and github.com/confluentinc/parallel-consumer.**

WHEN OFFSETS AREN'T FOUND

I mentioned earlier that Kafka stores offsets in an internal topic named `_offsets`. But what happens when a consumer can't find its offsets? Take the case of starting a new consumer against an existing topic. The new `group.id` will not have any commits associated with it. So the question becomes where to start consuming if offsets aren't found for a given consumer? The `KafkaConsumer` provides a configuration, `offset.reset.policy` which allows you to specify a relative position to start consuming in the case there's no offsets available for a consumer.

There are three settings:

1. earliest - reset the offset to the earliest one
2. latest - reset the offset to the latest one
3. none - throw an exception to the consumer

With a setting of `earliest` the implications are that you'll start processing from the head of the topic, meaning you'll see all the records currently available. Using a setting of `latest` means you'll only start receiving records that arrive at the topic once your consumer is online, skipping all the previous records currently in the topic. The setting of `none` means that an exception gets thrown to the consumer and depending if you are using any try/catch blocks your consumer may shut down.

The choice of which setting to use depends entirely on your use case. It may be that once a

consumer starts you only care about reading the latest data or it may be too costly to process all records.

Whew! That was quite a detour, but well worth the effort to learn some of the critical aspects of working with the `KafkaConsumer`.

So far we've covered how to build streaming applications using a `KafkaProducer` and `KafkaConsumer`. What's been discussed is good for those situations where your needs are met with *at-least-once processing*. But there are situations where you need to guarantee that you process records *exactly once*. For this functionality you'll want to consider using the **exactly once** semantics offered by Kafka.

4.3 Exactly once delivery in Kafka

The 0.11 release of Apache Kafka saw the `KafkaProducer` introduce exactly once message delivery. There are two modes for the `KafkaProducer` to deliver exactly once message semantics; the idempotent producer and the transactional producer.

> **NOTE** **Idempotence means you can perform an operation multiple times and the result won't change beyond what it was after the first application of the operation.**

The idempotent producer guarantees that the producer will deliver messages in-order and only once to a topic-partition. The transactional producer allows you to produce messages to multiple topics atomically, meaning all messages across all topics succeed together or none at all. In the following sections, we'll discuss the idempotent and the transactional producer.

4.3.1 Idempotent producer

To use the idempotent producer you only need to set the configuration `enable.idempotence=true`. There are some other configuration factors that come into play:

1. `max.in.flight.requests.per.connection` must not exceed a value of 5 (the default value is 5)
2. `retries` must be greater than 0 (the default value is Integer.MAX_VALUE)
3. `acks` must be set to `all`. If you do not specify a value for the `acks` configuration the producer will update to use the value of `all`, otherwise the producer throws a `ConfigException`.

Listing 4.6 KafkaProducer configured for idempotence

```
// Several details omitted for clarity
Map<String, Object> producerProps = new HashMap<>();
//Standard configs
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "somehost:9092");
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, ...);
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ...);

//Configs related to idempotence
producerProps.put(ProducerConfig.ACKS_CONFIG, "all");  ❶
producerProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);  ❷
producerProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);  ❸
producerProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);  ❹
```

❶ Setting acks to "all"

❷ Enabling idempotence

❸ Setting retries to Integer.MAX_VALUE - this is the default value shown here for completeness

❹ Setting max in flight requests per connection to 5 - this is the default value shown here for completeness

If you recall from our earlier discussion about the `KafkaProducer` we outlined a situation where due to errors and retries record batches within a partition can end up out of order. To avoid that situation, it was suggested to set the `max.infligh.requests.per.connection` to one. Using the idempotent producer removes the need for you to adjust that configuration. We also discussed in the message delivery semantics to avoid possible record duplication, you would need to set retries to zero risking possible data loss.

Using the idempotent producer avoids both of the records-out-of-order and possible-record-duplication-with-retries. If you requirements are for strict ordering within a partition and no duplicated deliver of records then using the idempotent producer is a must.

NOTE **As of the 3.0 release of Apache Kafka the idempotent producer settings are the default so you'll get the benefits of using it out of the box with no additional configuration needed.**

The idempotent producer uses two concepts to achieve its in-order and only-once semantics-unique producer ids and sequence numbers for messages. The idempotent producer gets initiated with a unique producer id (PID). Since each creation of a idempotent producer results in a new PID, idempotence for a producer is only guaranteed during a single producer session. For a given PID a monotonically sequence id (starting at 0) gets assigned to each batch of messages. There is a sequence number for each partition the producer sends records to.

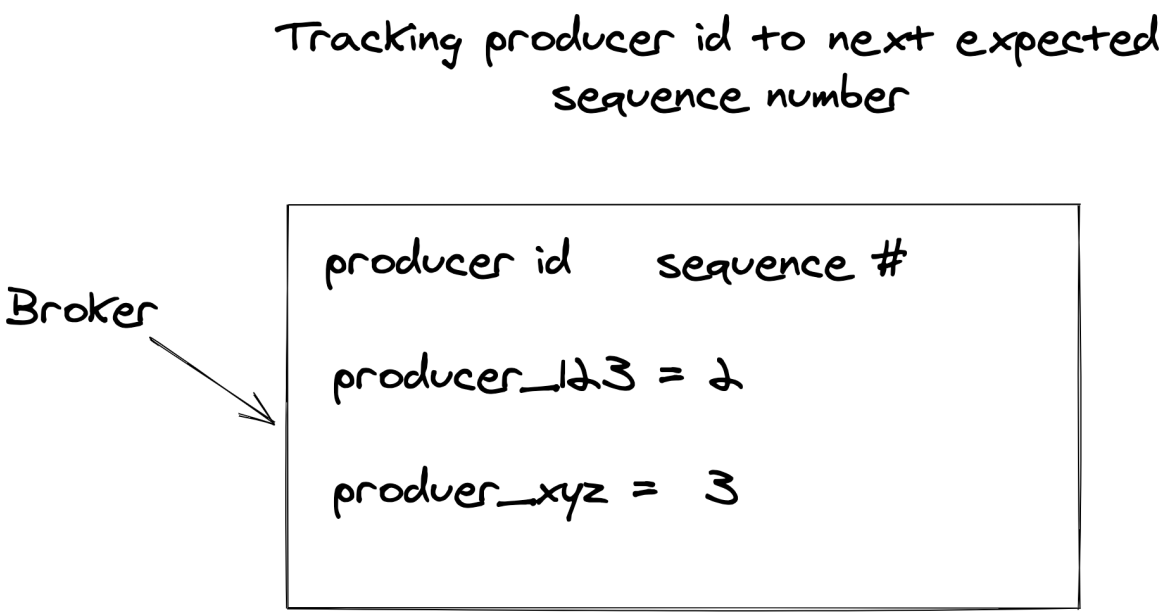

Figure 4.21 The broker keeps track of sequence numbers for each PID and topic-partition it receives

The broker maintains a listing (in-memory) of sequence numbers per topic-partition per PID. If the broker receives a sequence number not ***exactly one greater*** than the sequence number of the last committed record for the given PID and topic-partition, it will reject the produce request.

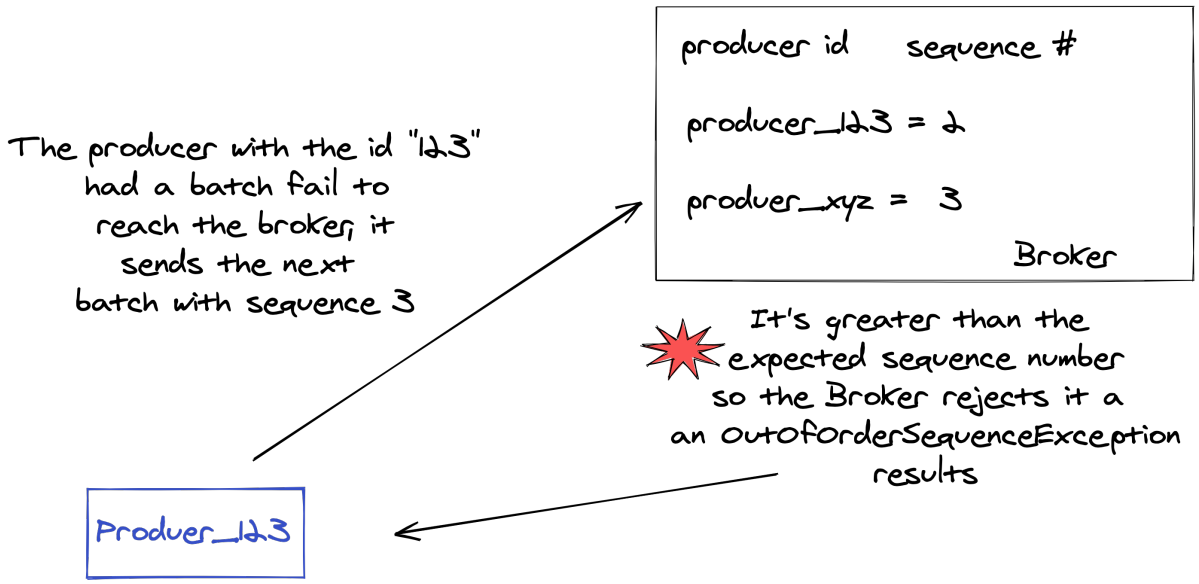

Figure 4.22 The broker rejects produce requests when the message sequence number doesn't match expected one

If the number is less than the expected sequence number, it's a duplication error which the producer ignores. If the number is higher than expected the produce request results in a `OutOfOrderSequenceException`. For the idempotent producer, the

`OutOfOrderSequenceException` is not fatal error and retries will continue. Essentially when there is a retryable error, if there are more than 1 in-flight requests, the broker will reject the subsequent requests and the producer will put them back in order to resend them to the broker.

So if you require strict ordering of records within a partition, then using the idempotent producer is a must. But what do you do if you need to write to multiple topic-partitions atomically? In that case you would opt to use the transactional producer which we'll cover next.

4.3.2 Transactional producer

Using the transactional producer allows you to write to multiple topic-partitions atomically; all of the writes succeed or none of them do. When would you want to use the transactional producer? In any scenario where you can't afford to have duplicate records, like in the financial industry for example.

To use the transaction producer, you need to set the producer configuration `transactional.id` to a unique value for the producer. Kafka brokers use the `transactional.id` to enable transaction recovery across multiple sessions from the same producer instance. Since the id needs to be unique for each producer and applications can have multiple producers, it's a good idea to come up with a strategy where the id for the producers represents the segment of the application its working on.

> **NOTE** **Kafka transaction are a deep subject and could take up an entire chapter on its own. For that reason I'm not going to go into details about the design of transactions. For readers interested in more details here's a link to the original KIP (KIP stands for Kafka Improvement Process) cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging#KIP98ExactlyOnceDeliveryandTransactionalMessaging-Brokerconfigs**

When you enable a producer to use transactions, it is automatically upgraded to an idempotent producer. You can use the idempotent producer without transactions, but you can't do the opposite, using transactions without the idempotent producer. Let's dive into an example. We'll take our previous code and make it transactional

Listing 4.7 KafkaProducer basics for transactions

```
HashMap<String, Object> producerProps = new HashMap<>();

producerProps.put("transactional.id", "set-a-unique-transactional-id");   ❶

Producer<String, String> producer = new KafkaProducer<>(producerProps);
producer.initTransactions();   ❶

try {
    producer.beginTransaction();   ❷
    producer.send(topic, "key", "value");   ❸
    producer.commitTransaction();     ❹
} catch (ProducerFencedException | OutOfOrderSequenceException
  | AuthorizationException e) {    ❺
      producer.close();
} catch (KafkaException e) {       ❻
     producer.abortTransaction();
     // safe to retry at this point   ❼
}
```

❶ Setting a unique id for the producer. Note that it's up to the user to provide this unique id.

❷ Calling initTransactions

❸ The beginning of the transaction, but does not start the clock for transaction timeouts

❹ Sending record(s), in practice probably you'd probably send more than one record but it's shortened here for clarity

❺ Committing the transaction after sending all the records

❻ Handling fatal exceptions, your only choice at this point is to close the producer and re-instantiate the producer instance

❼ Handling a non-fatal exception, you can begin a new transaction with the same producer and try again

After creating a transactional producer instance is to first thing you must here is execute the `initTransactions()` method. The `initTransaction` sends a message to the transaction coordinator (the transaction coordinator is a broker managing transactions for producers) so it can register the `transactional.id` for the producer to manage its transactions. The transaction coordinator is a broker managing transactions for producers.

If the previous transaction has started, but not finished, then this method blocks until its completed. Internally, it also retrieves some metadata including something called an `epoch` which this producer uses in future transactional operations.

Before you start sending records you call `beginTransaction()`, which starts the transaction for the producer. Once the transaction starts,The transaction coordinator will only wait for a period of time defined by the `transaction.timeout.ms` (one minute by default) and it without an update (a commit or abort) it will proactively abort the transaction. But the transaction

coordinator does not start the clock for transaction timeouts until the broker starts sending records. Then after the code completes processing and producing the records, you commit the transaction.

You should notice a subtle difference in error handling between the transactional example from the previous non-transactional one. With the transactional produce you don't have to check of an error occurred either with a `Callback` or checking the returned `Future`. Instead the transactional producer throws them directly for your code to handle.

It's important to note than with any of the exceptions in the first `catch` block are fatal and you must close the producer and to continue working you'll have to create a new instance. But any other exception is considered re-tryable and you just need to abort the current transaction and start over.

Of the fatal exceptions, we've already discussed the `OutOfOrderSequenceException` in the idempotent producer section and the `AuthorizationException` is self explanatory. Be we should quickly discuss the `ProducerFencedException`. Kafka has a strict requirement that there is only one producer instance with a given `transactional.id`. When a new transactional producer starts, it "fences" off any previous producer with the same id must close. However, there is another scenario where you can get a `ProducerFencedException` with out starting a new producer with the same id.

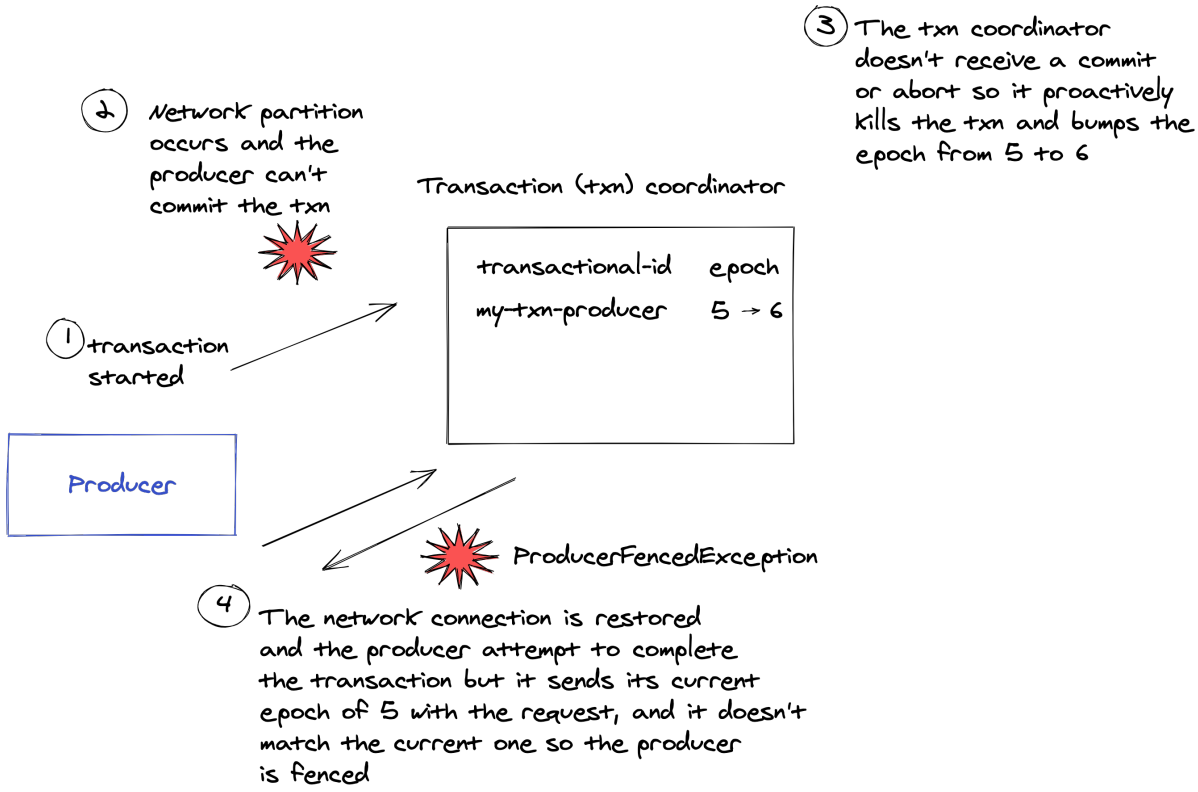

Figure 4.23 Transactions proactively aborted by the Transaction Coordinator cause an increase in the epoch associated with the transaction id

When you execute the `producer.initTransactions()` method, the transaction coordinator increments the producer epoch. The producer epoch is a number the transaction coordinator associates with the transactional id. When the producer makes any transactional request, it provides the epoch along with its transaction id. If the epoch in the request doesn't match the current epoch the transaction coordinator rejects the request and the producer is fenced.

But if the current producer can't communicate with the transaction coordinator for any reason and the timeout expires, as we discussed before, the coordinator proactively aborts the transaction and increments the epoch for that id. When the producer attempts to work again after the break in communication, it finds itself fenced and you must close the producer and restart at that point.

NOTE **There is example code for transactional producers in the form of a test located at src/test/java/bbejeck/chapter_4/TransactionalProducerConsumerTest.java in the source code.**

So far, I've only covered how to produce transactional records, so let's move on consuming them.

4.3.3 Consumers in transactions

Kafka consumers can subscribe to multiple topics at one time, with some of them containing transactional records and others not. But for transactional records, you'll only want to consume ones that have been successfully committed. Fortunately, it's only a matter of a simple configuration. To configure your consumers for transactional records you set `isolation.level` configuration to `read_committed`.

Listing 4.8 KafkaConsumer configuration for transactions

```
// Several details omitted for clarity

HashMap<String, Object> consumerProps = new HashMap<>();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "the-group-id");

consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");   ❶

consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer.class);
```

❶ Setting the isolation configuration for the consumer

With this configuration set, your consumer is guaranteed to only retrieve successfully committed transaction records. If you use the `read_uncommitted` setting, then the consumer will retrieve both successful and aborted transactional records. The consumer is guaranteed to retrieve

non-transactional records with either configuration set.

There is difference in highest offset a consumer can retrieve in the `read_committed` mode.

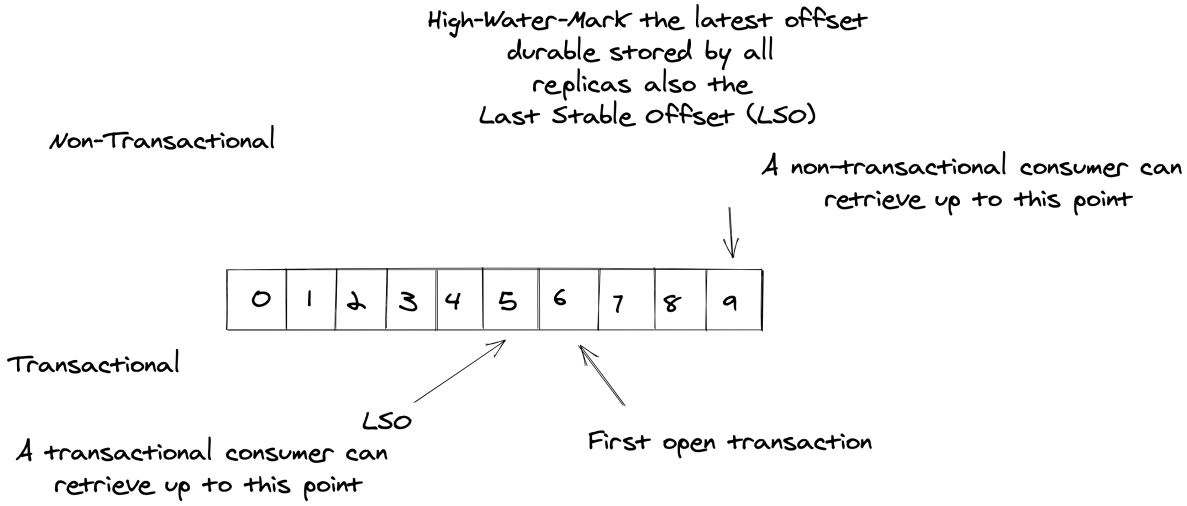

Figure 4.24 High water mark vs. last stable offset in a transactional environment

In Kafka there is a concept of the last stable offset (LSO) which is an offset where all offsets below it have been "decided". There's another concept known as the high water mark. The high water mark is the largest offset successfully written to all replicas. In a non-transactional environment, the LSO is the same as the high water mark as records are considered decided or durable written immediately. But with transactions, an offset can't be considered decided until the transaction is either committed or aborted, so this means the LSO is the offset of the first open transaction minus 1.

This a non-transactional environment, the consumer can retrieve up the the high water mark in a `poll()` call. But with transactions it will only retrieve up to the LSO.

NOTE **The test located src/test/java/bbejeck/chapter_4/TransactionalProducerConsumerTest.java also contains a couple of tests demonstrating consumer behavior with both `read_committed` read `read_uncommitted` configuration.**

So far we've covered how to use a producer and a consumer separately. But there's one more case to consider and that is using a consumer and producer together within a transaction.

4.3.4 Producers and consumers within a transaction

When building applications to work with Kafka it's a fairly common practice to consume records from a topic, perform some type of transformation on the records, then produce those transformed records back to Kafka in a different topic. Records are considered consumed when the consumer commits the offsets. If you recall, committing offsets is simply writing to a topic (_offsets).

So if you are doing a consume - transform - produce cycle, you'd want to make sure that committing offsets is part of the transaction as well. Otherwise you could end up in a situation where you've committed offsets for consumed records, but transaction fails and restarting the application skips the recently processed records as the consumer committed the offsets.

Imagine you have a stock reporting application and you need to provide broker compliance reporting. It's very important that the compliance reports are sent only once so you decide that the best approach is to consume the stock transactions and build the compliance reports within a transaction. This way you are guaranteed that your reports are sent only once.

Listing 4.9 Example of the consume-transform-produce with transactions found in src/test/java/chapter_4/TransactionalConsumeTransformProduceTest.java

```
// Note that details are left out here for clarity

Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();   ❶
producer.beginTransaction();                                        ❷
consumerRecords.partitions().forEach(topicPartition -> {
    consumerRecords.records(topicPartition).forEach(record -> {
        lastOffset.set(record.offset());
        StockTransaction stockTransaction = record.value();
        BrokerSummary brokerSummary = BrokerSummary.newBuilder()    ❸


        producer.send(new ProducerRecord<>(outputTopic, brokerSummary));
    });
    offsets.put(topicPartition,
      new OffsetAndMetadata(lastOffset.get() + 1L));                ❹
});
try {
    producer.sendOffsetsToTransaction(offsets,
      consumer.groupMetadata());                                    ❺
    producer.commitTransaction();                                   ❻
 }
```

❶ Creating the HashMap to hold the offsets to commit

❷ Starting the transaction

❸ Transforming the StockTransaction object into a BrokerSummary

❹ Storing the `TopicPartition` and `OffsetAndMetadata` in the map

❺ Committing the offsets for the consumed records in the transaction

❻ Committing the transaction

From looking at code above, the biggest difference from a non-transactional consume-transform-produce application is that we keep track of the `TopicPartition` objects and the offset of the records. We do this because we need to provide the offsets of the records we just processed to the `KafkaProducer.setOffsetsToTransaction` method. In consume-transform-produce applications with transactions, it's the producer that sends offsets to the consumer group coordinator, ensuring that the offsets are part of the transaction. Should the transaction fail or get aborted, then the offsets are not committed. By having the producer commit the offsets, you don't need any coordination between the producer and consumer in the cases of rolled-back transactions.

So far we've covered using producer and consumer clients for sending and receiving records to and from a Kafka topic. But there's another type of client which uses the `Admin` API and it allows you to perform topic and consumer group related administrative functions programmatically.

4.4 Using the Admin API for programmatic topic management

Kafka provides an administrative client for inspecting topics, broker, ACLs (Access Control Lists) and configuration. While there are several functions you can use the admin client, I'm going to focus on the administrative functions for working with topics and records. The reason I'm doing this is I'm presenting what I feel are the use cases most developers will see in ***development*** of their applications. Most of the time, you'll have a operations team responsible for the management of your Kafka brokers in production. What I'm presenting here are things you can do to facilitate testing a prototyping an application using Kafka.

4.4.1 Working with topics programmatically

To create topics with the admin client is simply a matter of creating the admin client instance and then executing the command to create the topic(s).

Listing 4.10 Creating a topic

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");

try (Admin adminClient = Admin.create(adminProps)) {   ❶

     final List<NewTopic> topics = new ArrayList<>)();    ❷

    topics.add(new NewTopic("topic-one", 1, 1));   ❸
    topics.add(new NewTopic("topic-two", 1, 1));

    adminClient.createTopics(topics);   ❹
}
```

❶ Creating the Admin instance, note the use of a try with resources block

❷ The list to hold the `NewTopic` objects

❸ Creating the `NewTopic` objects and adding them to the list

❹ Executing the command to create the topics

> **NOTE** **I'm referring to an admin client but the type is the interface `Admin`. There is an abstract class `AdminClient`, but it's use is discouraged over using the `Admin` interface instead. An upcoming release may remove the `AdminClient` class.**

This code can be especially useful when you are prototyping building new applications by ensuring the topics exist before running the code. Let's expand this example some and show how you can list topics and optionally delete one as well.

Listing 4.11 More topic operations

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");

try (Admin adminClient = Admin.create(adminProps)) {

     Set<String> topicNames = adminClient.listTopics().names.get();   ❶
     System.out.println(topicNames);   ❷
     adminClient.deleteTopics(Collections.singletonList("topic-two"));   ❸
}
```

❶ In this example you're listing all the non-internal topics in the cluster. Note that if you wanted to include the internal topics you would provide a `ListTopicOptions` object where you would call the `ListTopicOptions.listInternal(true)` method.

❷ Printing the current topics found

❸ You delete a topic and list all of the topics again, but you should not see the recently deleted topic in the list.

An additional note for annotation one above, is that the `Admin.listTopics()` returns a `ListTopicResult` object. To get the topic names you use the `ListTopicResult.names()` which returns a `KafkaFuture<Set<String>>` so you use use the `get()` method which blocks until the admin client request completes. Since we're using a broker container running on your local machine, chances are this command completes immediately.

There are several other methods you can execute with the admin client such as deleting records and describing topics. But the way you execute them is very similar, so I wont list them here, but look at the source code (src/test/java/bbejeck/chapter_4/AdminClientTest.java) to see more examples of using the admin client.

TIP	Since we're working on a Kafka broker running in a docker container on your local machine, we can execute all the admin client topic and record operations risk free. But you should exercise caution if you are working in a shared environment to make sure you don't create issues for other developers. Additionally, keep in mind you might not have the opportunity to use the admin client commands in your work environment. And I should stress that you should never attempt to modify topics on the fly in production environments.

That wraps up our coverage of using the admin API. In our next and final section we'll talk about the considerations you take into account for those times when you want to produce multiple event types to a topic.

4.5 Handling multiple event types in a single topic

Let's say you've building an application to track activity on commerce web site. You need to track the click-stream events such as logins and searches and any purchases. Conventional wisdom says that the different events (logins, searches) and purchases could go into separate topics as they are separate events. But there's information you can gain from examining how these related events occurred in sequence.

But you'll need to consume the records from the different topics then try and stitch the records together in proper order. Remember, Kafka guarantees record order within a partition of a topic, but not across partitions of the same topic not to mention partitions of other topics.

Is there another approach you can take? The answer is yes, you can produce those different event types to the same topic. Assuming you providing a consistent key across the event types, you are going receive the different events in-order, on the same topic-partition.

At the end of chapter three (Schema Registry), I covered how you can use multiple event types in a topic, but I deferred on showing an example with producers and consumers. Now we'll go through an example now on how you can produce multiple event types and consume multiple event types safely with Schema Registry.

In chapter three, specifically the `Schema references and multiple events per topic` section I discussed how you can use Schema Registry to support multiple event types in a single topic. I didn't go through an example using a producer or consumer at that point, as I think it fits better in this chapter. So that's what we're going to cover now.

NOTE	Since chapter three covered Schema Registry, I'm not going to do any review in this section. I may mention some terms introduced in chapter three, so you may need to refer back to refresh your memory if needed.

Let's start with the producer side.

4.5.1 Producing multiple event types

We'll use this Protobuf schema in this example:

```
{
syntax = "proto3";

package bbejeck.chapter_4.proto;

import "purchase_event.proto";
import "login_event.proto";
import "search_event.proto";

option java_outer_classname = "EventsProto";

message Events {
      oneof type {
        PurchaseEvent purchase_event = 1;
        LogInEvent login_event = 2;
        SearchEvent search_event = 3;
      }
      string key = 4;
    }
}
```

What happens when you generate the code from the protobuf definition you get a `EventsProto.Events` object that contains a single field `type` that accepts one of the possible three event objects (a Protobuf `oneof` field).

Listing 4.12 Example of creating KafkaProducer using Protobuf with a oneof field

```
// Details left out for clarity
...
producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaProtobufSerializer.class);   ❶
...

Producer<String, EventsProto.Events> producer = new KafkaProducer<>(producerConfigs));   ❷
```

❶ Configure the producer to use the Protobuf serializer

❷ Creating the `KafkaProducer` instance

Since Protobuf doesn't allow the `oneof` field as a top level element, the events you produce always have an outer class container. As a result your producer code doesn't look any different for the case when you're sending a single event type. So the generic type for the `KafkaProducer`

and `ProducerRecord` is the class of the Protobuf outer class, `EventsProto.Events` in this case.

In contrast, if you were to use an Avro union for the schema like this example here:

Listing 4.13 Avro schema of a union type

```
[
  "bbejeck.chapter_3.avro.TruckEvent",
  "bbejeck.chapter_3.avro.PlaneEvent",
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

Your producer code will change to use a common interface type of all generated Avro classes:

Listing 4.14 KafkaProducer instantiation with Avro union type schema

```
//Some details left out for clarity

producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaAvroSerializer.class);   ❶
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_SCHEMAS,
  false);   ❷
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VERSION,
  true);   ❸


Producer<String, SpecificRecord> producer = new KafkaProducer<>(
  producerConfigs())   ❹
```

❶ Specifying to use the Kafka Avro serializer

❷ Configuring to producer to not auto register schemas

❸ Setting the use latest schema version to true

❹ Instantiating the producer

Because you don't have an outer class in this case each event in the schema is a concrete class of either a `TruckEvent`, `PlaneEvent`, or a `DeliveryEvent`. To satisfy the generics of the `KafkaProducer` you need to use the `SpecificRecord` interface as every Avro generated class implements it. As we covered in chapter three, it's crucial when using Avro schema references with a union as the top-level entry is to disable auto-registration of schemas (annotation two above) and to enable using the latest schema version (annotation three).

Now let's move to the other side of the equation, consuming multiple event types.

4.5.2 Consuming multiple event types

When consuming from a topic with multiple event types, depending how your approach, you may need to instantiate the `KafkaConsumer` with a generic type of a common base class or interface that all of the records implement.

Let's consider using Protobuf first. Since you will always have an outer wrapper class, that's the class you'll use in the generic type parameter, the value parameter in this example.

Listing 4.15 Configuring the consumer for working with multiple event types in Protobuf

```
//Other configurations details left out for clarity

consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaProtobufDeserializer.class);   ❶
consumerProps.put(
  KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
    EventsProto.Events.class);   ❷

Consumer<EventsProto.Events> consumer = new KafkaConsumer<>(
  consumerProps);   ❸
```

❶ Using Protobuf deserializer

❷ Setting the Protobuf deserializer to return a specific type

❸ Creating the `KafkaConsumer`

You are setting up your consumer as you've seen before; you're configuring the deserializer to return a specific type, which is the `EventsProto.Events` class in this case. With Protobuf, when you have a `oneof` field, the generated Java code includes methods to help you determine the type of the field with `hasXXX` methods. In our case the `EventsProto.Events` object contains the following 3 methods:

```
hasSearchEvent()
hasPurchaseEvent()
hasLoginEvent()
```

The protobuf generated Java code also contains an enum named `<oneof field name>Case`. In this example, we've named the `oneof` field `type` so it's named `TypeCase` and you access by calling `EventsProto.Events.getTypeCase()`. You can use the enum to determine the underlying object succinctly:

```
//Details left out for clarity
switch (event.getTypeCase()) {
    case LOGIN_EVENT -> {     ❶
        logins.add(event.getLoginEvent());   ❷
    }
    case SEARCH_EVENT -> {
        searches.add(event.getSearchEvent());
    }
    case PURCHASE_EVENT ->  {
        purchases.add(event.getPurchaseEvent());
    }
}
```

❶ Individual case statement base on the enum

❷ Retrieving the event object using `getXXX` methods for each potential type in the `oneof` field

Which approach you use for determining the type is a matter of personal choice.

Next let's see how you would set up your consumer for multiple types with the Avro union schema:

Listing 4.16 Configuring the consumer for working with union schema with Avro

```
//Other configurations details left out for clarity

consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaAvroDeserializer.class);   ❶
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
  true);   ❷

Consumer<SpecificRecord> consumer = new KafkaConsumer<>(consumerProps);   ❸
```

❶ Using Avro deserializer

❷ Specifying the deserializer to return a specific Avro type

❸ Creating the `KafkaConsumer`

As you've seen before you specify the `KafkaAvroDeserializer` for the deserializer configuration. We also covered before how Avro is slightly different from Protobuf and JSON Schema in that you tell it to return the specific class type, but you don't provide the class name. So when you have multiple event types in a topic and you are using Avro, the consumer needs to use the `SpecificRecord` interface again in the generics shown in annotation three.

So by using the `SpecificRecord` interface when you start retrieving records from the `Consumer.poll` call you'll need to determine the concrete type to do any work with it.

Listing 4.17 Determining the concrete type of a record returned from a consumer with Avro union schemas

```
// Details left out for clarity

SpecificRecord avroRecord = record.value();
if (avroRecord instanceof PlaneEvent) {
    PlaneEvent planeEvent = (PlaneEvent) avroRecord;
    ....
} else if (avroRecord instanceof TruckEvent) {
    TruckEvent truckEvent = (TruckEvent) avroRecord;
    ....
} else if (avroRecord instanceof DeliveryEvent) {
    DeliveryEvent deliveryEvent = (DeliveryEvent) avroRecord;
   ....
}
```

The approach here is similar to that of what you did with Protobuf but this is at the class level instead of the field level. You could also choose to model your Avro approach to something similar of Protobuf and define record that contains a field representing the union. Here's an example:

Listing 4.18 Avro with embedding the union field in a record

```
{
  "type": "record",
  "namespace": "bbejeck.chapter_4.avro",
  "name": "TransportationEvent",   ❶

  "fields" : [
    {"name": "txn_type", "type": [    ❷
      "bbejeck.chapter_4.avro.TruckEvent",
      "bbejeck.chapter_4.avro.PlaneEvent",
      "bbejeck.chapter_4.avro.DeliveryEvent"
    ]}
  ]
}
```

❶ Outer class definition

❷ Avro union type at the field level

In this case, the generated Java code provides a single method `getTxnType()`, but it has return type of `Object`. As a result you'll need to use the same approach of checking for the instance type as you did above when using a union schema, essentially just pushing the issue of determining the record type from the class level to the field level.

NOTE **Java 16 introduces pattern matching with the `instanceof` keyword that removes the need for casting the object after the `instanceof` check**

4.6 Summary

- Kafka Producers send records in batches to topics located on the Kafka broker and will continue to retry sending failed batches until the `delivery.timeout.ms` configuration expires. You can configure a Kafka Producer to be an idempotent producer meaning it guarantees to send records only once and in-order for a given partition. Kafka producers also have a transactional mode that guarantees exactly once delivery of records across multiple topics. You enable the Kafka transactional API in producers by using the configuration `transactional.id` which must be a unique id for each producer. When using consumers in the transactional API, you want to make sure you set the `isolation.level` to read committed so you only consume committed records from transactional topics.
- Kafka Consumers read records from topics. Multiple consumers with the same group id get topic-partition assignments and work together as one logical consumer. Should one member of the group fail its topic-partition assignment(s) are redistributed to other members of the group via process known as rebalancing. Consumers periodically commit the offsets of consumed records so restarting after a shut-down they pick up where they left of processing.
- Kafka producers and consumers offer three different types of delivery guarantees at least once, at most once, and exactly once. At least once means no records are lost, but you may receive duplicates due to retries. At most once means that you won't receive duplicate records but there could be records lost due to errors. Exactly once delivery means you don't receive duplicates and you won't lose any records due to errors.
- Static membership provides you with stability in environments where consumers frequently drop off, only to come back online within a reasonable amount of time.
- The `CooperativeStickyAssignor` provides the much improved rebalance behavior. The cooperative rebalance protocol is probably the best choice to use in most cases as it significantly reduces the amount of downtime during a rebalance.
- The Admin API provides a way to create and manage topics, partitions and records programmatically.
- When you have different event types but the events are related and processing them in-order is important it's worth considering placing the multiple event types in a single topic.