2 Getting to know Kafka

This chapters covers

- The high-level architecture of Kafka
- Understanding client options
- How applications communicate with a broker
- Producing and consuming your first message
- Using Kafka clients with a Java application

Now that we have a high-level view of where Kafka shines and why one would use it, let's dive into the Kafka components that make up the whole system. Apache Kafka is a distributed system at heart, but it is also possible to install and run it on a single host. That gives us a starting point to dive into our sample use cases. As is often the case, the real questions start flowing once the hands hit the keyboard. By the end of this chapter, you will be able to send and retrieve your first Kafka message from the command line. Let's get started with Kafka and then spend a little more time digging into Kafka's architectural details.

NOTE Visit appendix A if you do not have a Kafka cluster to use or are interested in starting one locally on your machine. Appendix A works on updating

the default configuration of Apache Kafka and on starting the three brokers we will use in our examples. Confirm that your instances are up and running before attempting any examples in this book! If any examples don't seem to work, please check the source code on GitHub for tips, errata, and suggestions.

2.1 Producing and consuming a message

A *message*, also called a *record*, is the basic piece of data flowing through Kafka. Messages are how Kafka represents your data. Each message has a timestamp, a value, and an optional key. Custom headers can be used if desired as well [1]. A simple example of a message could be something like the following: the machine with host ID "1234567" (a *message key*) failed with the message "Alert: Machine Failed" (a *message value*) at "2020-10-02T10:34:11.654Z" (a *message timestamp*). Chapter 9 shows an example of using a custom header to set a key-value pair for a tracing use case.

Figure 2.1 shows probably the most important and common parts of a message that users deal with directly. Keys and values will be the focus of most of our discussion in this chapter, which require analysis when designing our messages. Each key and value can interact in its own specific ways to serialize or deserialize its data. The details of how to use serialization will start to come into focus when covering producing messages in chapter 4.

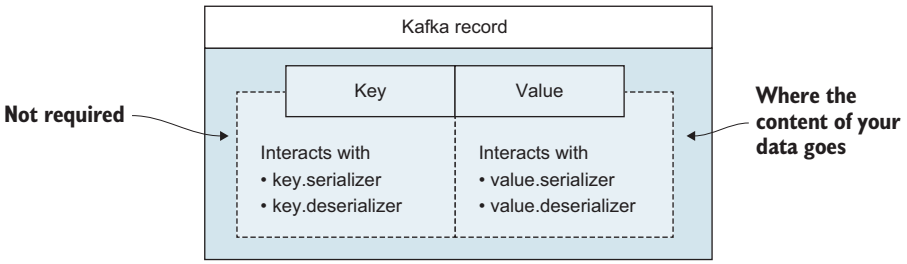

Figure 2.1 Kafka messages are made up of a key and a value (timestamp and optional headers are not shown).

Now that we have a record, how do we let Kafka know about it? You will deliver this message to Kafka by sending it to what are known as *brokers*.

2.2 What are brokers?

Brokers can be thought of as the server side of Kafka [1]. Before virtual machines and Kubernetes®, you may have seen one physical server hosting one broker. Because almost all clusters have more than one server (or node), we will have three Kafka servers running for most of our examples. This local test setup should let us see the output of commands against more than one broker, which will be similar to running with multiple brokers across different machines.

For our first example, we will create a topic and send our first message to Kafka from the command line. One thing to note is that Kafka was built with the command

line in mind. There is no GUI that we will use, so we need to have a way to interact with the operating system's command line interface. The commands are entered into a text-based prompt. Whether you use vi, Emacs, Nano, or whatever, make sure that it is something you feel comfortable editing with.

> **NOTE** Although Kafka can be used on many operating systems, it is often deployed in production on Linux, and command line skills will be helpful when using this product.

Shell helper

If you are a command line user and want a shortcut to autocomplete commands (and to help with the available arguments), check out a Kafka autocomplete project at http://mng.bz/K480. If you are a Zsh user, you may also want to check out and install Kafka's Zsh-completion plugin from https://github.com/Dabz/kafka-zsh-completions.

To send our first message, we will need a place to send it. To create a topic, we will run the `kafka-topics.sh` command in a shell window with the `--create` option (listing 2.1). You will find this script in the installation directory of Kafka, where the path might look like this: ~/kafka_2.13-2.7.1/bin. Note that Windows users can use the .bat files with the same name as the shell equivalent. For example, `kafka-topics.sh` has the Windows equivalent script named `kafka-topics.bat`, which should be located in the <kafka_install_directory>/bin/windows directory.

> **NOTE** The references in this work to `kinaction` and `ka` (like used in `kaProperties`) are meant to represent different abbreviations of *Kafka in Action* and are not associated with any product or company.

Listing 2.1 Creating the `kinaction_helloworld` topic

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9094
  --topic kinaction_helloworld --partitions 3 --replication-factor 3
```

You should see the output on the console where you just ran the command: `Created topic kinaction_helloworld`. In listing 2.1, the name `kinaction_helloworld` is used for our topic. We could have used any name, of course, but a popular option is to follow general Unix/Linux naming conventions, including not using spaces. We can avoid many frustrating errors and warnings by not including spaces or various special characters. These do not always play nicely with command line interfaces and autocompletion.

There are a couple of other options whose meaning may not be clear just yet, but to keep moving forward with our exploration, we will quickly define them. These topics will be covered in greater detail in chapter 6.

The `--partitions` option determines how many parts we want the topic to be split into. For example, because we have three brokers, using three partitions gives us one

partition per broker. For our test workloads, we might not need this many, based on data needs alone. However, creating more than one partition at this stage lets us see how the system works in spreading data across partitions. The `--replication-factor` also is set to three in this example. In essence, this says that for each partition, we want to have three replicas. These copies are a crucial part of our design to improve reliability and fault tolerance. The `--bootstrap-server` option points to our local Kafka broker. This is why the broker should be running before invoking this script. For our work right now, the most important goal is to get a picture of the layout. We will dig into how to best estimate the numbers we need in other use cases when we get into the broker details later.

We can also look at all existing topics that have been created and make sure that our new one is on the list. The `--list` option is what we can reach for to achieve this output. Again, we run the next listing in the terminal window.

Listing 2.2 Verifying the topic

```
bin/kafka-topics.sh --list --bootstrap-server localhost:9094
```

To get a feel for how our new topic looks, listing 2.3 shows another command that we can run to give us a little more insight into our cluster. Note that our topic is not like a traditional single topic in other messaging systems: we have replicas and partitions. The numbers we see next to the labels for the `Leader`, `Replicas`, and `Isr` fields are the `broker.id`s that correspond to the value for our three brokers that we set in our configuration files. Briefly looking at the output, we can see that our topic consists of three partitions: `Partition 0`, `Partition 1`, and `Partition 2`. Each partition was replicated three times as we intended on topic creation.

Listing 2.3 Describing the topic `kinaction_helloworld`

```
bin/kafka-topics.sh --bootstrap-server localhost:9094 \
  --describe --topic kinaction_helloworld
```

--describe lets us look at the details of the topic we pass in.

```
Topic:kinaction_helloworld PartitionCount:3 ReplicationFactor:3  Configs:
Topic: kinaction_helloworld Partition: 0 Leader: 0 Replicas: 0,1,2  Isr: 0,1,2
Topic: kinaction_helloworld Partition: 1 Leader: 1 Replicas: 1,2,0  Isr: 1,2,0
Topic: kinaction_helloworld Partition: 2 Leader: 2 Replicas: 2,0,1  Isr: 2,0,1
```

The output from listing 2.3 shows in the first line a quick data view of the total count of partitions and replicas that this topic has. The following lines show each partition for the topic. The second line of output is specific to the partition labeled `0` and so forth. Let's zoom in on partition 0, which has its replica copy leader on broker 0. This partition also has replicas that exist on brokers 1 and 2. The last column, `Isr`, stands for *in-sync replicas*. In-sync replicas show which brokers are current and not lagging behind the leader. Having a partition replica copy that is out of date or behind the leader is an issue that we will cover later. Still, it is critical to remember that replica

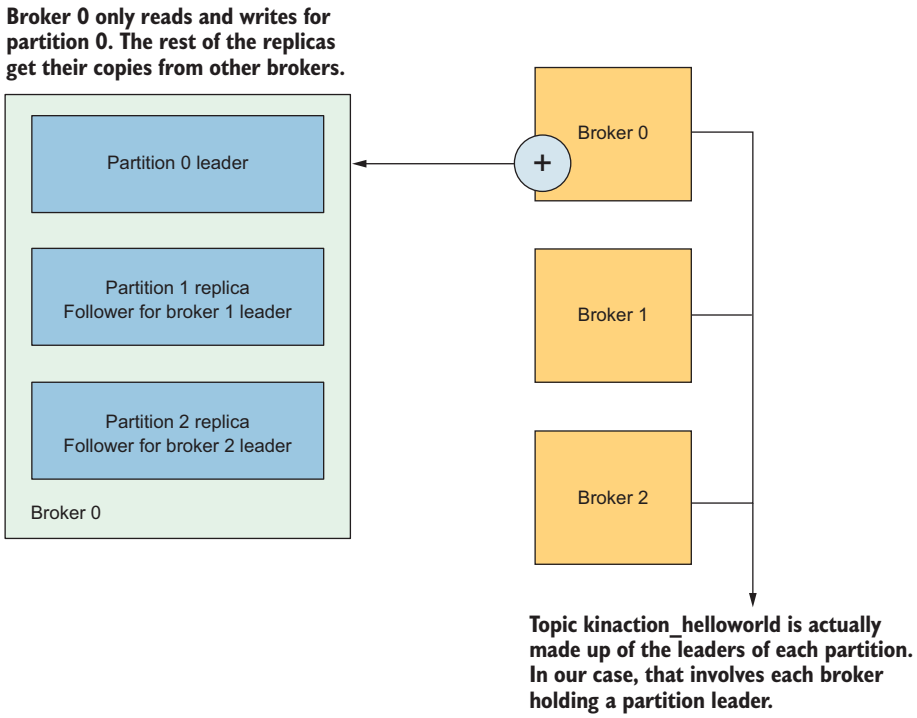

Figure 2.2 View of one broker

health in a distributed system is something that we will want to keep an eye on. Figure 2.2 shows a view if we look at the one broker with ID `0`.

For our `kinaction_helloworld` topic, note how broker 0 holds the leader replica for partition 0. It also holds replica copies for partitions 1 and 2 for which it is not the leader replica. In the case of its copy of partition 1, the data for this replica will be copied from broker 1.

> **NOTE** When we reference a partition leader in the image, we are referring to a *replica leader*. It is important to know that a partition can consist of one or more replicas, but only one replica will be a leader. A leader's role involves being updated by external clients, whereas nonleaders take updates only from their leader.

Now once we have created our topic and verified that it exists, we can start sending real messages! Those who have worked with Kafka before might ask why we took the preceding step to create the topic before sending a message. There is a configuration to enable or disable the autocreation of topics. However, it is usually best to control the creation of topics as a specific action because we do not want new topics to randomly show up if someone mistypes a topic name once or twice or to be recreated due to producer retries.

To send a message, we will start a terminal tab or window to run a producer as a console application to take user input [2]. The command in listing 2.4 starts an interactive program that takes over the shell; you won't get your prompt back to type more commands until you press Ctrl-C to quit the running application. You can just start typing, maybe something as simple as the default programmer's first print statement with a prefix of `kinaction` (for *Kafka In Action*) as the following listing demonstrates. We use `kinaction_helloworld` in the vein of the "hello, world" example found in the book, *The C Programming Language* [3].

Listing 2.4 Kafka producer console command

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9094 \
  --topic kinaction_helloworld
```

Notice in listing 2.4 that we reference the topic that we want to interact with using a `bootstrap-server` parameter. This parameter can be just one (or a list) of the current brokers in our cluster. By supplying this information, the cluster can obtain the metadata it needs to work with the topic.

Now, we will start a new terminal tab or window to run a consumer that also runs as a console application. The command in listing 2.5 starts a program that takes over the shell as well [2]. On this end, we should see the message we wrote in the producer console. Make sure that you use the same `topic` parameter for both commands; otherwise, you won't see anything.

Listing 2.5 Kafka consumer command

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 \
  --topic kinaction_helloworld --from-beginning
```

The following listing shows an example of the output you might see in your console window.

Listing 2.6 Example consumer output for `kinaction_helloworld`

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9094 \
  --topic kinaction_helloworld --from-beginning

kinaction_helloworld
...
```

As we send more messages and confirm the delivery to the consumer application, we can terminate the process and eliminate the `--from-beginning` option when we restart it. Notice that we didn't see all of the previously sent messages. Only those messages produced since the consumer console was started show up. The knowledge of which messages to read next and the ability to consume from a specific offset are tools we will leverage later as we discuss consumers in chapter 5. Now that we've seen a simple example in action, we have a little more background to discuss the parts we utilized.

2.3 Tour of Kafka

Table 2.1 shows the major components and their roles within the Kafka architecture. In the following sections, we'll dig into each of these items further to get a solid foundation for the following chapters.

Table 2.1 The Kafka architecture

Component	Role
Producer	Sends messages to Kafka
Consumer	Retrieves messages from Kafka
Topics	Logical name of where messages are stored in the broker
ZooKeeper ensemble	Helps maintain consensus in the cluster
Broker	Handles the commit log (how messages are stored on the disk)

2.3.1 Producers and consumers

Let's pause for a moment on the first stop on our tour: producers and consumers. Figure 2.3 highlights how producers and consumers differ in the direction of their data in relation to the cluster.

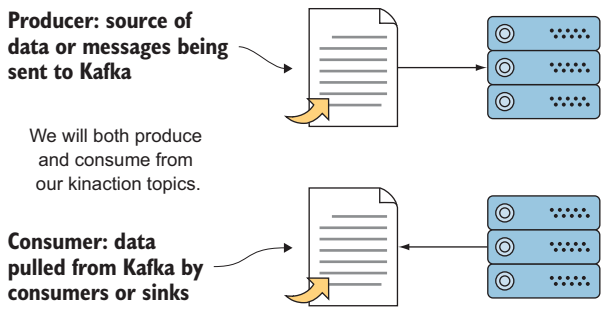

Figure 2.3 Producers vs. consumers

A *producer* is a tool for sending messages to Kafka topics [1]. As mentioned in our use cases in chapter 1, a good example is a log file that is produced from an application. Those files are not a part of the Kafka system until they are collected and sent to Kafka. When you think of input (or data) going into Kafka, you are looking at a producer being involved somewhere internally.

There are no default producers, per se, but the APIs that interact with Kafka use producers in their own implementation code. Some entry paths into Kafka might include using a separate tool such as Flume or even other Kafka APIs such as Connect and Streams. `WorkerSourceTask`, inside the Apache Kafka Connect source code (from version 1.0), is one example where a producer is used internally of its implementation. It provides its own higher-level API. This specific version 1.0 code is available under an Apache 2 license (https://github.com/apache/kafka/blob/trunk/LICENSE) and is viewable on GitHub (see http://mng.bz/9N4r). A producer is also used to send messages inside Kafka itself. For example, if we are reading data from a specific topic and want to send it to a different topic, we would also use a producer.

To get a feel for what our own producer will look like, it might be helpful to look at code similar in concept to `WorkerSourceTask`, which is the Java class that we mentioned earlier. Listing 2.7 shows our example code. Not all of the source code is listed for the `main` method, but what is shown is the logic of sending a message with the standard `KafkaProducer`. It is not vital to understand each part of the following example. Just try to get familiar with the producer's usage in the listing.

Listing 2.7 A producer sending messages

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

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

producer.close();
```

The ProducerRecord holds each message sent into Kafka.

Makes the actual call to send to our brokers

Callbacks can be used for asynchronous sending of messages.

To send data to Kafka, we created a `ProducerRecord` in listing 2.7. This object lets us define our message and specify the topic (in this case, `kinaction_alert`) to which we want to send the message. We used a custom `Alert` object as our key in the message. Next, we invoked the `send` method to send our `ProducerRecord`. While we can wait for the message, we can also use a callback to send asynchronous messages but still handle any errors. Chapter 4 provides this entire example in detail.

Figure 2.4 shows a user interaction that could start the process of sending data into a producer. A user on a web page that clicks might cause an audit event that would be produced in a Kafka cluster.

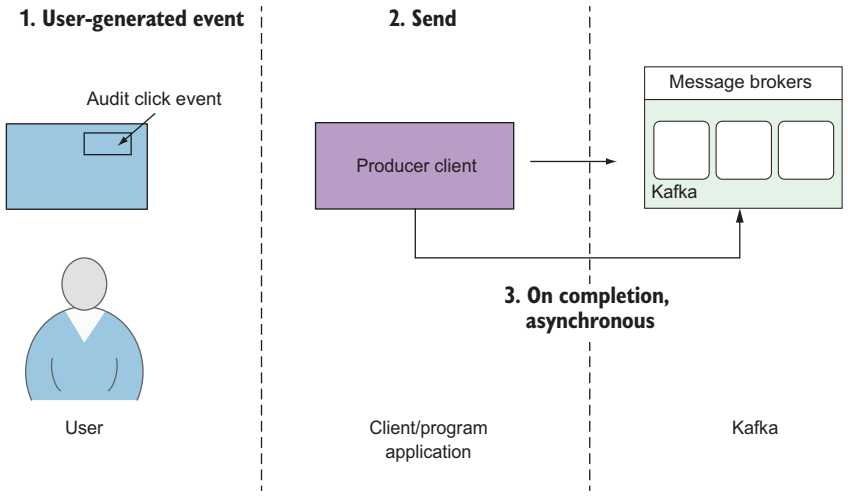

Figure 2.4 Producer example for user event

In contrast to a producer, a *consumer* is a tool for retrieving messages from Kafka [1]. In the same vein as producers, if we are talking about getting data out of Kafka, we look at consumers as being involved directly or indirectly. `WorkerSinkTask` is another class inside the Apache Kafka Connect source code from version 1.0 that shows the use of a consumer that is parallel with the producer example from Connect as well (see http://mng.bz/WrRW). Consuming applications subscribe to the topics that they are interested in and continuously poll for data. `WorkerSinkTask` provides a real example in which a consumer is used to retrieve records from topics in Kafka. The following listing shows the consumer example we will create in chapter 5. It displays concepts similar to `WorkerSinkTask.java`.

Listing 2.8 Consuming messages

```
...
consumer.subscribe(List.of("kinaction_audit"));
while (keepConsuming) {
  var records = consumer.
    poll(Duration.ofMillis(250));
  for (ConsumerRecord<String, String> record : records) {
    log.info("kinaction_info offset = {}, kinaction_value = {}",
             record.offset(), record.value());

    OffsetAndMetadata offsetMeta =
      new OffsetAndMetadata(++record.offset(), "");

    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap = new HashMap<>();
    kaOffsetMap.put(new TopicPartition("kinaction_audit",
      record.partition()), offsetMeta);

    consumer.commitSync(kaOffsetMap);
  }
}
...
```

The consumer subscribes to the topics that it cares about.

Messages are returned from a poll of data.

Listing 2.8 shows how a consumer object calls a `subscribe` method, passing in a list of topics that it wants to gather data from (in this case, `kinaction_audit`). The consumer then polls the topic(s) (see figure 2.5) and handles any data brought back as `ConsumerRecords`.

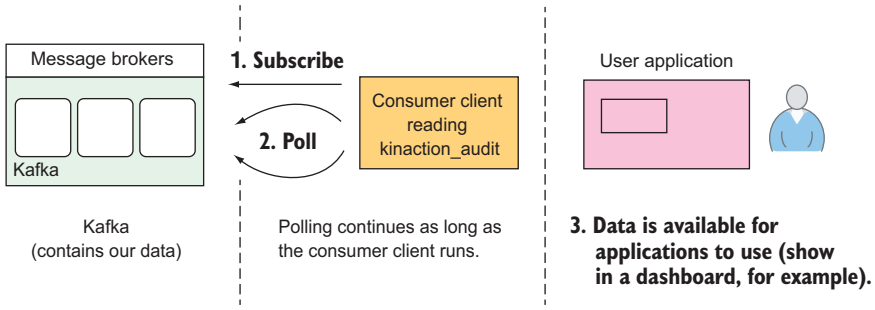

Figure 2.5 Consumer example flow

The previous code listings 2.7 and 2.8 show two parts of a concrete use case example as displayed in figures 2.4 and 2.5. Let's say that a company wants to know how many users clicked on their web page for a new factory command action. The click events generated by users would be the data going into the Kafka ecosystem. The data's consumers would be the factory itself, which would be able to use its applications to make sense of the data.

Putting data into Kafka and out of Kafka with code like the previous (or even with Kafka Connect itself) allows users to work with the data that can impact their business requirements and goals. Kafka does not focus on processing the data for applications: the consuming applications are where the data really starts to provide business value. Now that we know how to get data into and out of Kafka, the next area to focus on is where it lands in our cluster.

2.3.2 Topics overview

Topics are where most users start to think about the logic of what messages should go where. Topics consist of units called *partitions* [1]. In other words, one or more partitions can make up a single topic. As far as what is actually implemented on the computer's disk, partitions are what Kafka works with for the most part.

> **NOTE** A single partition replica only exists on one broker and cannot be split between brokers.

Figure 2.6 shows how each partition replica leader exists on a single Kafka broker and cannot be divided smaller than that unit. Think back to our first example, the `kinaction_helloworld` topic. If you're looking for reliability and want three copies of the data, the topic itself is not one entity (or a single file) that is copied; instead, it is the various partitions that are replicated three times each.

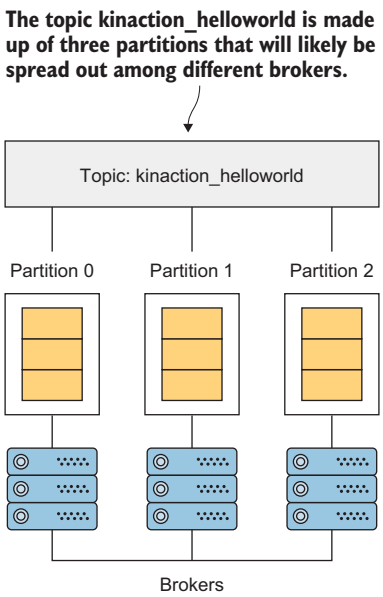

Figure 2.6 Partitions make up topics.

> **NOTE** The partition is even further broken up into segment files written on the disk drive. We will cover these files' details and their location when we talk about brokers in later chapters. Although segment files make up partitions, you will likely not interact directly with them, and this should be considered an internal implementation detail.

One of the most important concepts to understand at this point is the idea that one of the partition copies (replicas) will be what is referred to as a *leader*. For example, if you have a topic made up of three partitions and a total of three copies of each partition, every partition will have an

elected leader replica. That leader will be one of the copies of the partition, and the other two (in this case, not shown in figure 2.6) will be *followers*, which update their information from their partition replica leader [1]. Producers and consumers only read or write from the leader replica of each partition it is assigned to during scenarios where there are no exceptions or failures (also known as a "happy path" scenario). But how does your producer or consumer know which partition replica is the leader? In the event of distributed computing and random failures, that answer is often influenced with help from ZooKeeper, the next stop on our tour.

2.3.3 ZooKeeper usage

One of the oldest sources of feared added complexity in the Kafka ecosystem might be that it uses ZooKeeper. Apache ZooKeeper (http://zookeeper.apache.org/) is a distributed store that provides discovery, configuration, and synchronization services in a highly available way. In versions of Kafka since 0.9, changes were made in ZooKeeper that allowed for a consumer to have the option not to store information about how far it had consumed messages (called *offsets*). We will cover the importance of offsets in later chapters. This reduced usage did not get rid of the need for consensus and coordination in distributed systems, however.

ZooKeeper removal

To simplify the requirements of running Kafka, there was a proposal for the replacement of ZooKeeper with its own managed quorum [4]. Because this work was not yet complete at the time of publication, with an early access release version 2.8.0, ZooKeeper is still discussed in this work. Why is ZooKeeper still important?

This book covers version 2.7.1, and you are likely to see older versions in production that will use ZooKeeper for a while, until the changes are fully implemented. Also, although ZooKeeper will be replaced by the Kafka Raft Metadata mode (KRaft), the concepts of needing coordination in a distributed system are still valid, and understanding the role that ZooKeeper plays currently will, hopefully, lay the foundation of that understanding. Although Kafka provides fault tolerance and resilience, something has to provide coordination, and ZooKeeper enables that piece of the overall system. We will not cover the internals of ZooKeeper in detail but will touch on how Kafka uses it throughout the following chapters.

As you already saw, our cluster for Kafka includes more than one broker (server). To act as one correct application, these brokers need to not only communicate with each other, they also need to reach an *agreement*. Agreeing on which one is the replica leader of a partition is one example of the practical application of ZooKeeper within the Kafka ecosystem. For a real-world comparison, most of us have seen examples of clocks getting out of sync and how it becomes impossible to tell the correct time if multiple clocks are showing different times. The agreement can be challenging across separate brokers. Something is needed to keep Kafka coordinated and working in both success and failure scenarios.

One thing to note for any production use case is that ZooKeeper will be an ensemble, but we will run just one server in our local setup [5]. Figure 2.7 shows the ZooKeeper cluster and how Kafka's interaction is with the brokers and not the clients. KIP-500 refers to this usage as the "current" cluster design [4].

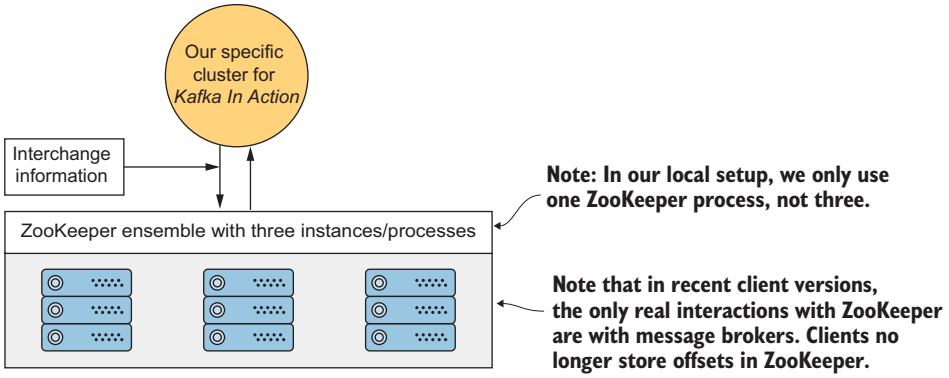

Figure 2.7 ZooKeeper interaction

> **TIP** If you are familiar with `znodes` or have experience with ZooKeeper already, one good place to start looking at the interactions inside Kafka's source code is `ZkUtils.scala`.

Knowing the fundamentals of the preceding concepts increases our ability to make a practical application with Kafka. Also, we will start to see how existing systems that use Kafka are likely to interact to complete real use cases.

2.3.4 Kafka's high-level architecture

In general, core Kafka can be thought of as Scala application processes that run on a Java virtual machine (JVM). Although noted for being able to handle millions of messages quickly, what is it about Kafka's design that makes this possible? One of Kafka's keys is its usage of the operating system's *page cache* (as shown in figure 2.8). By avoiding caching in the JVM heap, the brokers can help prevent some of the issues that large heaps may have (for example, long or frequent garbage collection pauses) [6].

Another design consideration is the access pattern of data. When new messages flood in, it is likely that the latest messages are of more interest to many consumers, which can then be served from this cache. Serving from a page cache instead of a disk is likely faster in most cases. Where there are exceptions, adding more RAM helps more of your workload to fall into the page cache.

As mentioned earlier, Kafka uses its own protocol [7]. Using an existing protocol like AMQP (Advanced Message Queuing Protocol) was noted by Kafka's creators as

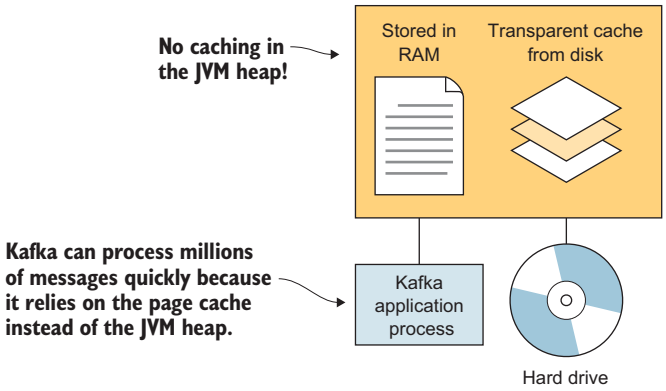

Figure 2.8 The operating system's page cache

having too large a part in the impacts on the actual implementation. For example, new fields were added to the message header to implement the exactly-once semantics of the 0.11 release. Also, that same release reworked the message format to compress messages more effectively. The protocol could change and be specific to the needs of the creators of Kafka.

We are almost at the end of our tour. There's just one more stop—brokers and the commit log.

2.3.5 The commit log

One of the core concepts to help you master Kafka's foundation is to understand the commit log. The concept is simple but powerful. This becomes clearer as you understand the significance of this design choice. To clarify, the log we are talking about is not the same as the log use case that involved aggregating the output from loggers from an application process such as the `LOGGER.error` messages in Java.

Figure 2.9 shows how simple the concept of a commit log can be as messages are added over time [8]. Although there are more mechanics that take place, such as what happens when a log file needs to come back from a broker failure, this basic concept is a crucial part of understanding Kafka. The log used in Kafka is not just a detail that is hidden in

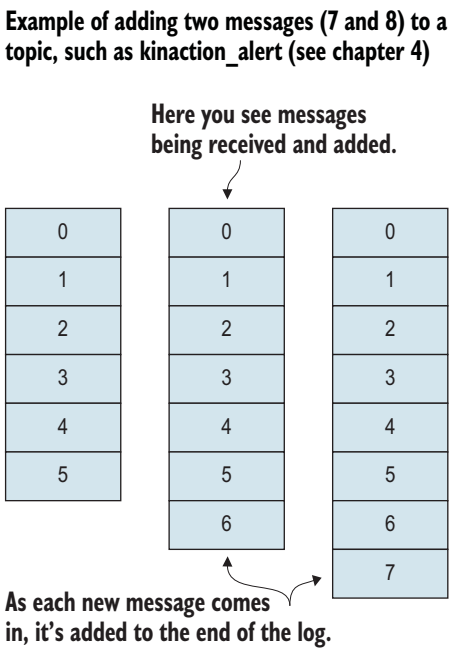

Figure 2.9 Commit log

other systems that might use something similar (like a write-ahead log for a database). It is front and center, and its users employ offsets to know where they are in that log.

What makes the commit log special is its append-only nature in which events are always added to the end. The persistence as a log itself for storage is a major part of what separates Kafka from other message brokers. Reading a message does not remove it from the system or exclude it from other consumers.

One common question then becomes, how long can I retain data in Kafka? In various companies today, it is not rare to see that after the data in Kafka commit logs hits a configurable size or time retention period, the data is often moved into a permanent store. However, this is a matter of how much disk space you need and your processing workflow. The *New York Times* has a single partition that holds less than 100 GB [9]. Kafka is made to keep its performance fast even while keeping its messages. Retention details will be covered when we talk about brokers in chapter 6. For now, just understand that log data retention can be controlled by age or size using configuration properties.

2.4 Various source code packages and what they do

Kafka is often mentioned in the titles of various APIs. There are also certain components that are described as standalone products. We are going to look at some of these to see what options we have. The packages in the following sections are APIs found in the same source code repository as Kafka core, except for ksqlDB [10].

2.4.1 Kafka Streams

Kafka Streams has grabbed a lot of attention compared to core Kafka itself. This API is found in the Kafka source code project's streams directory and is mostly written in Java. One of the sweet spots for Kafka Streams is that no separate processing cluster is needed. It is meant to be a lightweight library to use in your application. You aren't required to have cluster or resource management software like Apache Hadoop to run your workloads. However, it still has powerful features, including local state with fault tolerance, one-at-a-time message processing, and exactly-once support [10]. The more you move throughout this book, the more you will understand the foundations of how the Kafka Streams API uses the existing core of Kafka to do some exciting and powerful work.

This API was made to ensure that creating streaming applications is as easy as possible, and it provides a fluent API, similar to Java 8's Stream API (also referred to as a domain-specific language, or DSL). Kafka Streams takes the core parts of Kafka and works on top of those smaller pieces by adding stateful processing and distributed joins, for example, without much more complexity or overhead [10].

Microservice designs are also being influenced by this API. Instead of data being isolated in various applications, it is pulled into applications that can use data independently. Figure 2.10 shows a before and after view of using Kafka to

implement a microservice system (see the YouTube video, "Microservices Explained by Confluent" [11]).

Although the top part of figure 2.10 (without Kafka) relies on each application talking directly to other applications at multiple interfaces, the bottom shows an approach that uses Kafka. Using Kafka not only exposes the data to all applications without some service munging it first, but it provides a single interface for all applications to consume. The benefit of not being tied to each application directly shows how Kafka can help loosen dependencies between specific applications.

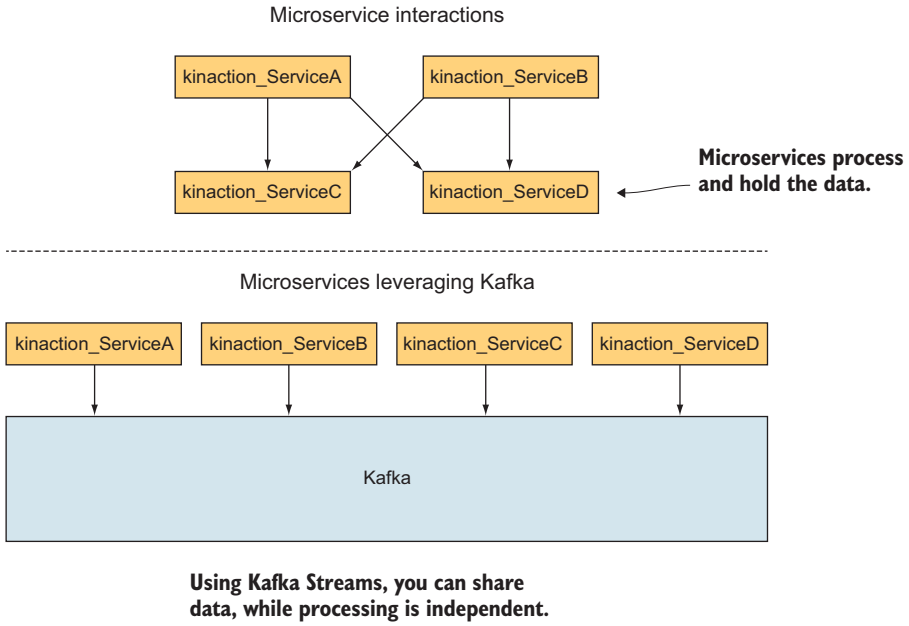

Figure 2.10 Microservice design

2.4.2 Kafka Connect

Kafka Connect is found in the core Kafka Connect folder and is also mostly written in Java. This framework was created to make integrations with other systems easier [10]. In many ways, it can be thought to help replace other tools such as the Apache project Gobblin™ and Apache Flume. If you are familiar with Flume, some of the terms used will likely seem familiar.

Source connectors are used to import data from a source into Kafka. For example, if we want to move data from MySQL® tables to Kafka's topics, we would use a Connect source to produce those messages into Kafka. On the other hand, sink connectors are used to export data from Kafka into different systems. For example, if we want messages

in some topic to be maintained long term, we would use a sink connector to consume those messages from the topic and place them somewhere like cloud storage. Figure 2.11 shows this data flow from the database to Connect and then finally to a storage location in the cloud similar to a use case talked about in the article "The Simplest Useful Kafka Connect Data Pipeline in the World…or Thereabouts – Part 1" [12].

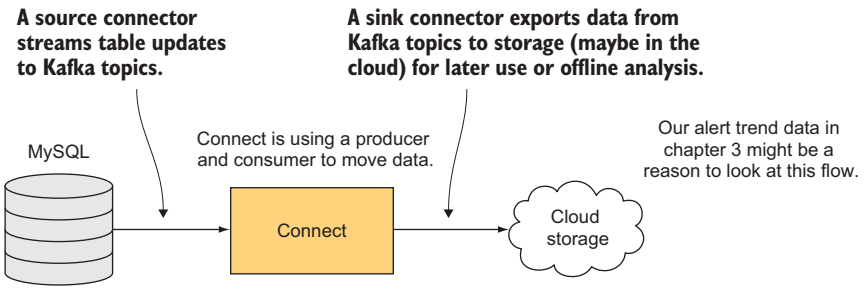

Figure 2.11 Connect use case

As a note, a direct replacement of Apache Flume features is probably not the intention or primary goal of Kafka Connect. Kafka Connect does not have an agent per Kafka node setup and is designed to integrate well with stream-processing frameworks to copy data. Overall, Kafka Connect is an excellent choice for making quick and simple data pipelines that tie together common systems.

2.4.3 AdminClient package

Kafka introduced the AdminClient API recently. Before this API, scripts and other programs that wanted to perform specific administrative actions would either have to run shell scripts (which Kafka provides) or invoke internal classes often used by those shell scripts. This API is part of the kafka-clients.jar file, which is a different JAR than the other APIs discussed previously. This interface is a great tool that will come in handy the more involved we become with Kafka's administration [10]. This tool also uses a similar configuration that producers and consumers use. The source code can be found in the org/apache/kafka/clients/admin package.

2.4.4 ksqlDB

In late 2017, Confluent released a developer preview of a new SQL engine for Kafka that was called KSQL before being renamed to ksqlDB. This allowed developers and data analysts who used mostly SQL for data analysis to leverage streams by using the interface they have known for years. Although the syntax might be somewhat familiar, there are still significant differences.

Most queries that relational database users are familiar with involve on-demand or one-time queries that include lookups. The mindset shift to a continuous query over a data stream is a significant shift and a new viewpoint for developers. As with the Kafka Streams API, ksqlDB is making it easier to use the power of continuous data flows. Although the interface for data engineers will be a familiar SQL-like grammar, the idea that queries are continuously running and updating is where use cases like dashboards on service outages would likely replace applications that once used point-in-time SELECT statements.

2.5 Confluent clients

Due to Kafka's popularity, the choice of which language to interact with Kafka usually isn't a problem. For our exercises and examples, we will use the Java clients created by the core Kafka project itself. There are many other clients supported by Confluent as well [13].

Since all clients are not the same feature-wise, Confluent provides a matrix of supported features by programming language at the following site to help you out: https://docs.confluent.io/ current/clients/index.html. As a side note, taking a look at other open source clients can help you develop your own client or even help you learn a new language.

Because using a client is the most likely way you will interact with Kafka in your applications, let's look at using the Java client (listing 2.9). We will do the same produce-and-consume process that we did when using the command line earlier. With a bit of additional boilerplate code (not listed here to focus on the Kafka-specific parts only), you can run this code in a Java main method to produce a message.

Listing 2.9 Java client producer

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

    Properties kaProperties =
      new Properties();
    kaProperties.put("bootstrap.servers",
      "localhost:9092,localhost:9093,localhost:9094");

    kaProperties.put("key.serializer",
      "org.apache.kafka.common.serialization.StringSerializer");
    kaProperties.put("value.serializer",
      "org.apache.kafka.common.serialization.StringSerializer");

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

      ProducerRecord<String, String> producerRecord =
        new ProducerRecord<>("kinaction_helloworld",
```

The producer takes a map of name-value items to configure its various options.

This property can take a list of Kafka brokers.

Tells the message's key and value what format to serialize

Creates a producer instance. Producers implement the closable interface that's closed automatically by the Java runtime.

```
        null, "hello world again!");

      producer.send(producerRecord);
    }
  }
}
```

Represents our message

Sends the record to the Kafka broker

The code in listing 2.9 is a simple producer. The first step to create a producer involves setting up configuration properties. The properties are set in a way that anyone who has used a map will be comfortable using.

The `bootstrap.servers` parameter is one essential configuration item, and its purpose may not be apparent at first glance. This is a list of your Kafka brokers. The list does not have to be every server you have, though, because after the client connects, it will find the information about the rest of the cluster's brokers and not depend on that list.

The `key.serializer` and `value.serializer` parameters are also something to take note of in development. We need to provide a class that will serialize the data as it moves into Kafka. Keys and values do not have to use the same serializer.

Figure 2.12 displays the flow that happens when a producer sends a message. The producer we created takes in the configuration properties as an argument in the constructor we used. With this producer, we can now send messages. The `ProducerRecord` contains the actual input that we want to send. In our examples, `kinaction_helloworld` is the name of the topic that we sent. The next fields are the message key followed by the message value. We will discuss keys more in chapter 4, but it is enough to know that these can, indeed, be a null value, which makes our current example less complicated.

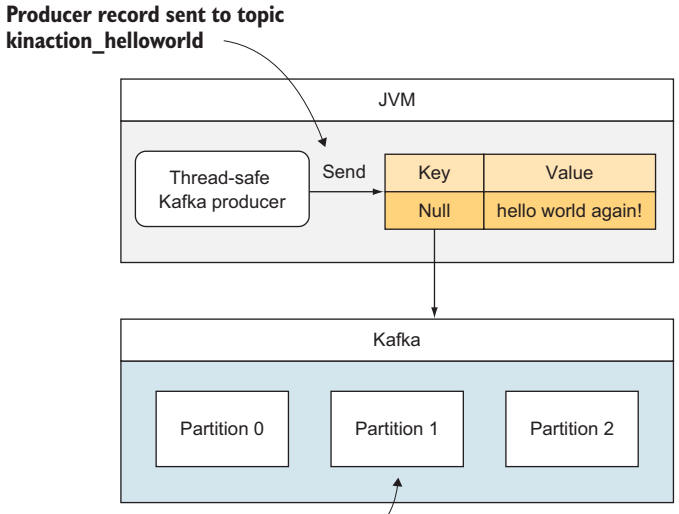

Figure 2.12 Producer flow

The message we send as the last argument is something different from the first message we sent with our console producer. Do you know why we want to make sure the message is different? We are working with the same topic with both producers, and because we have a new consumer, we should be retrieving the old message we produced before in our Java client-initiated message. Once our message is ready, we asynchronously send it using the producer. In this case, because we are only sending one message, we close the producer, which waits until previously sent requests complete and then shuts down gracefully.

Before running these Java client examples, we'll need to make sure we have the entry in the following listing in our pom.xml file [14]. We will use Apache Maven™ in all of the examples in this book.

Listing 2.10 Java client POM entry

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.7.1</version>
</dependency>
```

Now that we have created a new message, let's use our Java client as in the following listing to create a consumer that can see the message. We can run the code inside a Java `main` method and terminate the program after we are done reading messages.

Listing 2.11 Java client consumer

```
public class HelloWorldConsumer {

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

  private volatile boolean keepConsuming = true;

  public static void main(String[] args) {
    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers",
      "localhost:9092,localhost:9093,localhost:9094");
    kaProperties.put("group.id", "kinaction_helloconsumer");
    kaProperties.put("enable.auto.commit", "true");
    kaProperties.put("auto.commit.interval.ms", "1000");
    kaProperties.put("key.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer");
    kaProperties.put("value.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer");

    HelloWorldConsumer helloWorldConsumer = new HelloWorldConsumer();
    helloWorldConsumer.consume(kaProperties);
    Runtime.getRuntime().
      addShutdownHook(new Thread(helloWorldConsumer::shutdown));
  }
```

Properties are set the same way as producers.

```
  private void consume(Properties kaProperties) {
    try (KafkaConsumer<String, String> consumer =
      new KafkaConsumer<>(kaProperties)) {
      consumer.subscribe(
        List.of(
          "kinaction_helloworld"
        )
      );

      while (keepConsuming) {
        ConsumerRecords<String, String> records =
          consumer.poll(Duration.ofMillis(250));
        for (ConsumerRecord<String, String> record :
          records) {
          log.info("kinaction_info offset = {}, kinaction_value = {}",
            record.offset(), record.value());
        }
      }
    }
  }

  private void shutdown() {
    keepConsuming = false;
  }
}
```

The consumer tells Kafka what topics it's interested in.

Polls for new messages as they come in

To see the result, prints each record that we consume to the console

One thing that jumps out is that we have an infinite loop in listing 2.11. It seems weird to do that on purpose, but we want to handle an infinite stream of data. The consumer is similar to the producer in taking a map of properties to create a consumer. However, unlike the producer, the Java consumer client is not thread safe [15]. We will need to take that into account as we scale past one consumer in later sections. Our code is responsible for ensuring that any access is synchronized: one simple option is having only one consumer per Java thread. Also, whereas we told the producer where to send the message, we now have the consumer subscribe to the topics it wants. A `subscribe` command can subscribe to more than one topic at a time.

One of the most important sections to note in listing 2.11 is the `poll` call on the consumer. This is what is actively trying to bring messages to our application. No messages, one message, or many messages can all come back with a single poll, so it is important to note that our logic should account for more than one result with each poll call.

Finally, we can Ctrl-C the consumer program when we retrieve the test messages and be done for now. As a note, these examples rely on many configuration properties that are enabled by default. We will have a chance to dig into them more in later chapters.

2.6 Stream processing and terminology

We are not going to challenge distributed systems theories or certain definitions that could have various meanings, but rather look at how Kafka works. As you start to think of applying Kafka to your work, you will be presented with the following terms and can, hopefully, use the following descriptions as a lens through which to view your processing mindset.

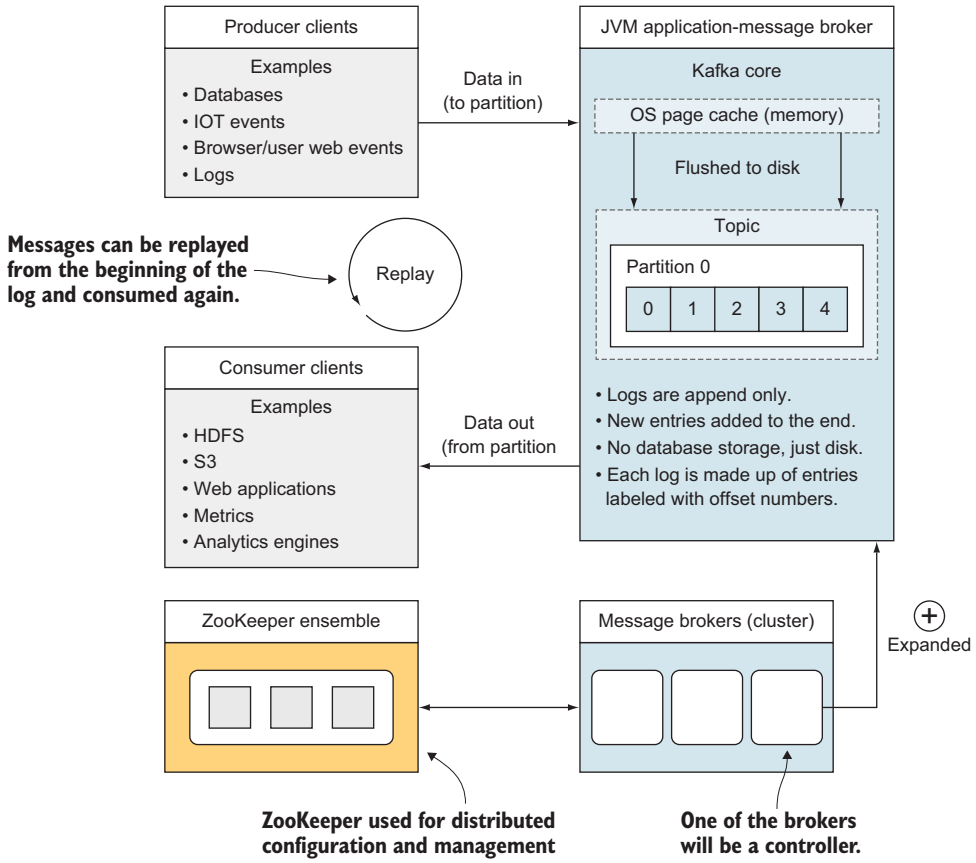

Figure 2.13 Kafka overview

Figure 2.13 provides a high-level view of what Kafka does. Kafka has many moving parts that depend on data coming into and out of its core to provide value to its users. Producers send data into Kafka, which works as a distributed system for reliability and scale, with logs, which are the basis for storage. Once data is inside the Kafka ecosystem, consumers can help users utilize that data in their other applications and use cases. Our brokers make up the cluster and coordinate with a ZooKeeper cluster to maintain metadata. Because Kafka stores data on disk, the ability to replay data in case of an application failure is also part of Kafka's feature set. These attributes allow Kafka to become the foundation of powerful stream-processing applications.

2.6.1 Stream processing

Stream processing seems to have various definitions throughout various projects. The core principle of streaming data is that data will keep arriving and will not end [16]. Also, your code should be processing this data all the time and not wait for a request

or time frame with which to run. As we saw earlier, an infinite loop in our code hinted at this constant flow of data that does not have a defined endpoint.

This approach does not batch data and then process it in groups. The idea of a nightly or monthly run is also not a part of this workflow. If you think of a never-ending waterfall, the same principles apply. Sometimes there is a massive amount of data to transit and sometimes not that much, but it continuously flows between destinations.

Figure 2.14 shows that the Kafka Streams API depends on core Kafka. While event messages continue to come into the cluster, a consumer application can provide the end user with updated information continuously rather than wait for a query to pull a static snapshot of the events. No more refreshing the web page after 5 minutes for users to see the latest events!

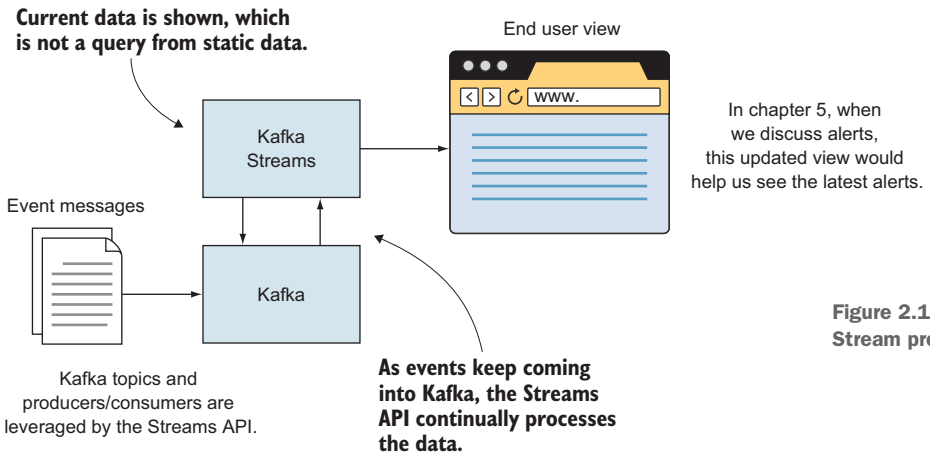

Figure 2.14 Stream process

2.6.2 What exactly-once means

One of the most exciting and maybe most discussed features in Kafka is its exactly-once semantics. This book will not discuss the theory behind those views; however, we will touch on what these semantics mean for Kafka's everyday usage.

One important thing to note is that the easiest way to maintain exactly-once is to stay within Kafka's walls (and topics). Having a closed system that can be completed as a transaction is why using the Streams API is one of the easiest paths to exactly-once. Various Kafka Connect connectors also support exactly-once and are great examples of bringing data out of Kafka because it won't always be the final endpoint for all data in every scenario.

Summary

- Messages represent your data in Kafka. Kafka's cluster of brokers handles this data and interacts with outside systems and clients.

- Kafka's use of a commit log helps in understanding the system overall.
- Messages appended to the end of a log frame how data is stored and how it can be used again. By being able to start at the beginning of the log, applications can reprocess data in a specific order to fulfill different use cases.
- Producers are clients that help move data into the Kafka ecosystem. Populating existing information from other data sources like databases into Kafka can help expose data that was once siloed in systems that provided a data interface for other applications.
- Consumer clients retrieve messages from Kafka. Many consumers can read the same data at the same time. The ability for separate consumers to start reading at various positions also shows the flexibility of consumption possible from Kafka topics.
- Continuously flowing data between destinations with Kafka can help us redesign systems that used to be limited to batch or time-delayed workflows.

References

1. "Main Concepts and Terminology." Apache Software Foundation (n.d.). https://kafka.apache.org/documentation.html#intro_concepts_and_terms (accessed May 22, 2019).
2. "Apache Kafka Quickstart." Apache Software Foundation (2017). https://kafka.apache.org/quickstart (accessed July 15, 2020).
3. B. Kernighan and D. Ritchie. *The C Programming Language*, 1st ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1978.
4. KIP-500: "Replace ZooKeeper with a Self-Managed Metadata Quorum." Wiki for Apache Kafka. Apache Software Foundation (July 09, 2020). https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum (accessed August 22, 2020).
5. "ZooKeeper Administrator's Guide." Apache Software Foundation. (n.d.). https://zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html (accessed June 10, 2020).
6. "Kafka Design: Persistence." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html#persistence (accessed November 19, 2020).
7. "A Guide To The Kafka Protocol: Some Common Philosophical Questions." Wiki for Apache Kafka. Apache Software Foundation (n.d.). https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-SomeCommonPhilosophicalQuestions (accessed August 21, 2019).
8. "Documentation: Topics and Logs." Apache Software Foundation (n.d.). https://kafka.apache.org/23/documentation.html#intro_topics (accessed May 25, 2020).

9 B. Svingen. "Publishing with Apache Kafka at The New York Times." Confluent blog (September 6, 2017). https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/ (accessed September 25, 2018).

10 "Documentation: Kafka APIs." Apache Software Foundation (n.d.). https://kafka.apache.org/documentation.html#intro_apis (accessed June 15, 2021).

11 "Microservices Explained by Confluent." Confluent. Web presentation (August 23, 2017). https://youtu.be/aWI7iU36qv0 (accessed August 9, 2021).

12 R. Moffatt. "The Simplest Useful Kafka Connect Data Pipeline in the World…or Thereabouts – Part 1." Confluent blog (August 11, 2017). https://www.confluent.io/blog/simplest-useful-kafka-connect-data-pipeline-world-thereabouts-part-1/ (accessed December 17, 2017).

13 "Kafka Clients." Confluent documentation (n.d.). https://docs.confluent.io/current/clients/index.html (accessed June 15, 2020).

14 "Kafka Java Client." Confluent documentation (n.d.). https://docs.confluent.io/clients-kafka-java/current/overview.html (accessed June 21, 2021).

15 "Class KafkaConsumer<K,V>." Apache Software Foundation (November 09, 2019). https://kafka.apache.org/24/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html (accessed November 20, 2019).

16 "Streams Concepts." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/streams/concepts.html (accessed June 17, 2020).