4 Producers: Sourcing data

This chapters covers

- Sending messages and the producer
- Creating our own producer serializers and partitioners
- Examining configuration options to solve a company's requirements

In the previous chapter, we looked at the requirements that an organization might have regarding their data. Some design decisions we made have practical impacts on how we send data to Kafka. Let's now enter the world of an event-streaming platform through the portal gate of a Kafka producer. After reading this chapter, you will be well on your way to solving fundamental requirements of a Kafka project by producing data in a couple of different ways.

The producer, despite its importance, is only one part of this system. In fact, we can change some producer configuration options or set these at the broker or topic level. We will discover those options as we get further along, but getting data into Kafka is our first concern in this chapter.

4.1 An example

The producer provides the way to push data into the Kafka system for our example project. As a refresher, figure 4.1 illustrates where producers fit into Kafka.

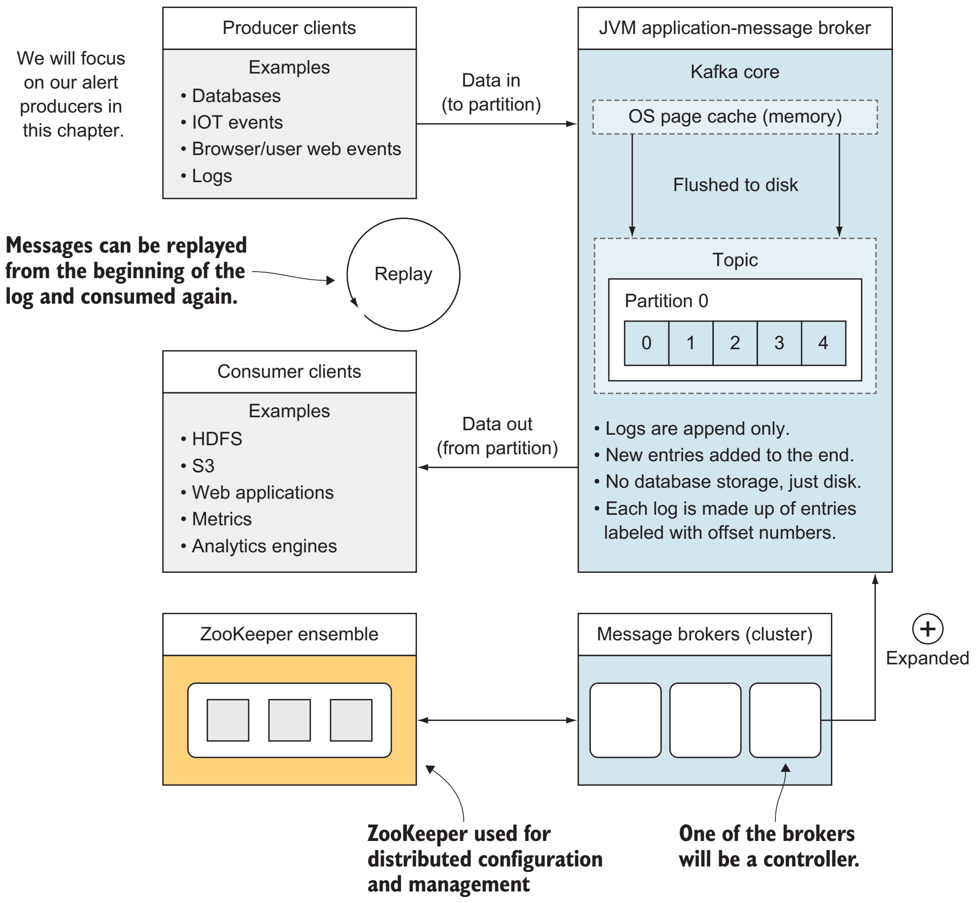
We will focus on our alert producers in this chapter.
Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Topic
Partition 0
0
1
2
3
4
Messages can be replayed from the beginning of the log and consumed again.
Replay
Consumer clients
Examples
• HDFS
• S3
• Web applications
• Metrics
• Analytics engines
Data out (from partition)
• Logs are append only.
• New entries added to the end.
• No database storage, just disk.
• Each log is made up of entries labeled with offset numbers.
ZooKeeper ensemble
Message brokers (cluster)
Expanded
ZooKeeper used for distributed configuration and management
One of the brokers will be a controller.

Figure 4.1 Kafka producers

Looking at figure 4.1, let's focus on the top-left corner (the producer clients), which shows examples of data being produced into Kafka. This data could be the IoT events we are using in our fictional company. To make the idea of producing data more concrete, let's imagine a practical example that we might have written for one of our projects. Let's look at an application that takes user feedback on how a website is working for its customers.

Currently, the user submits a form on the website that generates email to a support account or chatbot. Every now and then, one of our support staff checks the inbox to see what suggestions or issues customers have encountered. Looking to the future, we want to keep this information coming to us but in a way that allows the data to be more accessible than in an email inbox. If we instead send this message into a Kafka topic, we could produce more robust and varied replies, rather than just reactive email responses to customers. The benefit of flexibility comes from having the event in Kafka for any consuming applications to use.

Let's first look at what using email as part of our data pipeline impacts. Looking at figure 4.2, it might be helpful to focus on the format that the data is stored in once a user submits a form with feedback on our website.

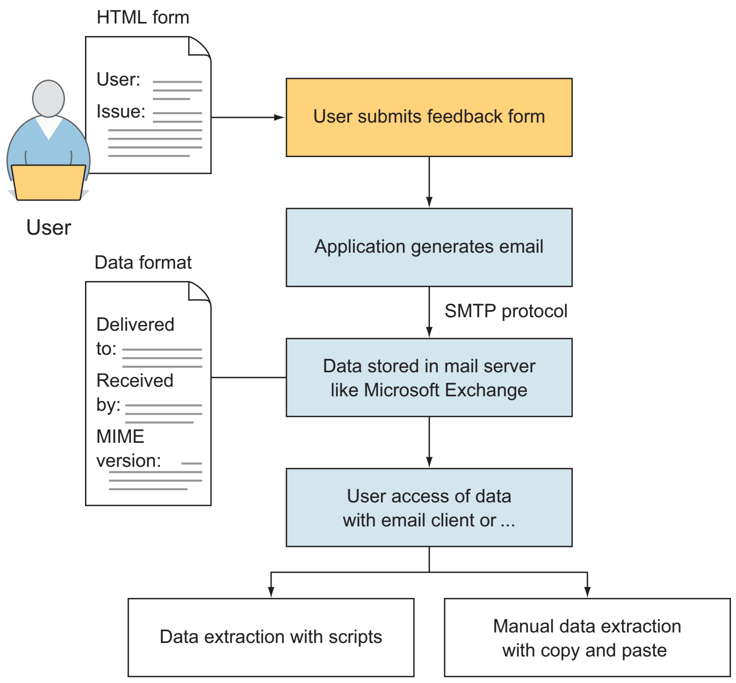
HTML form
User:
Issue:
User
User submits feedback form
Application generates email
SMTP protocol
Data format
Delivered to:
Received by:
MIME version:
Data stored in mail server like Microsoft Exchange
User access of data with email client or ...
Data extraction with scripts
Manual data extraction with copy and paste

Figure 4.2 Sending data in email

A traditional email uses Simple Mail Transfer Protocol (SMTP), and we will see that reflected in how the email event itself is presented and sometimes stored. We can use email clients like Microsoft® Outlook® to retrieve the data quickly, but rather than just reading email, how else can we pull data out of that system for other uses? Copy and paste are common manual steps, as well as email-parsing scripts. (Parsing scripts includes using a tool or programming language and libraries or frameworks to get the parsing correct.) In comparison, although Kafka uses its own protocol, it does not impose any specific format for our message data. We should be able to write the data in whatever format we choose.

NOTE In the previous chapter, we looked at the Apache Avro format as one of the common formats that the Kafka community uses. Protobuf and JSON are also widely popular [1].

Another usage pattern that comes to mind is to treat notifications of customer issues or website outages as temporary alerts that we can delete after replying to the customer. However, this customer input might serve more than one purpose. What if we are able to look for trends in outages that customers report? Does the site always slow to a crawl after sale coupon codes go out in mass-marketing emails? Could this data help us find features that our users are missing from our site? Do 40% of our user emails involve having trouble finding the Privacy settings for their account? Having this data present in a topic that can be replayed or read by several applications with different purposes can add more value to the customer than an automated support or bot email that is then deleted.

Also, if we have retention needs, those would be controlled by the teams running our email infrastructure versus a configuration setting we can control with Kafka. Looking again at figure 4.3, notice that the application has an HTML form but writes to a

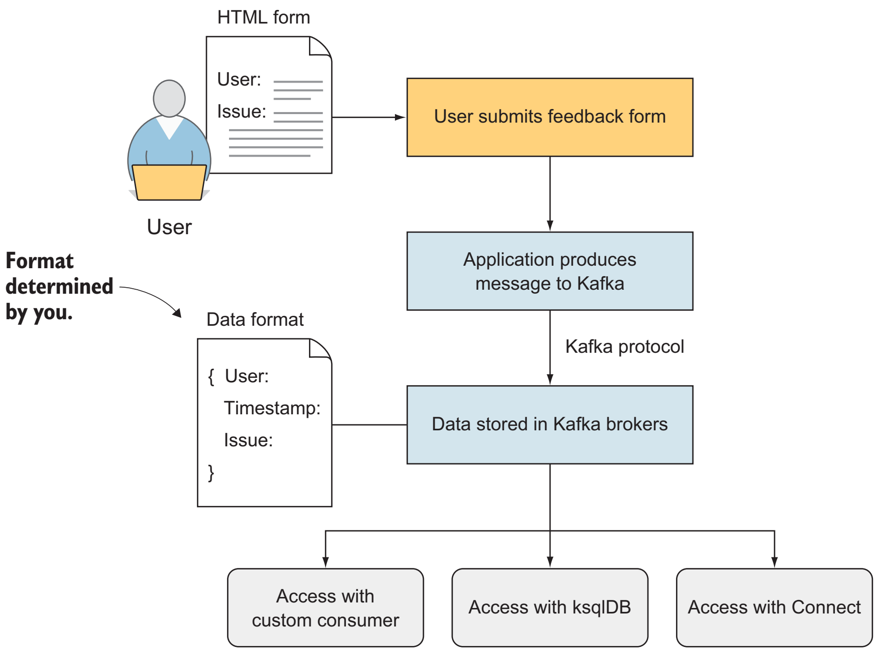
HTML form
User:
Issue:
User
User submits feedback form
Application produces message to Kafka
Format determined by you.
Data format
{ User:
Timestamp:
Issue:
}
Kafka protocol
Data stored in Kafka brokers
Access with custom consumer
Access with ksqlDB
Access with Connect

Figure 4.3 Sending data to Kafka

Kafka topic, not to an email server. With this approach, we can extract the information that is important for us in whatever format we need, and it can be used in many ways. Consuming applications can use schemes to work with the data and not be tied to a single protocol format. We can retain and reprocess these messages for new use cases because we control the retention of those events. Now that we have looked at why we might use a producer, let's quickly check out some details of a producer interacting with the Kafka brokers.

4.1.1 Producer notes

The producer's job includes fetching metadata about the cluster [2]. Because producers can only write to the replica leader of the partition they are assigned to, the metadata helps the producer determine which broker to write to as the user might have only included a topic name without any other details. This is nice because the producer's end user does not have to make a separate call to get that information. The end user, however, needs to have at least one running broker to connect to, and the Java client library figures out the rest.

Because this distributed system is designed to account for momentary errors such as a network blip, the logic for retries is already built in. However, if the ordering of the messages is essential, like for our audit messages, then besides setting the `retries` to a number like 3, we also need to set the `max.in.flight.requests.per.connection` value to `1` and set `acks` (the number of brokers that send acknowledgments back) to `all` [3] [4]. In our opinion, this is one of the safest methods to ensure that your producer's messages arrive in the order you intend [4]. We can set the values for both `acks` and `retries` as configuration parameters.

Another option to be aware of is using an idempotent producer. The term *idempotent* refers to how sending the same message multiple times only results in producing the message once. To use an idempotent producer, we can set the configuration property `enable.idempotence=true` [5]. We will not be using the idempotent producer in our following examples.

One thing we do not have to worry about is one producer getting in the way of another producer's data. Thread safety is not an issue because data will not be overwritten but handled by the broker itself and appended to the broker's log [6]. Now it is time to look at how to enable the values like `max.in.flight.requests.per.connection` in code.

4.2 Producer options

One of the things that was interesting when we started working with sending data into Kafka was the ease of setting options using the Java clients that we will specifically focus on in this book. If you have worked with other queue or messaging systems, the other systems' setups can include things like providing remote and local queues lists, manager hostnames, starting connections, connection factories, sessions, and more.

Although far from being set up totally hassle free, the producer works from the configuration on its own to retrieve much of the information it needs, such as a list of all of our Kafka brokers. Using the value from the property `bootstrap.servers` as a starting point, the producer fetches metadata about brokers and partitions that it uses for all subsequent writes.

As mentioned earlier, Kafka allows you to change key behaviors just by changing some configuration values. One way to deal with all of the producer configuration key names is to use the constants provided in the Java class `ProducerConfig` when developing producer code (see http://mng.bz/ZYdA) and by looking for the Importance label of "high" in the Confluent website [7]. However, in our examples, we will use the property names themselves for clarity.

Table 4.1 lists some of the most crucial producer configurations that support our specific examples. In the following sections, we'll look at what we need to complete our factory work.

Table 4.1 Important producer configurations

Key	Purpose
`acks`	Number of replica acknowledgments that a producer requires before success is established
`bootstrap.servers`	One or more Kafka brokers to connect for startup
`value.serializer`	The class that's used for serialization of the value
`key.serializer`	The class that's used for serialization of the key

4.2.1 Configuring the broker list

From our examples of writing messages to Kafka, it is clear that we have to tell the producer which topic to send messages to. Recall that topics are made up of partitions, but how does Kafka know where a topic partition resides? We, however, do not have to know the details of those partitions when we send messages. Perhaps an illustration will help clarify this conundrum. One of the required configuration options for producers is `bootstrap.servers`. Figure 4.4 shows an example of a producer that has only broker 0 in its list of bootstrap servers, but it will be able to learn about all three brokers in the cluster by starting with one only.

The `bootstrap.servers` property can take many or just one initial broker as in figure 4.4. By connecting to this broker, the client can discover the metadata it needs, which includes data about other brokers in the cluster as well [8].

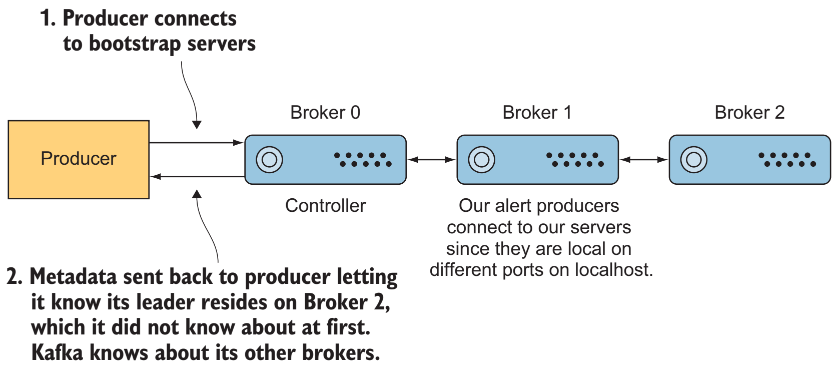
1. Producer connects to bootstrap servers
Producer
Broker 0
Broker 1
Broker 2
Controller
Our alert producers connect to our servers since they are local on different ports on localhost.
2. Metadata sent back to producer letting it know its leader resides on Broker 2, which it did not know about at first. Kafka knows about its other brokers.

Figure 4.4 Bootstrap servers

This configuration is key to helping the producer find a broker to talk to. Once the producer is connected to the cluster, it can obtain the metadata it needs to get the details (such as where the leader replica for the partition resides on disk) we did not previously provide. Producer clients can also overcome a failure of the partition leader they are writing to by using the information about the cluster to find a new leader. You might have noticed that ZooKeeper's information is not part of the configuration. Any metadata the producer needs will be handled without the producer client having to provide ZooKeeper cluster details.

4.2.2 How to go fast (or go safer)

Asynchronous message patterns are one reason that many use queue-type systems, and this powerful feature is also available in Kafka. We can wait in our code for the result of a producer send request, or we can handle success or failure asynchronously with callbacks or `Future` objects. If we want to go faster and not wait for a reply, we can still handle the results at a later time with our own custom logic.

Another configuration property that applies to our scenario is the `acks` key, which stands for *acknowledgments*. This controls how many acknowledgments the producer needs to receive from the partition leader's followers before it returns a completed request. The valid values for this property are `all`, `-1`, `1`, and `0` [9].

Figure 4.5 shows how a message with `ack` set to `0` behaves. Setting this value to `0` will probably get us the lowest latency but at the cost of safety. Additionally, guarantees are not made if any broker receives the message and, also, retries are not attempted [9]. As a sample use case, say that we have a web-tracking platform that collects the clicks on a page and sends these events to Kafka. In this situation, it might not be a big deal to lose a single link press or hover event. If one is lost, there is no real business impact.

In essence, the event in figure 4.5 was sent from the producer and forgotten. The message might have never made it to the partition. If the message did, by chance, make it to the leader replica, the producer will not know if any follower replica copies were successful.

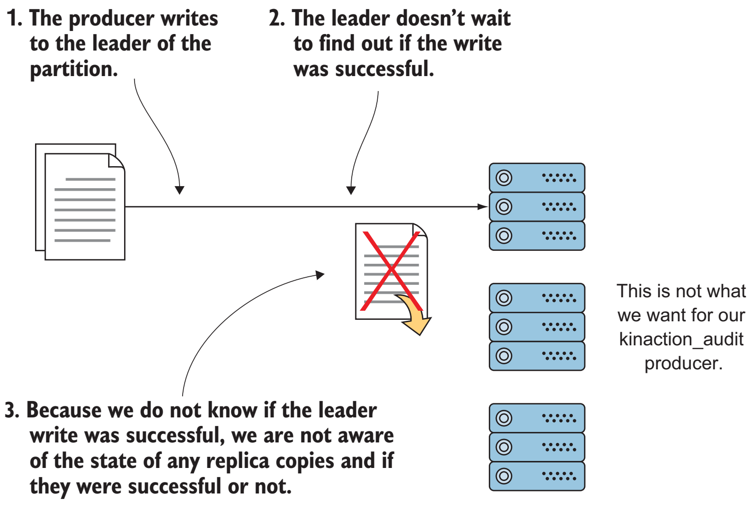
1. The producer writes to the leader of the partition.
2. The leader doesn't wait to find out if the write was successful.
This is not what we want for our kinaction_audit producer.
3. Because we do not know if the leader write was successful, we are not aware of the state of any replica copies and if they were successful or not.

Figure 4.5 The property `acks` equals `0`.

What we would consider the opposite setting to that used previously would be `acks` with values `all` or `-1`. The values `all` or `-1` are the strongest available option for this configuration setting. Figure 4.6 shows how the value `all` means that a partition leader's replica waits on the entire list of its in-sync replicas (ISRs) to acknowledge completion [9]. In other words, the producer will not get an acknowledgment of success until after all replicas for a partition are successful. It is easy to see that it won't be the quickest due to the dependencies it has on other brokers. In many cases, it is

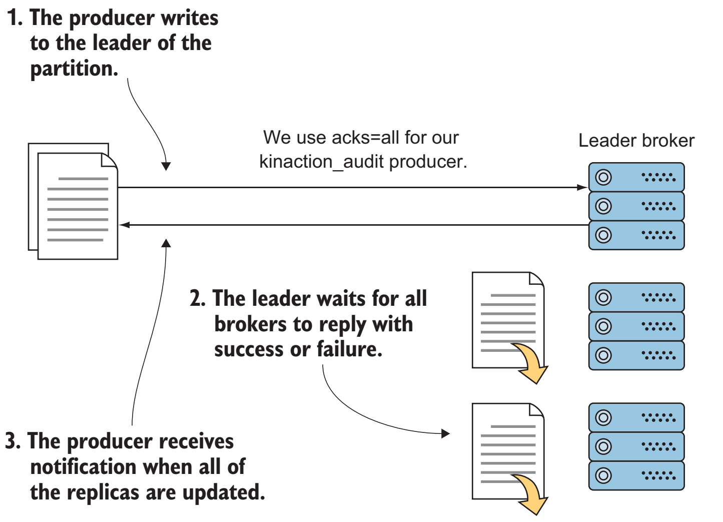
1. The producer writes to the leader of the partition.
We use acks=all for our kinaction_audit producer.
Leader broker
2. The leader waits for all brokers to reply with success or failure.
3. The producer receives notification when all of the replicas are updated.

Figure 4.6 The property `acks` equals `all`.

worth paying the performance price in order to prevent data loss. With many brokers in a cluster, we need to be aware of the number of brokers the leader has to wait on. The broker that takes the longest to reply is the determining factor for how long until a producer receives a success message.

Figure 4.7 shows the impact of setting the `acks` value to `1` and asking for an acknowledgment. An acknowledgment involves the receiver of the message (the leader replica of the specific partition) sending confirmation back to the producer. The producer client waits for that acknowledgment. However, the followers might not have copied the message before a failure brings down the leader. If that situation occurs before a copy is made, the message never appears on the replica followers for that partition [9]. Figure 4.7 shows that while the message was acknowledged by the leader replica and sent to the producer, a failure of any replica to make a copy of the message would appear as if the message never made it to the cluster.

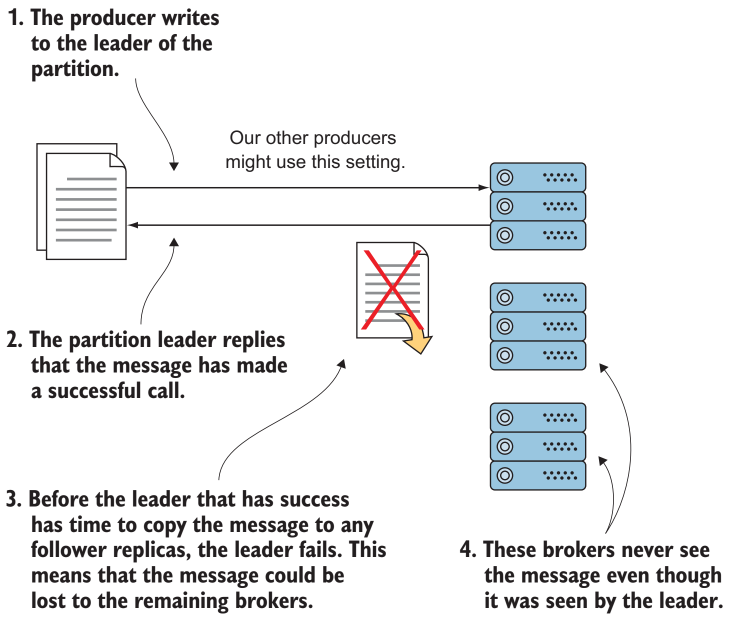
1. The producer writes to the leader of the partition.
Our other producers might use this setting.
2. The partition leader replies that the message has made a successful call.
3. Before the leader that has success has time to copy the message to any follower replicas, the leader fails. This means that the message could be lost to the remaining brokers.
4. These brokers never see the message even though it was seen by the leader.

Figure 4.7 The property `acks` equals `1`.

NOTE This is closely related to the ideas of at-most and at-least semantics that we covered in chapter 1 [10]. The `acks` setting is a part of that larger picture.

4.2.3 Timestamps

Recent versions of the producer record contain a timestamp on the events you send. A user can either pass the time into the constructor as a Java type `long` when sending a `ProducerRecord` Java object or the current system time. The actual time that is used in the message can stay as this value, or it can be a broker timestamp that occurs when the message is logged. Setting the topic configuration `message.timestamp.type` to `CreateTime` uses the time set by the client, whereas setting it to `LogAppendTime` uses the broker time [11].

Why would you want to choose one over the other? You might want to use the created time in order to have the time that a transaction (like a sales order) takes place rather than when it made its way to the broker. Using the broker time can be useful when the created time is handled inside the message itself or an actual event time is not business or order relevant.

As always, timestamps can be tricky. For example, we might get a record with an earlier timestamp than that of a record before it. This can happen in cases where a failure occurred and a different message with a later timestamp was committed before the retry of the first record completed. The data is ordered in the log by offsets and not by timestamp. Although reading timestamped data is often thought of as a consumer client concern, it is also a producer concern because the producer takes the first steps in ensuring message order.

As discussed earlier, this is also why `max.in.flight.requests.per.connection` is important when considering whether you want to allow retries or many inflight requests at a time. If a retry happens and other requests succeed on their first attempt, earlier messages might be added after the later ones. Figure 4.8 provides an example of when a message can get out of order. Even though message 1 was sent first, it does not make it into the log in an ordered manner because retries were enabled.

As a reminder, with Kafka versions before 0.10, timestamp information is not available as that feature was not included in earlier releases. We can still include a timestamp, though, but we would need to store it in the value of the message itself.

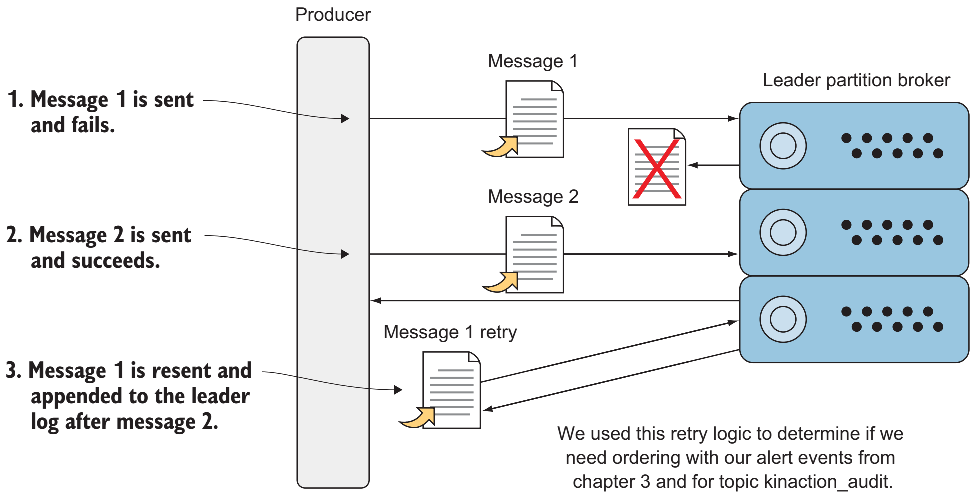
Producer
Message 1
Leader partition broker
1. Message 1 is sent and fails.
Message 2
2. Message 2 is sent and succeeds.
Message 1 retry
3. Message 1 is resent and appended to the leader log after message 2.
We used this retry logic to determine if we need ordering with our alert events from chapter 3 and for topic kinaction_audit.

Figure 4.8 Retry impact on order

Another option when using a producer is to create producer interceptors. These were introduced in KIP-42 (Kafka Improvement Proposal). Its main goal was to help support measuring and monitoring [12]. In comparison to using a Kafka Streams workflow to filter or aggregate data, or even creating different topics specifically for modified data, the usage of these interceptors might not be our first choice. At present, there are no default interceptors that run in the life cycle. In chapter 9, we will show a use case for tracing messages from producer clients to consumer clients with interceptors adding a trace ID.

4.3 Generating code for our requirements

Let's try to use the information we gathered about how producers work on our own solutions. We'll start with the audit checklist that we designed in chapter 3 for use with Kafka in our e-bike factory. As noted in chapter 3, we want to make sure that we do not lose any audit messages when operators complete commands against the sensors. One requirement was that there was no need to correlate (or group together) any events. Another requirement was to make sure we don't lose any messages. The following listing shows how we would start our producer configuration and how to make sure that we are safe for message acknowledgment by setting `acks` to `all`.

Listing 4.1 Configuring the audit producer

```
public class AuditProducer {

...
private static final Logger log = LoggerFactory.getLogger
(AuditProducer.class);Properties kaProperties = new Properties();

kaProperties.put( "bootstrap.servers",
  "localhost:9092,localhost:9093,localhost:9094");
kaProperties.put("acks", "all");
kaProperties.put("retries", "3");
kaProperties.put("max.in.flight.requests.per.connection", "1");
...
```

Creates properties as before for our configuration

Sets acks to all to get the strongest guarantee

Lets the client retry in case of failure so we don't have to implement our own failure logic

Notice that we did not have to touch anything except the configuration we send to the producer to address the concern of message loss. The `acks` configuration change is a small but powerful feature that has a significant impact on if a message arrives or not. Because we do not have to correlate (group) any events together, we are not using a key for these messages. However, there is a foundational part that we want to change in order to wait for the result before moving on. The following listing shows the `get` method, which is how we can bring about waiting for the response to complete synchronously before moving on in the code. Note that the following listing was informed by examples located at: https://docs.confluent.io/2.0.0/clients/producer.html#examples.

Listing 4.2 Waiting for a result

```
RecordMetadata result =
  producer.send(producerRecord).get();          <-- Waits on the response from the send call
log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
        result.offset(), result.topic(), result.timestamp());
  producer.close();
```

Waiting on the response directly in a synchronous way ensures that the code is handling each record's results as they come back before another message is sent. The focus is on delivering the messages without loss, more than on speed!

So far, we have used a couple of prebuilt serializers in earlier chapters. For plain text messages, our producer uses a serializer called `StringSerializer`. And when we talked about Avro in chapter 3, we reached for the class `io.confluent.kafka.serializers.KafkaAvroSerializer`. But what if we have a specific format we want to produce? This often happens when trying to work with custom objects. We'll use serialization to translate data into a format that can be transmitted, stored, and then retrieved to achieve a clone of our original data. The following listing shows the code for our `Alert` class.

Listing 4.3 `Alert` class

```
public class Alert implements Serializable {

  private final int alertId;
  private String stageId;
  private final String alertLevel;
  private final String alertMessage;

  public Alert(int alertId,
    String stageId,
    String alertLevel,
    String alertMessage) {      <-- Holds the alert's ID, level, and messages

    this.alertId = alertId;
    this.stageId = stageId;
    this.alertLevel = alertLevel;
    this.alertMessage = alertMessage;
  }

  public int getAlertId() {
    return alertId;
  }

  public String getStageId() {
    return stageId;
  }
  public void setStageId(String stageId) {
    this.stageId = stageId;
  }

  public String getAlertLevel() {
    return alertLevel;
  }

  public String getAlertMessage() {
    return alertMessage;
  }
}
```

Listing 4.3 shows code that we use to create a bean named `Alert` to hold the information we want to send. Those familiar with Java will notice that the listing is nothing more than getters and setters and a constructor for the `Alert` class. Now that there is a format for the `Alert` data object, it is time to use it in making a simple alert `Serializer` called `AlertKeySerde` as the following listing shows.

Listing 4.4 Our `Alert` serializer

```
public class AlertKeySerde implements Serializer<Alert>,
                                      Deserializer<Alert> {

  public byte[] serialize(String topic, Alert key) {     <- Sends the topic and the
    if (key == null) {                                      Alert object to our method
      return null;
    }
    return key.getStageId()
      .getBytes(StandardCharsets.UTF_8);     <- Converts objects to
  }                                             bytes (our end goal)

  public Alert deserialize
    (String topic, byte[] value) {              <- The rest of the interface methods do
    //could return Alert in future if needed       not need any logic at this point.
    return null;
  }

  //...
}
```

In listing 4.5, we use this custom class only as the key serializer for the moment, leaving the value serializer as a `StringSerializer`. It is interesting to note that we can serialize keys and values with different serializers on the same message. But we should be mindful of our intended serializers and the configuration values for both. The code implements the `Serializer` interface and only pulls out the field `stageId` to use as a key for our message. This example should be straightforward because the focus is on the technique of using a serde. Other options for serdes that are often used are JSON and Avro implementations.

NOTE If you see or hear the term *serde*, it means that the serializer and deserializer are both handled by the same implementation of that interface [13].

However, it is still common to see each interface defined separately. Just watch when you use `StringSerializer` versus `StringDeserializer`; the difference can be hard to spot!

Another thing to keep in mind is that knowing how to deserialize the values involves the consumers in relation to how the values were serialized by the producer. Some sort of agreement or coordinator is needed for the data formats for clients even though Kafka does not care what data it stores on the brokers.

Another goal of our design for the factory was to capture the alert trend status of our stages to track their alerts over time. Because we care about the information for each stage (and not all sensors at a time), it might be helpful to think of how we are going to group these events. In this case, as each stage ID is unique, it makes sense that we can use that ID as a key. The following listing shows the `key.serializer` property that we'll set, as well as sending a `CRITICAL` alert.

Listing 4.5 Alert trending producer

```
public class AlertTrendingProducer {

  private static final Logger log =
      LoggerFactory.getLogger(AlertTrendingProducer.class);

  public static void main(String[] args)
      throws InterruptedException, ExecutionException {

    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers",
      "localhost:9092,localhost:9093,localhost:9094");
    kaProperties.put("key.serializer",
      AlertKeySerde.class.getName());
    kaProperties.put("value.serializer",
      "org.apache.kafka.common.serialization.StringSerializer");

    try (Producer<Alert, String> producer =
      new KafkaProducer<>(kaProperties)) {

      Alert alert = new Alert(0, "Stage 0", "CRITICAL", "Stage 0 stopped");
      ProducerRecord<Alert, String> producerRecord =
          new ProducerRecord<>("kinaction_alerttrend",
            alert, alert.getAlertMessage());

      RecordMetadata result = producer.send(producerRecord).get();
      log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
               result.offset(), result.topic(), result.timestamp());
    }
  }
}
```

Tells our producer client how to serialize our custom Alert object into a key

Instead of null for the second parameter, uses the actual object we want to populate the key

In general, the same key should produce the same partition assignment, and nothing will need to be changed. In other words, the same stage IDs (the keys) are grouped together just by using the correct key. We will keep an eye on the distribution of the size of the partitions to note if they become uneven in the future, but for now, we will go along with this. Also, note that for our specific classes that we created in the manuscript, we are setting the class properties in a different way to show a different option. Instead of hardcoding the entire path of the class, you can use something like `AlertKeySerde.class.getName()` or even `AlertKeySerde.class` for the value of the property.

Our last requirement was to have alerts quickly processed to let operators know about any critical outages so we can group by the stage ID in this case as well. One reason for doing this is that we can tell if a sensor failed or recovered (any state change) by looking at only the last event for that stage ID. We do not care about the history of the status checks, only the current scenario. In this case, we also want to partition our alerts.

So far in our examples of writing to Kafka, the data was directed to a topic with no additional metadata provided from the client. Because the topics are made up of partitions that sit on the brokers, Kafka provides a default way to send messages to a specific partition. The default for a message with no key (which we used in the examples thus far) was a round-robin assignment strategy prior to Kafka version 2.4. In versions after 2.4, messages without keys use a sticky partition strategy [14]. However, sometimes we have specific ways that we want our data to be partitioned. One way to take control of this is to write our own unique partitioner class.

The client also has the ability to control what partition it writes to by configuring a unique partitioner. One example to think about is the alert levels from our sensor-monitoring service that was discussed in chapter 3. Some sensors' information might be more important than others; these might be on the critical path of our e-bike, which would cause downtime if not addressed. Let's say we have four levels of alerts: Critical, Major, Minor, and Warning. We could create a partitioner that places the different levels in different partitions. Our consumer clients would always make sure to read the critical alerts before processing the others.

If our consumers keep up with the messages being logged, critical alerts probably would not be a huge concern. However, listing 4.6 shows that we could change the partition assignment with a class to make sure that our critical alerts are directed to a specific partition (like partition 0). (Note that other alerts could end up on partition 0 as well due to our logic, but that critical alerts will always end up there.) The logic mirrors an example of the `DefaultPartitioner` used in Kafka itself [15].

Listing 4.6 Partitioner for alert levels

```
public int partition(final String topic
                     # ...
```

AlertLevelPartitioner needs to implement the partition method for its core logic.

```
  int criticalLevelPartition = findCriticalPartitionNumber(cluster, topic);
```

```
    return isCriticalLevel(((Alert) objectKey).getAlertLevel()) ?
      criticalLevelPartition :
        findRandomPartition(cluster, topic, objectKey);
  }
  //...
```

Critical alerts should end up the partition returned from findCriticalPartitionNumber

By implementing the `Partitioner` interface, we can use the `partition` method to send back the specific partition we want our producer to write to. In this case, the value of the key ensures that any `CRITICAL` event makes it to a specific place, partition 0 can be imagined to be sent back from the method `findCriticalPartitionNumber`, for example. In addition to creating the class itself, listing 4.7 shows how we need to set the configuration key, `partitioner.class`, for our producer to use the specific class we created. The configuration that powers Kafka is used to leverage our new class.

Listing 4.7 Configuring the `partitioner` class

```
Properties kaProperties = new Properties();
//...
kaProperties.put("partitioner.class",
          AlertLevelPartitioner.class.getName());
```

Updates the producer configuration to reference and use the custom partitioner AlertLevelPartitioner

This example, in which a specific partition number is always sent back, can be expanded on or made even more dynamic. We can use custom code to accomplish the specific logic of our business needs.

Listing 4.8 shows the configuration of the producer to add the `partitioner.class` value to use as our specific partitioner. The intention is for us to have the data available in a specific partition, so consumers that process the data can have access to the critical alerts specifically and can go after other alerts (in other partitions) when they are handled.

Listing 4.8 Alert producer

```
public class AlertProducer {
  public static void main(String[] args) {

    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers",
      "localhost:9092,localhost:9093");
    kaProperties.put("key.serializer",
      AlertKeySerde.class.getName());
    kaProperties.put("value.serializer",
      "org.apache.kafka.common.serialization.StringSerializer");
    kaProperties.put("partitioner.class",
      AlertLevelPartitioner.class.getName());

    try (Producer<Alert, String> producer =
      new KafkaProducer<>(kaProperties)) {
```

Reuses the Alert key serializer

Uses the property partitioner.class to set our specific partitioner class

```
      Alert alert = new Alert(1, "Stage 1", "CRITICAL", "Stage 1 stopped");
      ProducerRecord<Alert, String>
          producerRecord = new ProducerRecord<>
            ("kinaction_alert", alert, alert.getAlertMessage());

      producer.send(producerRecord,
                    new AlertCallback());
    }
  }
}
```

This is the first time we've used a callback to handle the completion or failure of a send.

One addition we see in listing 4.8 is how we added a callback to run on completion. Although we said that we are not 100% concerned with message failures from time to time, due to the frequency of events, we want to make sure that we do not see a high failure rate that could be a hint at application-related errors. The following listing shows an example of implementing a `Callback` interface. The callback would log a message only if an error occurs. Note that the following listing was informed by examples located at https://docs.confluent.io/2.0.0/clients/producer.html#examples.

Listing 4.9 Alert callback

```
public class AlertCallback implements Callback {

  private static final Logger log =
    LoggerFactory.getLogger(AlertCallback.class);

  public void onCompletion
    (RecordMetadata metadata,
     Exception exception) {

    if (exception != null) {
      log.error("kinaction_error", exception);
    } else {
      log.info("kinaction_info offset = {}, topic = {}, timestamp = {}",
               metadata.offset(), metadata.topic(), metadata.timestamp());
    }
  }
}
```

Implements the Kafka Callback interface

The completion can have success or failure.

Although we will focus on small samples in most of our material, we think that it is helpful to look at how to use a producer in a real project as well. As mentioned earlier, Apache Flume can be used alongside Kafka to provide various data features. When we use Kafka as a sink, Flume places data into Kafka. You might (or might not) be familiar with Flume, but we are not interested in its feature set for this. We want to see how it leverages Kafka producer code in a real situation.

In the following examples, we reference Flume version 1.8 (located at https://github.com/apache/flume/tree/flume-1.8, if you want to view more of the complete source code). The following listing shows a configuration snippet that would be used by a Flume agent.

Listing 4.10 Flume sink configuration

```
a1.sinks.k1.kafka.topic = kinaction_helloworld
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
```

Some configuration properties from listing 4.10 seem familiar: `topic`, `acks`, `bootstrap.servers`. In our previous examples, we declared the configurations as properties inside our code. However, listing 4.10 shows an example of an application that externalizes the configuration values, which is something we could do on our projects as well. The `KafkaSink` source code from Apache Flume, found at http://mng.bz/JvpZ, provides an example of taking data and placing it inside Kafka with producer code. The following listing is a different example of a producer using a similar idea, taking a configuration file like that in listing 4.10 and loading those values into a producer instance.

Listing 4.11 Reading the Kafka producer configuration from a file

```
...
Properties kaProperties = readConfig();
String topic = kaProperties.getProperty("topic");
kaProperties.remove("topic");

try (Producer<String, String> producer =
                        new KafkaProducer<>(kaProperties)) {
  ProducerRecord<String, String> producerRecord =
    new ProducerRecord<>(topic, null, "event");
  producer.send(producerRecord,
                new AlertCallback());      <- Our familiar producer.send
}                                             with a callback

private static Properties readConfig() {
  Path path = Paths.get("src/main/resources/kafkasink.conf");

  Properties kaProperties = new Properties();      Reads an external file
  try (Stream<String>  lines = Files.lines(path))  <- for configuration
      lines.forEachOrdered(line ->
                        determineProperty(line, kaProperties));
  } catch (IOException e) {
    System.out.println("kinaction_error" + e);
  }
  return kaProperties;
}

private static void determineProperty                Parses configuration properties
  (String line, Properties kaProperties) {        <- and sets those values
  if (line.contains("bootstrap")) {
    kaProperties.put("bootstrap.servers", line.split("=")[1]);
  } else if (line.contains("acks")) {
      kaProperties.put("acks", line.split("=")[1]);
  } else if (line.contains("compression.type")) {
    kaProperties.put("compression.type", line.split("=")[1]);
  } else if (line.contains("topic")) {
    kaProperties.put("topic", line.split("=")[1]);
  }
  ...
}
```

Although some code is omitted in listing 4.11, the core Kafka producer pieces might be starting to look familiar. Setting the configuration and the producer `send` method should all look like the code we wrote in this chapter. And now, hopefully, you have the confidence to dig into which configuration properties were set and what impacts they will have.

One exercise left for the reader would be to compare how `AlertCallback.java` stacks up to the Kafka Sink callback class `SinkCallback`, located in the source code at http://mng.bz/JvpZ. Both examples uses the `RecordMetadata` object to find more information about successful calls. This information can help us learn more about where the producer message was written, including the partition and offset within that specific partition.

It is true that you can use applications like Flume without ever having to dig into its source code and still be successful. However, we think that if you want to know what is going on internally or need to do some advanced troubleshooting, it is important to know what the tools are doing. With your new foundational knowledge of producers, it should be apparent that you can make powerful applications using these techniques yourself.

4.3.1 Client and broker versions

One important thing to note is that Kafka broker and client versions do not always have to match. If you are running a broker that is at Kafka version 0.10.0 and the Java producer client you are using is at 0.10.2, the broker will handle this upgrade in the message version [16]. However, because you can does not mean you should do it in all cases. To dig into more of the bidirectional version compatibility, take a peek at KIP-97 (http://mng.bz/7jAQ).

We crossed a significant hurdle by starting to get data into Kafka. Now that we are deeper into the Kafka ecosystem, we have other concepts to conquer before we are done with our end-to-end solution. The next question is, how can we start to pull this data back out so our other applications can consume it? We now have some ideas about *how* we get data into Kafka, so we can start to work on learning more about making that data useful to other applications by getting it out in the correct ways. Consumer clients are a vital part of this discovery and, as with producers, there are various configuration-driven behaviors that we can use to help us satisfy different requirements for consumption.

Summary

- Producer clients provide developers a way to get data into Kafka.
- A large number of configuration options are available to control client behavior without custom code.

- Data is stored on the brokers in what is known as partitions.
- The client can control which partition the data gets written to by providing their own logic with the `Partitioner` interface.
- Kafka generally sees data as a series of bytes. However, custom serializers can be used to deal with specific data formats.

References

1 J. Kreps. “Why Avro for Kafka Data?” Confluent blog (February 25, 2015). https://www.confluent.io/blog/avro-kafka-data/ (accessed November 23, 2017).

2 “Sender.java.” Apache Kafka. GitHub (n.d.). https://github.com/apache/kafka/blob/299eea88a5068f973dc055776c7137538ed01c62/clients/src/main/java/org/apache/kafka/clients/producer/internals/Sender.java (accessed August 20, 2021).

3 “Producer Configurations: Retries.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_retries (accessed May 29, 2020).

4 “Producer Configurations: max.in.flight.requests.per.connection.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#max.in.flight.requests.per.connection (accessed May 29, 2020).

5 “Producer Configurations: enable.idempotence.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_enable.idempotence (accessed May 29, 2020).

6 “KafkaProducer.” Apache Software Foundation (n.d.). https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html (accessed July 7, 2019).

7 “Producer Configurations.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html (accessed May 29, 2020).

8 “Producer Configurations: bootstrap.servers.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html #bootstrap.servers (accessed May 29, 2020).

9 “Producer Configurations: acks.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#acks (accessed May 29, 2020).

10 “Documentation: Message Delivery Semantics.” Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#semantics (accessed May 30, 2020).

11 “Topic Configurations: message.timestamp.type.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html#topicconfigs_message.timestamp.type (accessed July 22, 2020).

12 KIP-42: "Add Producer and Consumer Interceptors," Wiki for Apache Kafka, Apache Software Foundation. https://cwiki.apache.org/confluence/display/KAFKA/KIP-42%3A+Add+Producer+and+Consumer+Interceptors (accessed April 15, 2019).

13 "Kafka Streams Data Types and Serialization." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/streams/developer-guide/datatypes.html (accessed August 21, 2021).

14 J. Olshan. "Apache Kafka Producer Improvements with the Sticky Partitioner." Confluent blog (December 18, 2019). https://www.confluent.io/blog/apache-kafka-producer-improvements-sticky-partitioner/ (accessed August 21, 2021).

15 "DefaultPartitioner.java," Apache Software Foundation. GitHub (n.d.). https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/internals/DefaultPartitioner.java (accessed March 22, 2020).

16 C. McCabe. "Upgrading Apache Kafka Clients Just Got Easier." Confluent blog (July 18, 2017). https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier/ (accessed August 21, 2021).