Kafka IN ACTION

Dylan D. Scott
Viktor Gamov
Dave Klein

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kafka in Action
Version 16

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Kafka in Action*!

For me, reading has always been part of my preferred learning style. Part of the nostalgia is remembering the first practical programming book I ever really read: Elements of Programming with Perl by Andrew L Johnson. The content was something that registered with me. I could follow along and it was a joy to work through.

I hope to capture some of that practical content in regards to working with Apache Kafka. The excitement of learning something new that I had when reading Elements of Programming with Perl was the same way I felt when I started to work with Kafka for the first time. Kafka was unlike any other message broker or enterprise service bus (ESB) that I had used at the time. The speed to get started developing producers and consumers, the ability to reprocess data, and independent consumers moving at their own speeds without removing the data from other consumer applications were techniques that solved pain points I had seen in my own past development and impressed me when I started looking at Kafka.

I see Kafka as changing the standard for data platforms; it can help move batch and ETL workflows to near real-time data feeds. Since this foundation is likely a shift from past data architectures that many enterprise users are familiar with, I wanted to take a user from no prior knowledge of Kafka to the ability to work with Kafka producers and consumers and also be able to perform basic Kafka developer and admin tasks. By the end of this book, I also hope you will feel comfortable digging into more advanced Kafka topics such as cluster monitoring, metrics, and multi-site data replication with your new core Kafka knowledge.

Please remember, these chapters are still works in progress, and will likely change and hopefully get even better by the time the book is complete. I never used to understand how simple mistakes could make it into books, but seeing all of the edits, revisions, and deadlines involved, typos can still make their way into the content and source code. I appreciate your notes for corrections and patience if you run into anything. Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook's Discussion Forum section for Kafka in Action.

I really do hope to make this a useful book and appreciate feedback that you think could improve future versions as well.

Thanks again for your interest and for purchasing the MEAP!

—Dylan Scott

brief contents

1 Introduction to Kafka

This chapter covers

- Introducing why you might want to use Kafka
- Common myths in relation to Big Data and message systems
- Understanding Real World Use Cases where Kafka helps power messaging, streaming, and IoT data processing

As many companies are facing a world full of data being produced from every angle, they are often presented with the fact that legacy systems might not be the best option moving forward. One of the foundational pieces of new data infrastructures that has taken over the IT landscape has been Apache Kafka®. Kafka is changing the standards for data platforms. It is leading the way to move from extract, transform, load (ETL) and batch workflows, in which work was often held and processed in bulk at one various pre-defined time of day, to near real-time data feed [1]. Batch processing, which was once the standard workhorse of enterprise data processing, might not be something to turn back to after seeing the powerful feature set that Kafka provides. In fact, it might not be able to handle the growing snowball of data rolling toward enterprises of all sizes unless something new is approached. With so much data, systems can get easily overloaded with data. Legacy systems might be faced with nightly processing windows that run into the next day. To keep up with this ever-constant stream of data, some with evolving data, processing this stream of information as it happens is a way to keep up-to-date and current on the system's state.

Kafka is touching many of the newest and the most practical trends in today's IT fields and making its way into many users' daily work. For example, Kafka has already made its way into the microservice design and the Internet of Things (IoT). As a de-facto technology for more and more companies, this topic is not only for super-geeks or alpha-chasers.

Let's start looking at these features by introducing Kafka itself and understand more about the face of modern-day streaming platforms.

1.1 What is Kafka?

The Apache Kafka® site (kafka.apache.org/intro) defines it as a distributed streaming platform that has three main capabilities:

- Reading and writing records like a message queue.
- Store records with fault-tolerance.
- Process streams as they occur [2].

Readers who are not as familiar with queues or message brokers in their daily work might help discuss the general purpose and flow of such a system. As a generalization, a core piece of Kafka can be thought of as providing the IT equivalent of a receiver that sits in a home entertainment system.

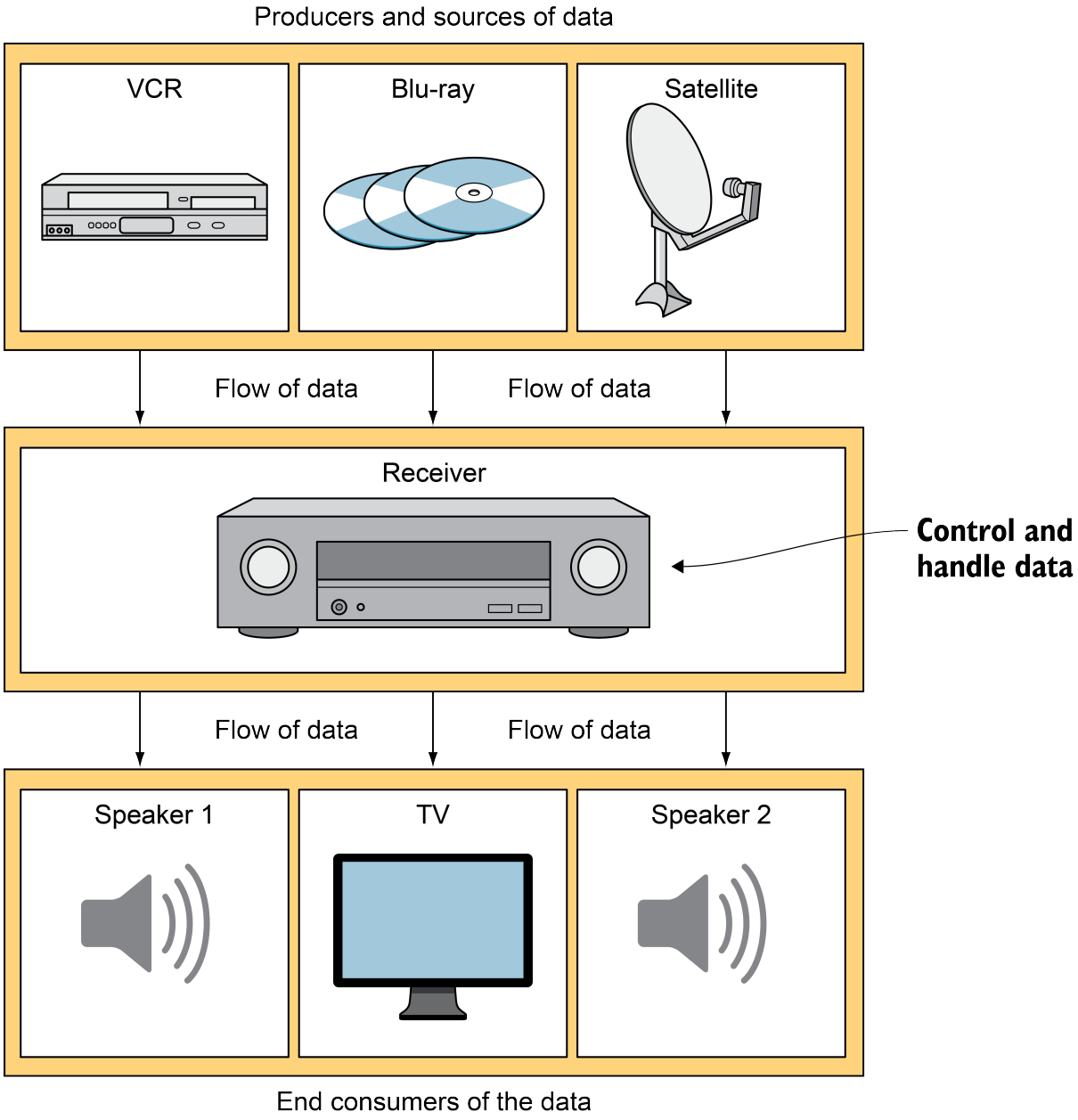

Figure 1.1 Receiver overview

As shown in Figure 1.1, digital satellite, cable, and Blu-ray™ players can connect to this central receiver. Simplifying, of course, you can think of those individual pieces as regularly sending data in a format that they know about. And unless you have some issues, that flow of data can be thought as near constant while the movie or cd is playing. The receiver deals with this constant stream of data and is able to convert it into a usable format for the external devices attached to the other end, i.e., the receiver sends the video onto your television and audio to a decoder as well as on to speakers.

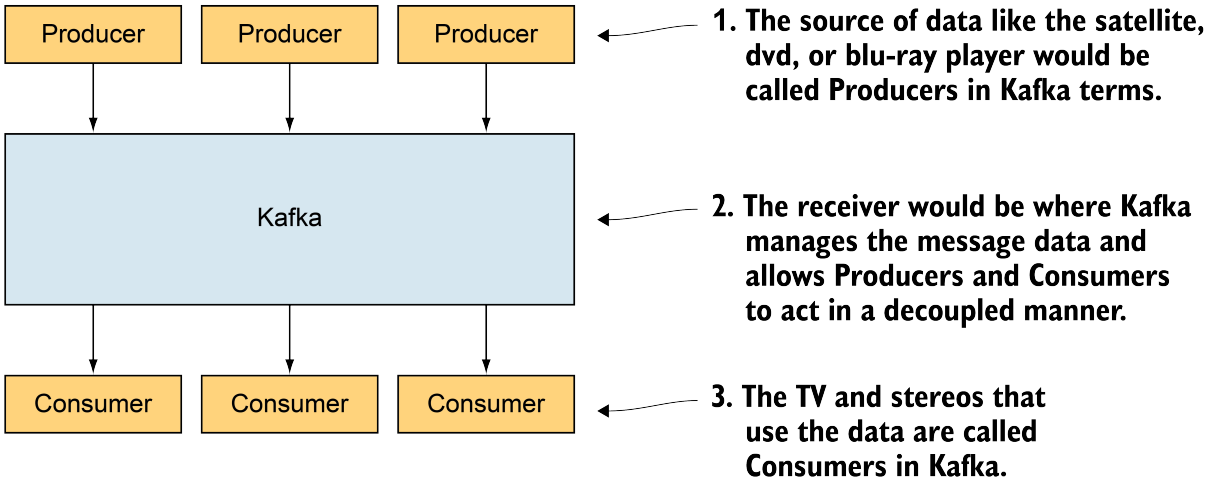

Figure 1.2 Kafka flow overview

So what does this have to do with Kafka exactly? Kafka includes clients to interface with the other systems. One such client type is called a Producer that can send multiple data streams to the Kafka brokers. The Kafka brokers serve a similar function as the receiver in Figure 1.1. It also includes Consumers, clients that can read data from the brokers and process it. Data does not only have to be limited to a single destination. The Producers and Consumers are completely decoupled, allowing each client to work independently. We'll dig into the details of how this is done in later chapters.

As do other messaging platforms, Kafka acts, in reductionist terms, like a middle man to data coming into the system (from producers) and out of the system (consumers). The loose coupling can be achieved by allowing this separation between the producer and end-user of the message. The producer can send whatever messages it wants and have no clue about if anyone is subscribed.

Further, Kafka also has various ways that it can deliver messages to fit your business case. Kafka message delivery can take at least the following three delivery methods [3]:

- At least once semantics - A message will be resent if needed until it is acknowledged.
- At most once semantics - A message will only be sent once and not resent on failure.
- Exactly once semantics - A message will only be seen once by the consumer of the message.

Let's dig into what those messaging options mean.

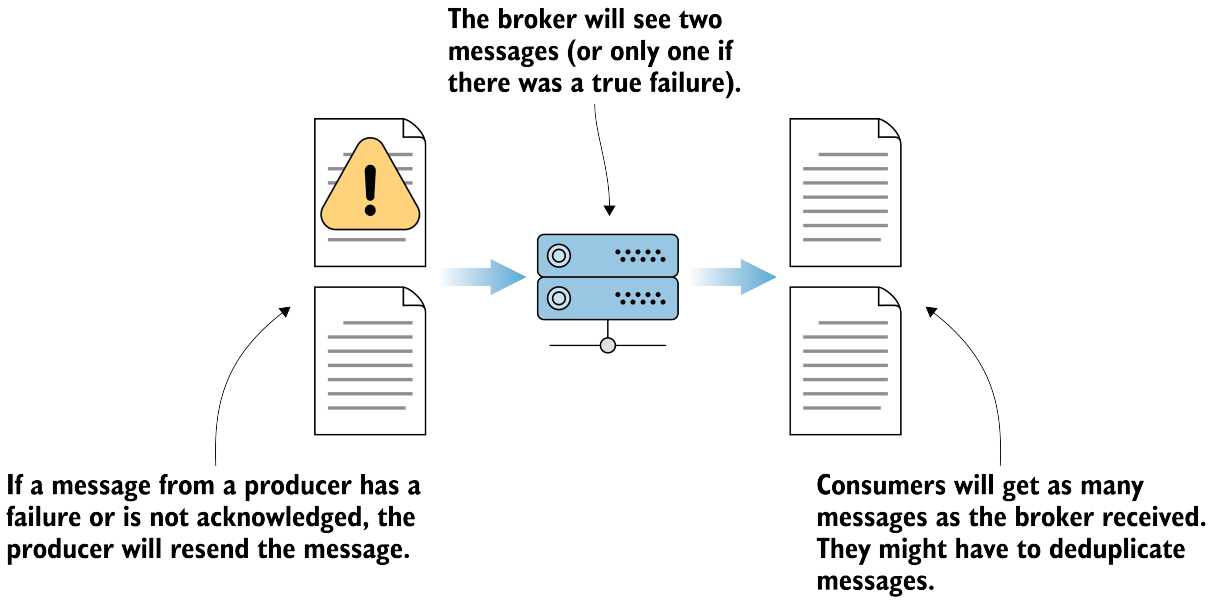

Figure 1.3 At Least Once message flow semantics

Kafka's default guarantee is "at least once" semantics. In this case, Kafka can be configured to allow a producer of messages to send the same message more than once and have it written to the brokers. When a message has not received a guarantee that it was written to the broker, the producer can send the message again to try again. For those cases where you can't miss a message, say that someone has paid an invoice. This guarantee might take some filtering on the consumer end but is one of the safest delivery methods.

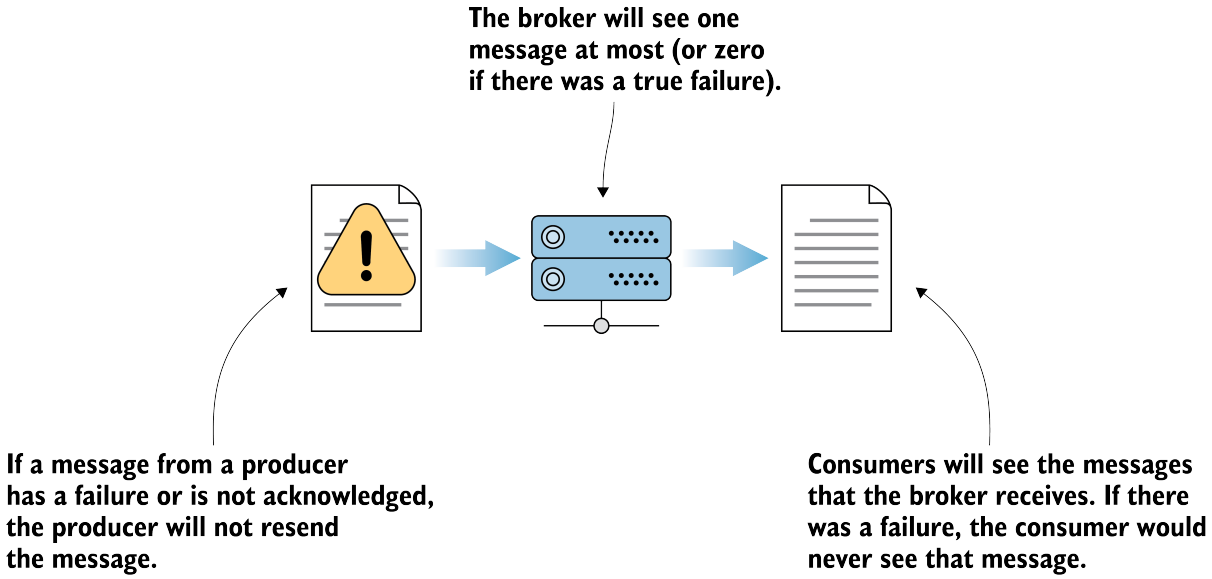

Figure 1.4 "At Most Once" message flow semantics

"At most once" semantics is when a producer of messages might send a message once and never retries. In the event of a failure, the producer moves on and never attempts to send again. Why would someone ever be okay with losing a message? If a popular website is tracking page views for visitors, it might be okay with missing a few page view events out of the millions they do

process per day. Keeping the system performing and not waiting on acknowledgments might outweigh any cost of lost data.

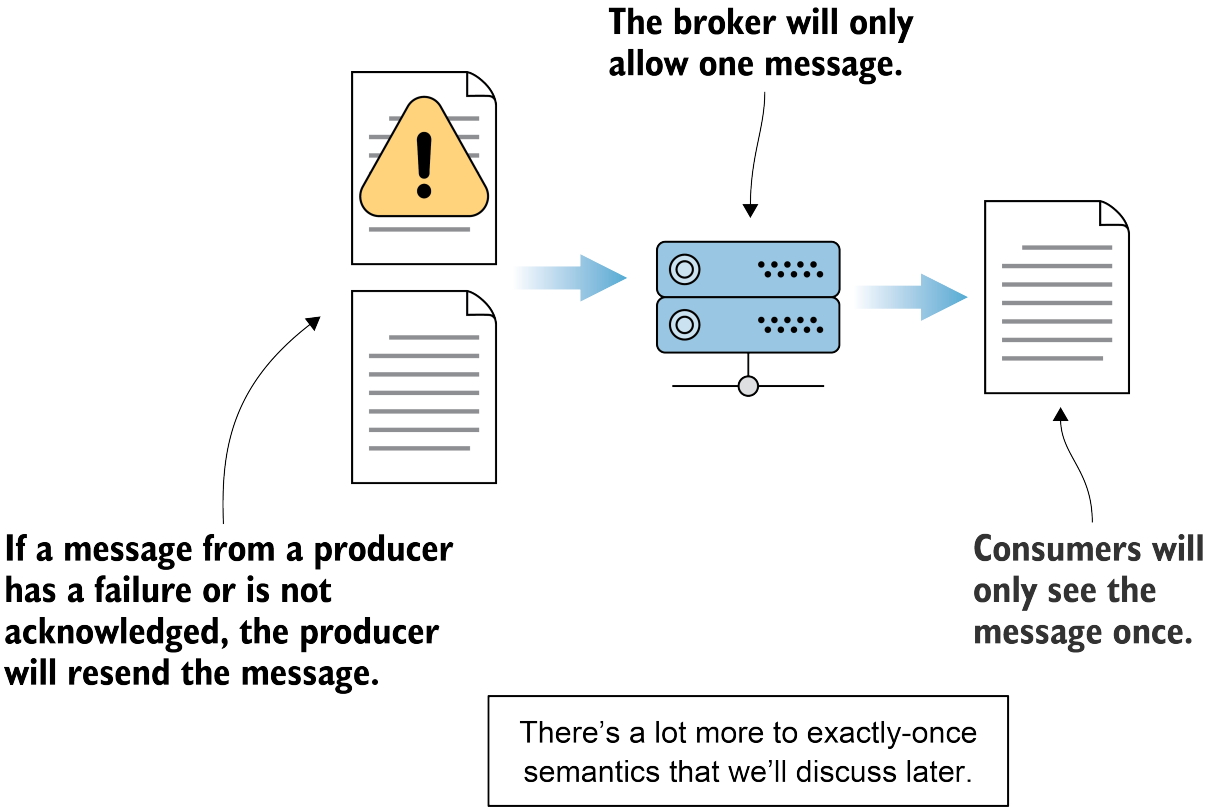

Figure 1.5 Exactly Once message flow semantics

"Exactly once" semantics (EOS), added to the Kafka feature set in version `0.11.0`, generated a lot of mixed discussion with its release. On the one hand, exactly-once semantics are ideal for a lot of use-cases. This guarantee makes logic to remove duplicate messages a thing of the past. Most developers appreciate sending one message in and receiving that same message on the consuming side as well. Another discussion that followed the release was a debate on if exactly once was even possible. While this goes into deeper computer science, topics-it is helpful to be aware of how Kafka defines their EOS feature [4]. In the context of a Kafka system, if a producer sends a message more than once, it would still be delivered only once to the end consumer. EOS has touchpoints at all Kafka layers, from producers, topics, brokers, and consumers, and will be tackled as we move along our discussion later in this book.

Besides various delivery options, another common message broker benefit is if the consuming application is down due to errors or maintenance, the producer does not need to wait on the consumer to handle the message. When consumers start to come back and process data, they should pick up where they left off and not drop messages.

1.2 Kafka Usage

With many traditional companies facing challenges of becoming more and more technical and software-driven, one of the questions is how they will be prepared for the future. Kafka is noted for being a high-performance message-delivery workhorse that features replication and fault-tolerance as a default.

Millions of messages per second were possible in production settings, all with a tool that was not at its 1.0 version release (which finally occurred in October of 2017). LinkedIn's use-cases include various features, including user activity events, logging, and newsfeed delivery [5]. Their enormous data processing needs are handled with their production usage of Kafka. These examples show the potential of what Kafka can do for even massive data use cases at scale.

However, besides these headline-grabbing facts, why would users want to start looking at Kafka?

1.2.1 Kafka for the Developer

Why would a software developer be interested in Kafka? Kafka usage is exploding, and the developer demand isn't being met.[6]. A shift in traditional data process thinking is needed. Various shared experiences or past pain points can help developers see why Kafka could be an appealing step forward in their data architectures.

One of the various on-ramps for newer developers to Kafka is applying things they know to help them with the unknown. For Oracle® Java developers use Spring concepts, and dependency injection (DI) Spring® for Apache Kafka® (projects.spring.io/spring-kafka) has already been through a couple of major release versions. Supporting projects, as well as Kafka itself, has a growing tool ecosystem of its own.

As a common developer experience, most programmers have likely confronted the challenges of coupling. For example, you want to make a change but you might have many other applications directly tied to it. Or you start to unit test and see a large number of mocks you have to create. Kafka, when applied thoughtfully, can help in these situations.

Take, for example an HR system that employees would use to submit paid vacation. If you are used to a create, read, update, and delete (CRUD) system, the submission of time off would likely be processed by not only payroll but also project burndown charts for forecasting work. So do you tie the two applications together? What if the payroll system was down? Should that impact the availability of the forecasting tooling? With Kafka, we will see the benefits of being able to decouple some of the applications that we have tied together in older designs. We look at maturing our data model and look more in-depth at this in Chapter 11. Kafka could be put in the middle of the workflow [7]. Your interface to data becomes Kafka instead of various APIs and databases.

Some will say that there are better simple solutions. What about using ETL to at least load the data into its databases for each application? That would only be one interface per application and easy, right? But what if the initial source of data was corrupted or updated? How often do you look for updates and allow for lag or consistency? And do those copies ever get out of date or diverge so far from the source that it would be hard to run that flow over and get the same results? What is the source of truth? Kafka can help avoid these issues.

Another interesting topic that might add credibility to the use of Kafka is how much it dog-foods itself. For example, when we dig into Consumers in Chapter 5, we will see how Kafka uses topics under the hood to manage consumer's offsets. After the release of 0.11, exactly-once semantics for Kafka also uses internal topics leveraged by the Transaction Coordination. The ability to have many consumers of the data use the same message is better than choosing your own adventure game, as all outcomes are possible.

Another developer question might be: Why not learn Kafka Streams, ksqlDB, Apache Spark™ Streaming, or other platforms and skip learning about core Kafka? For one, the number of applications that use Kafka under the covers is indeed impressive. While abstraction layers are often nice to have (and sometimes close to required with so many moving parts), I believe that Kafka itself is worth learning. There is a difference in knowing that Kafka is a channel option for Apache Flume™ and understanding what all of the config options mean. And while Kafka Streams can simplify examples you might see in this book, it is interesting to note how successful Kafka was before Kafka Streams were even introduced. The base is fundamental and will hopefully help you see why Kafka is used in some applications and what is happening under the hood. If you want to become an expert in streaming, it will likely be important to know the underlying distributed parts of your application and all of the knobs you can turn to tune your applications.

From a purely technical viewpoint, there are exciting computer science topics applied in practical ways. Perhaps the most talked about is the notion of a distributed commit log which we will discuss in-depth in Chapter 2. And a personal favorite, Hierarchical Timing Wheels [8]. These examples show you how Kafka handles a problem of scale by applying an interesting data structure to solve a practical problem.

I would also note that the fact that it's open-source is a positive for digging into the source code and having documentation and examples just by searching the internet. Resources are not just limited to internal knowledge alone based on a specific workplace.

1.2.2 Explaining Kafka to your manager

As often the case, sometimes members of the C-suite can hear the term 'Kafka' and might be more confused by the name than care about what it does. As conferences have taken to making a case for attendees to help convince their boss about attending conferences, it might be nice to explain the value found in this product. Also, it is good to take a step back and look at the larger picture of what the real value add is for this tool.

One of Kafka's most important features is the ability to take volumes of data and make it available for use by various business units. A data backbone that would make information coming into the enterprise available to all the multiple areas would allow flexibility and openness on a company-wide scale. Nothing is prescribed, but it is a potential outcome. Most executives will also know that more data than ever is flooding in, and they want insights as fast as possible. Rather than pay for data to molder on disk, the value can be derived from most of it as it arrives. Kafka is one way to move away from a daily batch job that limited how quickly that data could be turned into value. Fast Data seems to be a newer term that hints that real value focuses on something different from the promises of Big Data alone.

Running on the JVM should be a familiar and comfortable place for many enterprise development shops. The ability to run on-premise is a crucial driver for some whose data requires on-site oversight. And the Cloud and managed platforms are an option as well. It can grow horizontally and not depend alone on vertical scaling that might eventually reach an expensive peak.

And maybe one of the most important reasons to learn about Kafka is to see how startups and other disrupters in their industry can overcome the once prohibitive cost of computing power. Instead of relying on a bigger and beefier server or a mainframe that can cost millions of dollars, distributed applications and architectures put competitors quickly within reach with hopefully less financial outlay.

1.3 Kafka Myths

When you start to learn any new technology, it is often natural for the learner to try to map their existing knowledge to new concepts. While that technique can be used in learning Kafka, I wanted to note some of the most common misconceptions that I have run into in my work so far.

1.3.1 Kafka only works with Hadoop

As mentioned, Kafka is a powerful tool that is often used in various situations. However, it seemed to appear on radars when used in the Hadoop ecosystem and might have first appeared as a tool as part of a Cloudera™ or Hortonworks™ suite. It isn't uncommon to hear the myth that Kafka only works on Hadoop. What could cause this confusion? One of the causes is likely the various tools that use Kafka as part of their products. Spark Streaming and Flume are examples of tools that use Kafka (or did at one point). The dependency (depending on the version of Kafka) on Apache ZooKeeper™ is also a tool that is often found in Hadoop clusters and might tie Kafka further to this myth.

One other fundamental myth that often appears is that Kafka requires the Hadoop Filesystem - HDFS. That is not the case. Once we start to dig into how Kafka works, we will see the speed Kafka can process events with techniques that would likely be slowed down with a Node Manager in the middle of the process. Also, the replications of blocks, usually a part of HDFS, are not done in the same way. One such example is that in Kafka, replicas are not recovered by default. While both products use the ideas of replication in different ways, the durability that is marketed for Kafka topics might be easy to group under the Hadoop theme of expecting failure as a default (and thus planning for overcoming it) and are similar overall goals between Hadoop and Kafka.

1.3.2 Kafka is the same as other message brokers

Another big myth is that Kafka is just another message broker. Direct comparisons of the features of various tools such as RabbitMQ™ by Pivotal or IBM MQSeries™ to Kafka often have asterisks (or fine print) attached and are not always fair to the best use cases of each. Some tools over time have gained or will gain new features just as Kafka has added the "Exactly-Once" semantics. And default configurations can be changed to mirror features closer to other tools in the same space.

In general, some of the most exciting and different features are the following that we will dig into below:

- The ability to replay messages by default
- Parallel processing of data

Kafka was designed to have multiple consumers. What that means is one application reading a message off of the message brokers doesn't remove it from other applications that might want to consume it as well. One effect of this is that a consumer who has already seen that message can again choose to read that (and other messages). With some architecture models such as Lambda (discussed in chapter 8), programmer mistakes are expected just as much as hardware failure. Imagine consuming millions of messages, and you forgot to use a specific field from the original message. In some queues, that message would have been removed or sent to a duplicate or replay

location. However, Kafka provides a way for Consumers to seek specific points and read messages (with a few constraints) again by just seeking an earlier position on the topic.

As touched on briefly above, Kafka allows for parallel processing of data and can have multiple consumers on the same topic. Besides, Kafka has the concept of consumers being part of a Consumer Group, which will be covered in-depth in Chapter 5. Membership in a group determines which consumers get which messages and what work has been done across that group of consumers. Consumer groups act independently of any other groups and allow for multiple applications to consume messages at their own pace with as many consumers as they require working together. Processing can happen in various ways: Consumption by many consumers working on one application and consumption by many applications.

No matter what other message brokers support, let's now focus on the robust use cases that have made Kafka one of the default options developers turn to for getting work done.

1.4 Kafka in the Real World

Applying Kafka to practical use is the core aim of this book. One of the things to note with Kafka is that it's hard to say it does one specific function well; it has many specific uses to excel. While we have some basic ideas to grasp first, it might be helpful to discuss at a high-level some of the cases that Kafka has already been noted for use in real-world use cases. The Apache Kafka® site lists general areas used for Kafka in the real world [9].

1.4.1 Early Examples

Some users' first experience with Kafka (as was mine) was using it as a messaging tool. Personally, after years of using other tools like IBM WebSphere MQ (formerly MQ Series), Kafka (which was around version 0.8.3 at the time) seemed simple to use to get messages from point A to point B. It forgoes using popular protocols and standards like the Extensible Messaging and Presence Protocol (XMPP), Java Message Service (JMS) API (now part of Jakarta EE), or the OASIS® Advanced Message Queuing Protocol (AMQP) in favor of a custom TCP binary protocol.

We will dig in and see some complex uses later. As an end-user developing with a Kafka client, most of the details are in the configuration, and the logic becomes relatively straightforward, ie. I want to place a message on this topic.

Having a durable channel for sending messages is also why Kafka is used. Oftentimes, memory storage of data in RAM only will not be enough to protect your data; if that server dies, the messages are not persisted across a reboot. High availability and persistent storage are built into Kafka from the start. Apache Flume provides a Kafka Channel option since the replication and

availability allow Flume events to be made immediately available to other sinks if a Flume agent (or the server it is running on) crashes [10]. Kafka enables robust applications to be built and helps handle the expected failures that distributed applications are bound to run into at some point.

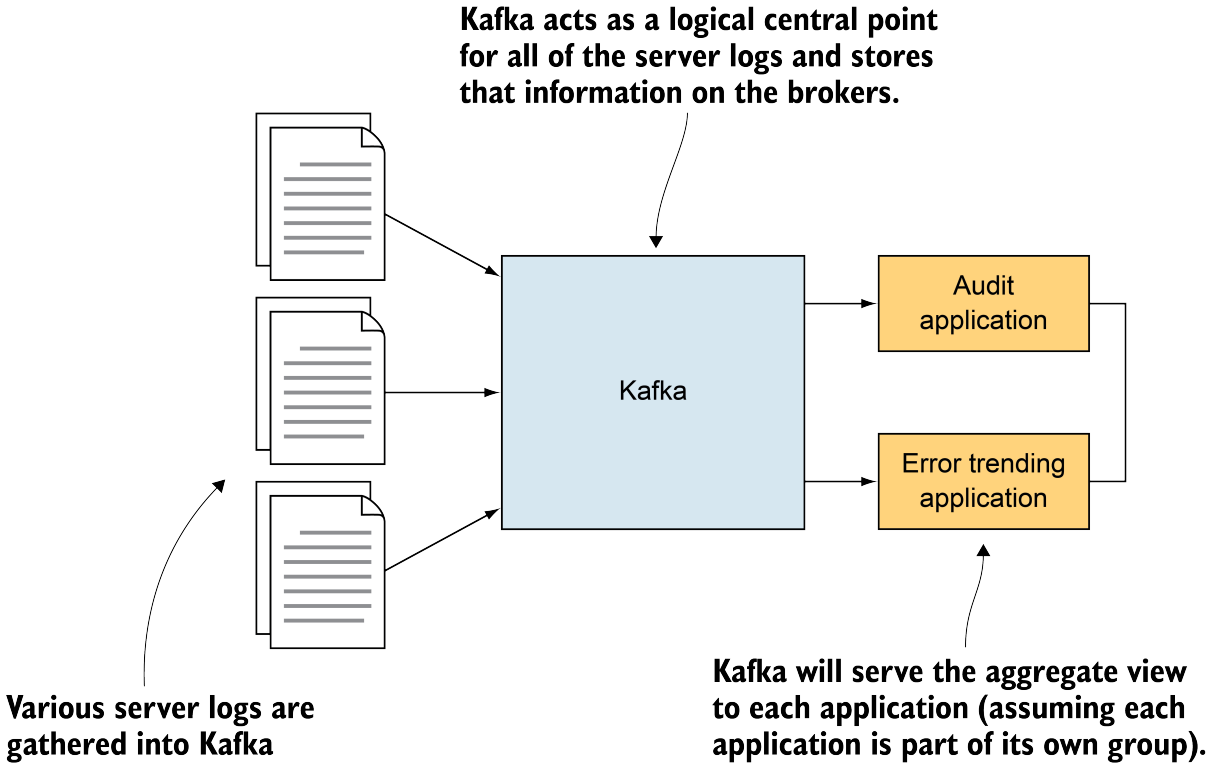

Figure 1.6 Kafka Log Aggregation

Log aggregation is useful in many situations, including when trying to gather application events that were written in distributed applications. In the figure above, the log files are being sent as messages into Kafka, and then different applications have a single logical topic to consume that information. With Kafka's ability to handle large amounts of data, collecting events from various servers or sources is a key feature. Depending on the contents of the log event itself, some organizations have been able to use it for auditing and failure detection trending. Kafka is also used in various logging tools (or as an input option).

How do all of these log file entries even allow Kafka to maintain performance without causing a server to run out of resources? The throughput of small messages can sometimes overwhelm a system since the processing of each method takes time and overhead. Kafka uses batching of messages for sending data as well as writing data. Writing to the end of a log helps as well rather than random access to the filesystem. We will discuss batching messages in Chapter 4 and more on the log format of messages in Chapters 6 and 7.

1.4.2 Later Examples

Microservices used to talk to each other with APIs like REST, but can now leverage Kafka to communicate between asynchronous services with events [11]. Microservices can use Kafka as the interface for their interactions rather than specific API calls. Kafka has placed itself as a fundamental piece for allowing developers to get data quickly. While Kafka Streams is now a likely default for many when starting work, Kafka had already established itself as a successful solution by the time the Streams API was released in 2016. The Streams API can be thought of as a layer that sits on top of producers and consumers. This abstraction layer is a client library that is providing a higher-level view of working with your data as an unbounded stream.

In the Kafka 0.11 release, exactly-once semantics was introduced. We will cover what that means in practice later on once we get a more solid foundation. However, users running end-to-end workloads on top of Kafka with the Streams API may benefit from hardened delivery guarantees. Imagine banks using Kafka to debit or credit your account. With the "at least once" semantics, you would have to ensure that debit was not completed twice (due to duplication of messages) through various consumer-side logic. Streams make this use case easier than it has ever been before to complete a flow without any custom application logic overhead of ensuring a message was only processed once from the beginning to the end of the transaction.

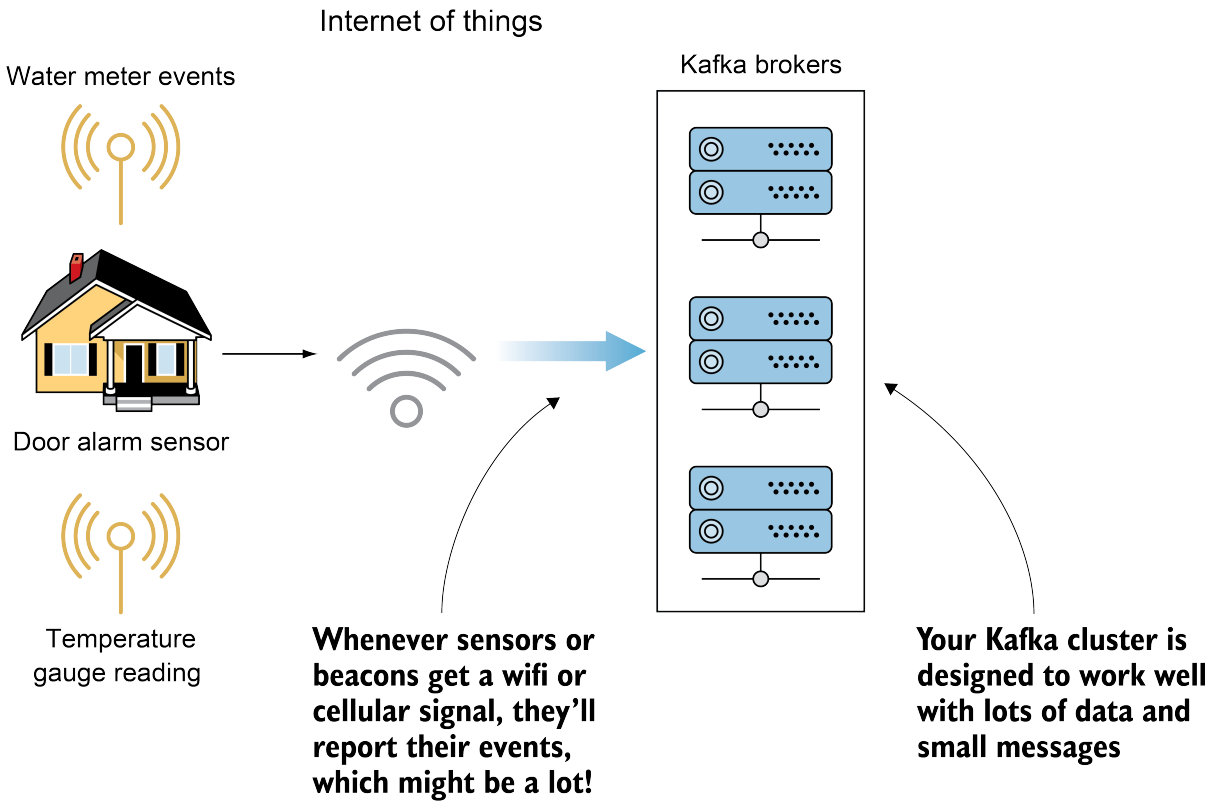

Figure 1.7 IoT

The number of devices for the Internet of Things will only increase with time. With all of those

devices sending messages, sometimes in bursts whenever they get wifi or cellular connection, something needs to be able to handle that data effectively. As you may have gathered, massive quantities of data are one of the critical areas where Kafka shines. As we discussed above, small messages are not a problem for Kafka. Beacons, cars, phones, homes, all will be sending data, and something needs to handle the firehose of data and make it available for action.

These are just a small selection of examples that are well-known uses for Kafka. As we will see in future chapters, Kafka has many practical application domains. Learning the upcoming foundational concepts is essential to see how even more practical applications are possible.

1.4.3 When Kafka might not be the right fit

It is important to note that while Kafka has been used in some interesting use cases, it is not always the best tool for the job at hand. Let's investigate some of the uses where other tools or code might shine.

What if you only need a once-monthly or even once yearly summary of aggregate data? Suppose you don't need an on-demand view, quick answer, or even the ability to reprocess data. In that case, you might not need Kafka running throughout the entire year for that task alone (notably if that amount of data is manageable to process at once as a batch). As always, your mileage may vary: different users have different thresholds on what is a large batch.

If your main access pattern for data is a mostly random lookup of data, Kafka might not be your best option. Linear read and writes is where Kafka shines and will keep your data moving as quickly as possible. Even if you have heard of Kafka having index files, they are not really what you would compare to a relational database having fields and primary keys that indexes are built.

Similarly, if you need the exact ordering of messages in Kafka for the entire topic, you will have to look at how practical your workload is in that situation. To avoid any unordered messages, care should have to be taken to ensure that only one producer request thread is the max simultaneously and that there is only one partition in the topic. There are various workarounds, but if you have vast amounts of data that depend on strict ordering, there are potential gotchas that might come into play once you notice that your consumption would be limited to one consumer per group at a time.

One of the other practical items that come to mind is that large messages are an exciting challenge. The default message size limit is about 1 MB [12]. With larger messages, you start to see memory pressure increase. In other words, the lower number of messages you can store in page cache could become a concern. So if you are planning on sending huge archives around, you might want to look at if there is a better way to manage those messages.

Keep in mind, while you could probably achieve your end goal with Kafka in the above situations (it's always possible), it might not be the first choice to reach for in the toolbox.

1.5 Online resources to get started

The community around Kafka has been one of the best (in my opinion) in making documentation available. Kafka has been a part of Apache (graduating from the Incubator in 2012) and keeps the current documentation at the project website at kafka.apache.org.

Another great resource for more information is Confluent® (www.confluent.io/resources). Confluent was founded by original Kafka's creators and is actively influencing the future direction of the work. They also build enterprise-specific features and support for companies to help develop their streaming platform. Their work helps support the Kafka open-source nature and has extended to presentations and lectures that have discussed production challenges and successes.

As we start to dig into more APIs and configuration options in later chapters, these resources will be a useful reference if further details are needed rather than list them all in these chapters. In Chapter 2, we will discover more details in which we use specific terms and start to get to know Apache Kafka® in a more tangible and hands-on way.

1.6 Summary

- Apache Kafka® is a streaming platform that can be leveraged to process large amounts of events quickly. While Kafka can be used as a message bus, using it only could ignore the capabilities that can provide real-time processing of data.
- While Kafka may have been associated with other Big Data solutions in the past, Kafka stands on its own to provide a scalable and durable system. While it uses same fault-tolerance and distributed systems techniques, Kafka can fill a modern data infrastructure's core needs as its own clustering capabilities.
- In instances of streaming a large number of events, like IoT data, Kafka is made to handle data fast. As more information is available to applications, Kafka can be used to provide results against that data quickly that was once processed offline in batch mode.

2 Getting to know Kafka

This chapters covers

- Reviewing the high-level architecture of Kafka
- Understanding client options and how applications can communicate with the broker
- Producing and consuming your first Kafka message from the command-line
- 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 also be able to send and retrieve your first Kafka message from the command-line.

NOTE **Please visit appendix A if you do not have a Kafka cluster to use or are interested in starting one local on your machine. Appendix A works on updating the default configuration of Apache Kafka and starting the three brokers we will be using in our examples. Please confirm that your instances are still up and running before attempting any examples in this book.**

Let's get started with Kafka and then spend a little more time digging into Kafka's architectural details.

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 your data is represented in Kafka. Each message has an optional key, a value, a timestamp, and optional metadata headers [13]. Here is a simple example of a message:

- Message key: 1234567
- Message value: "Alert: machine Failed"
- Message timestamp: "2020-10-02T10:34:11.654Z"

Figure 2.1 shows the most important parts of messages that users will be dealing with directly. Keys and values will be the focus of most of our discussion and require analysis when designing our messages. Each key and value can interact with their own specific ways to serialize or deserialize their data. The details of how to use serialization will start to come into focus when covering producing messages in Chapter 4.

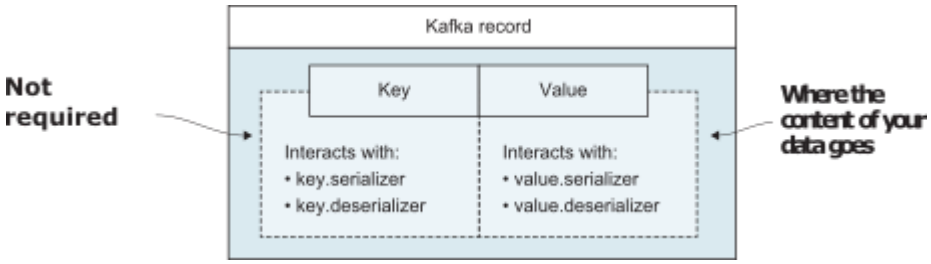

Figure 2.1 Kafka messages are made up of a key and a value (timestamp and optional headers are not shown)

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. Before virtual machines and Kubernetes, you would likely have seen one physical server hosting one broker. Since almost all clusters will have more than one server (or node) we are going to have three Kafka servers running for most of our examples. This local test setup should let us see 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 be creating a topic and sending 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 be using, so we need to have a way to interact with the operating system's command-line interface. The following commands will be 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. While 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.

> **TIP** **Shell Helper**
>
> If you are a command-line user and want a shortcut to auto-complete for commands (and help with the arguments available), check out a Kafka auto-complete project github.com/Kafka-In-Action-Book/kafka_tools_completion. If you are a `zsh` user, you may also want to check out and install Kafka ZSH Completion from 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. We will find this script in the installation directory of Kafka. For example, the path might look like this: `~/kafka_2.13-2.7.1/bin`.

Please 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.

Listing 2.1 Creating the `kinaction_helloworld` Topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create \
  --topic kinaction_helloworld --partitions 3 --replication-factor 3
```

We should see the output: `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 auto-completion.

There are a couple of other options whose meaning may not be clear yet, but to keep moving forward with our exploration, we will quickly define them. Please note that 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, since we have three brokers, using three partitions will give us one partition per broker. For our test workloads, we might not need this many based on data needs alone. However, doing more than one at this stage will let 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 are attempting to have three replicas. These copies will be 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 would 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 will run this command in the terminal window, as shown in Listing 2.2.

Listing 2.2 Verify the Topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

To get a feel for how our new topic looks, let's run another command that will give us a little more insight into our cluster shown in Listing 2.3. Note that it is not like a traditional single topic in other messaging systems: we have replicas and partitions! The number we see next to the labels for the Leader, Replicas, and Isr fields is the `broker.id` that corresponds to the value for the broker we set in our configuration files on our three brokers. 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 Describe Topic kinaction_helloworld

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic kinaction_helloworld
 ❶

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

❶ A `describe` option lets us look at the details of the topic we pass in.

❷ This line shows a quick summary of the total count of partitions and replicas that this topic has

❸ Each partition in the topic is shown. This line is specific for the partition labeled 0

Now, let's zoom-in on `Partition 0`. Partition 0 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 (ISR) 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 health in a distributed system is something that we will want to keep an eye out for. Figure 2.2 shows a view if we look at the one broker with id 0.

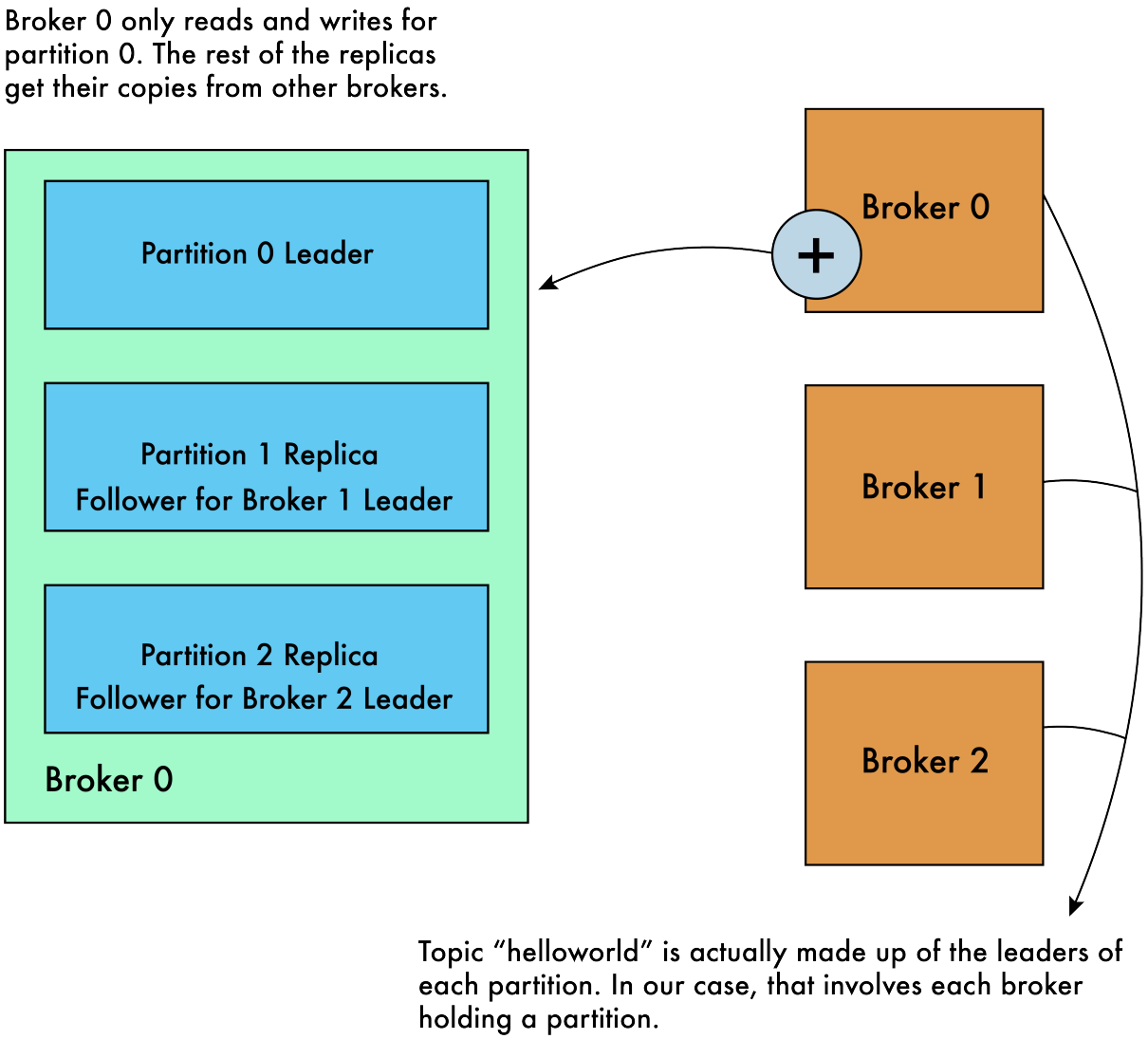

Figure 2.2 View of one broker

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 remember 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 non-leaders will 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 above step to create the topic before sending a message. There is a configuration to enable or disable the auto-creation of topics. However, it is usually best to control the creation of topics as a specific

action as we do not want new topics to randomly show up if one mistypes a topic name once or twice or be recreated due to producer retries.

To send a message, we will start-up a terminal tab or window to run a producer that will run as a console application and take user-input [14]. The below command will start an interactive program that will take over that 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: `kinaction_helloworld`.

Listing 2.4 Kafka Producer Console Command

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_helloworld
```

Notice in Listing 2.4 that we reference the topic that we want to interact with and a `bootstrap-server` parameter. This parameter can be just one (or a list) of our 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-up a new terminal tab or window to run a consumer that will also run as a console application. The command in Listing 2.5 will start a program that will take over that shell as well [15]. On this end, we should see the message we had written in the producer console! Note to make sure that we are using the same `topic` parameter for both commands, otherwise we won't see anything.

Listing 2.5 Kafka Consumer Command

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_helloworld --from-beginning
```

Figure 2.3 shows an example of the output you might see if you completed the command in Listing 2.5!

```
~/kafka $ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
>   --topic kinaction_helloworld --from-beginning
kinaction_helloworld
```

Figure 2.3 Example Consumer Output

As we send more messages and confirm the delivery to the consumer application, we can terminate the process and leave off 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 will be 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 context to discuss the parts we utilized above.

2.3 Tour of Kafka

Table 2.1 The Major Components And Their Roles Within 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

Let's dig into each of these items further to get a solid foundation for the following chapters.

2.3.1 Producers and Consumers

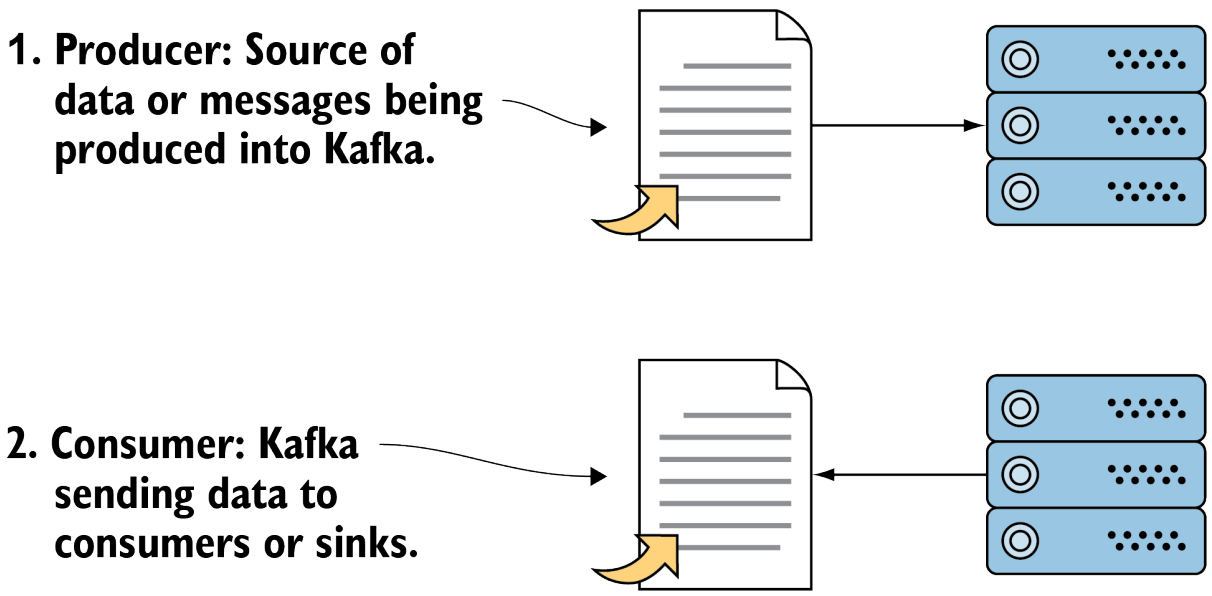

Figure 2.4 Producers vs. Consumer

Figure 2.4 highlights how producers and consumers differ in the direction of their data in relation to the cluster. A Producer is a tool for sending messages to Kafka topics. As mentioned in our use cases in Chapter 1, a good example is log files that are produced from an application. Those files are not a part of the Kafka system until they are collected and sent into Kafka. When you think of input or data going into Kafka, you are looking at a producer being involved somewhere under the covers. There are no default producers, per se, but APIs that interact with Kafka are using 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 under the hood of its implementation in order to provide its

own higher-level API. This specific version 1.0 code is available under an Apache 2 license (github.com/apache/kafka/blob/trunk/LICENSE) and is viewable on GitHub [16]. A producer is also used to send messages inside Kafka itself. For example, if we are reading data from a specific topic and wanted 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 in an upcoming example, it might be helpful to look at code similar in concept to `WorkerSourceTask` which is a Java class that we mentioned earlier. Not all of the source code is listed for the `main` method, but what is left is the logic of sending a message with the standard KafkaProducer. It is not vital to understand each part of the following example, but just try to get familiar with the producer's usage in the example shown in Listing 2.6.

Listing 2.6 Producer sending messages

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

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

producer.close();
```

❶ The `ProducerRecord` will hold each message sent into Kafka.

❷ `producer.send` is what makes the actual call to send to our brokers.

❸ A callback can be used for asynchronous sending of messages.

To send data to Kafka, we will create a `ProducerRecord`. This object lets us define our message and specify the topic, in this case `alert`, to which we want to send it. We are using a custom Alert object as our key in the message. Next we invoke the `send` method sending our `ProducerRecord`. While we can wait for the message, we can also use a callback to send messages asynchronous but still handle any errors. Chapter 4 will provide this entire example in detail.

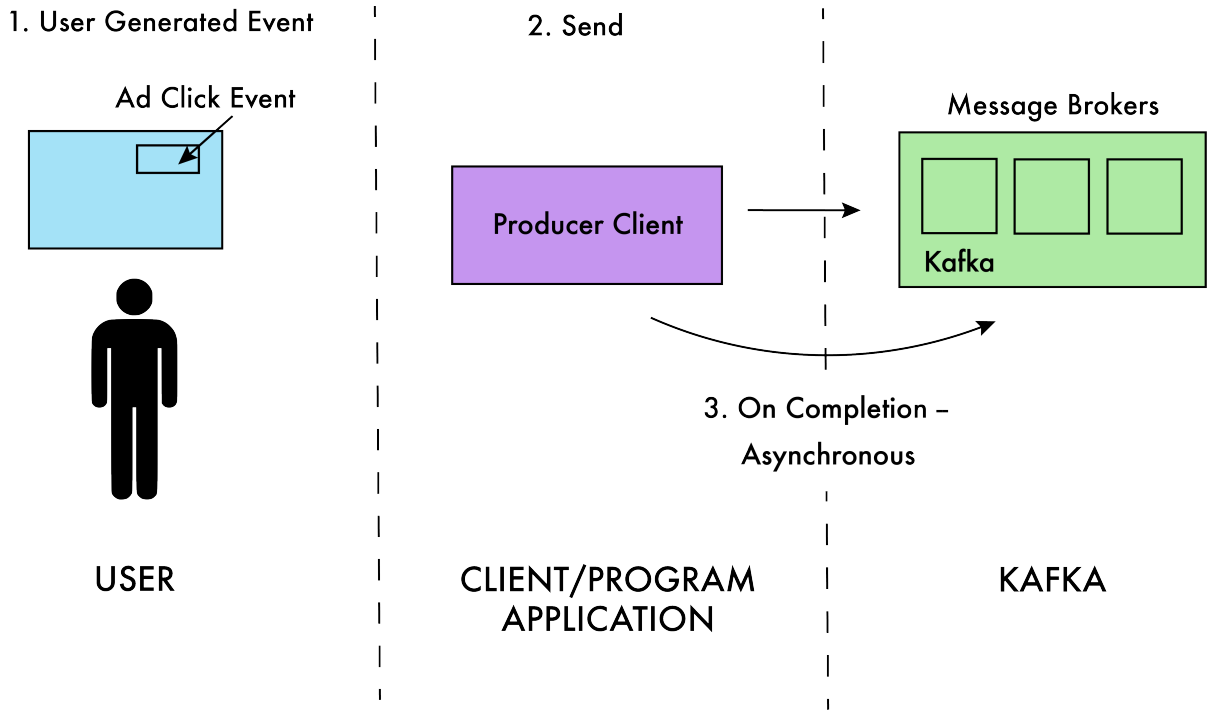

Figure 2.5 Producer Example For User Event

Figure 2.5 shows a user interaction that could start the process of sending data into a producer. A user on a web page clicking an advertisement might indeed fire off an event that would be produced into a Kafka cluster.

In contrast, a Consumer is a tool for retrieving messages from Kafka. In the same vein as producers, if we are talking about getting data out of Kafka, we are looking at consumers being involved directly or indirectly. `WorkerSinkTask` is another class inside the Apache Kafka Connect source code (from version 1.0) that shows the usage of a consumer in Connect that is parallel with the producer example from Connect as well [17]. Consuming applications subscribe to the topics that they are interested in and continuously poll for data. `WorkerSinkTask` is a real example in which a consumer is used to retrieve records from topics in Kafka and is found in the Connect source code at `connect/runtime/src/main/java/org/apache/kafka/connect/runtime/WorkerSinkTask.jav`. Listing 2.7 shows a consumer example we will create in Chapter 5 that displays similar concepts to `WorkerSinkTask.java`.

Listing 2.7 Consuming messages

```
...
consumer.subscribe(List.of("audit"));  ❶

while (keepConsuming) {
  var records = consumer.poll(Duration.ofMillis(100));  ❷
  for (ConsumerRecord<String, String> record : records) {
    log.info("offset = {}, key = {}, value = {}",
              record.offset(), record.key(), record.value());

    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(record.offset() + 1, "");

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

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

❶ The consumer needs to subscribe to the topics that it cares about.

❷ Messages are returned from a poll of data.

Listing 2.7 shows how a consumer object calls a `subscribe` method passing in a list of topics that it wishes to gather data from, in this case `audit`. The consumer then polls the topic(s) and handles any data brought back as `ConsumerRecords`.

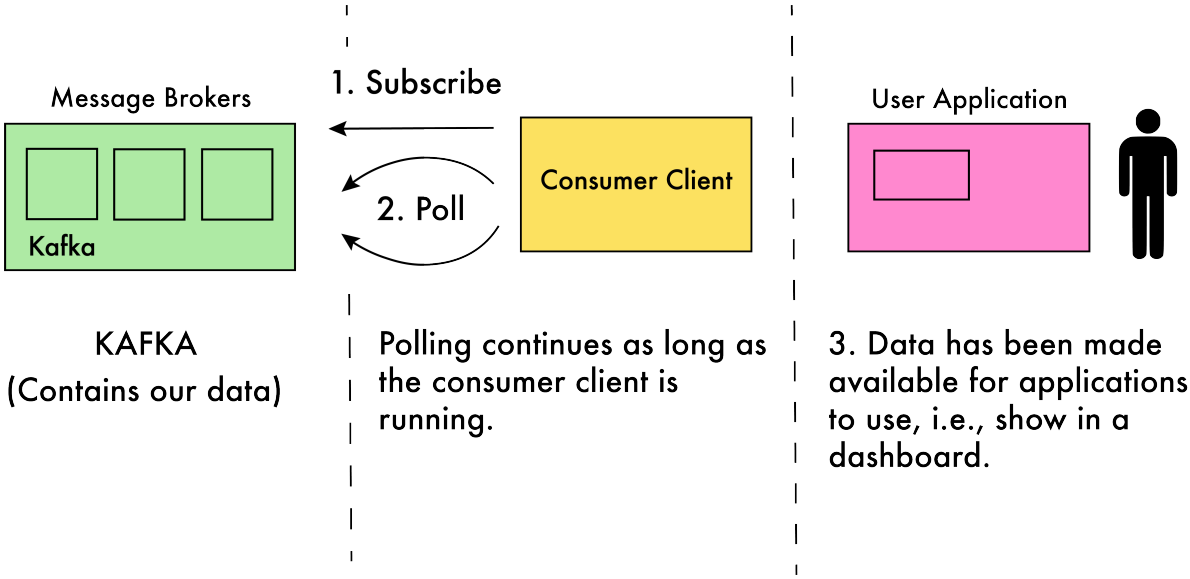

Figure 2.6 Consumer Example Flow

These code listings above show two parts of a concrete use-case example as shown in Figures 2.5 and 2.6. Let's say that an ad agency wants to know how many users clicked on an ad for a new product. The click events generated by users would be the data going into the Kafka ecosystem. The data's consumers would be the ad agency itself that would be able to use its applications to make sense of that data. Putting data into Kafka and out of Kafka with code like

the above (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 the processing of 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 will start to think about the logic of what messages should go where. Topics consist of units called partitions. In other words, one or many partitions can make up a single topic. As far as what is actually implemented on the computer's disk, partitions are what Kafka will be working with for the most part. A single partition replica only exists on one broker and will not be split between brokers.

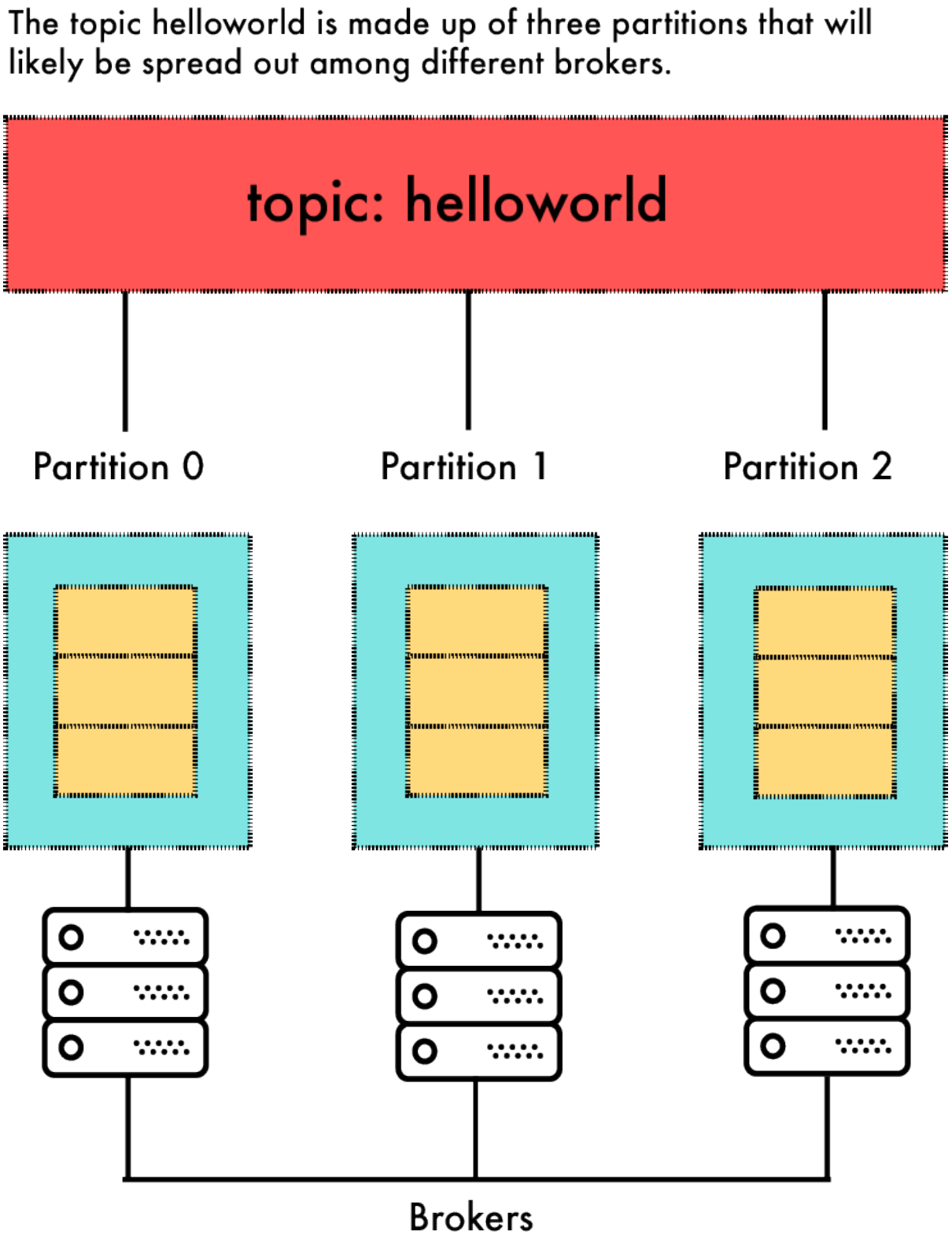

Figure 2.7 Partitions make up Topics.

Figure 2.7 shows how each partition replica leader will exist on a single Kafka broker and will not be divided smaller than that unit. Think back to our first example of the `kinaction_helloworld` topic. If you are looking at reliability and want three copies of the data,

the topic itself is not one entity (or single file) that is copied; instead, it is the various partitions that are replicated three times each.

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. While segment files make up partitions, you will likely not interact directly with them, and it 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 replication factor of three, every partition will have a leader replica elected. That leader will be one of the copies of the partition, and the other 2 (in this case not shown in Figure 2.7) will be followers that update their information from their partition replica leader. Producers and consumers will only read or write from the leader replica of each partition it is assigned during scenarios where there are no exceptions or failures (also known as happy path scenarios). 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.

2.3.3 The What And Why Of Zookeeper

One of the oldest sources of added complexity in the Kafka ecosystem might be that it uses ZooKeeper. Apache ZooKeeper (zookeeper.apache.org/) is a distributed store that is used to provide discovery, configuration, and synchronization services in a highly available way. In versions of Kafka since version 0.9, changes were made for the consumers to have the option not to store information about how far it had consumed messages (called offsets) in ZooKeeper. 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.

IMPORTANT	ZooKeeper Removal To simplify the requirements of running Kafka, the Apache Kafka community proposed the replacement of ZooKeeper with its own managed quorum [18]. Since this work was not yet complete at the time of publication, with an early access release version arriving in 2.8.0, ZooKeeper is still discussed in this work. Why is ZooKeeper still important? This book cover version 2.7.1 and you are likely to see older versions in production use Zookeeper for a while until the changes are fully implemented. Also, while 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.

While Kafka provides fault-tolerance and resilience, someone 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 have seen already, 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 whom the replica leader of a partition is, 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 alone getting out of sync and how it becomes impossible to tell the correct time if all of them are showing different times. The agreement can be challenging across separate brokers, and something is needed to keep Kafka coordinated and working as one in both success and failure scenarios.

One thing to note for any production use cases is that ZooKeeper will be a cluster (called an ensemble), but we will be running just one server in our local setup.

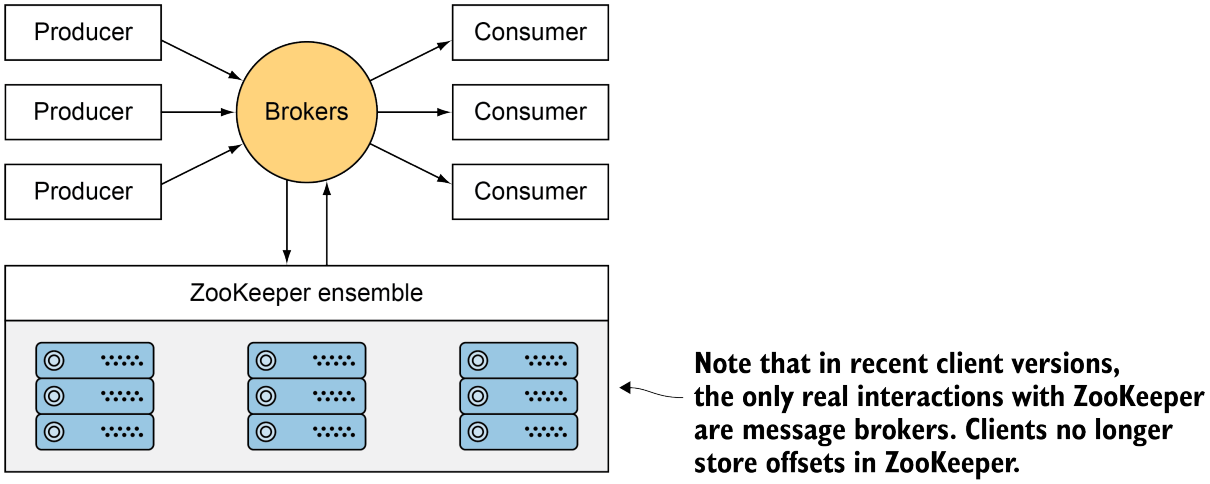

Figure 2.8 ZooKeeper Interaction

Figure 2.8 shows the ZooKeeper clusters and how Kafka's interaction is with the brokers and not the clients.

TIP **ZooKeeper Utils**

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 above concepts will help us be able to add up these building blocks to make a practical application with Kafka. Also, we will start seeing how existing systems that leverage Kafka are likely interacting 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 the JVM. While noted for being able to handle millions of messages quickly, what is it about Kafka's design that makes this possible?

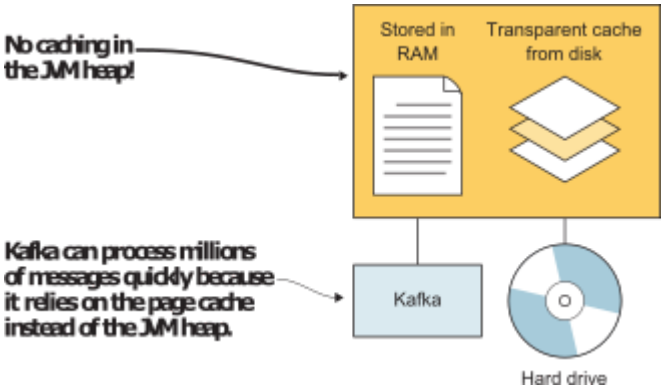

Figure 2.9 OS Pagecache

One of Kafka's keys is its usage of the *pagecache* of the operating system, as shown in Figure 2.9. By avoiding caching in the JVM heap, the brokers can help prevent some of the issues that large heaps can have: i.e. long or frequent garbage collection pauses. Another design consideration was the access pattern of data. While new messages flood in, it is likely that the latest messages would be of most interest to many consumers which could then be served from this cache. Serving from pagecache instead of a disk is likely faster in most cases unless solid-state drives (SSDs) performance catches up in your workload testing as being as performant. In most cases, adding more RAM will help more of your workload fall into the pagecache.

As mentioned before, Kafka uses its own protocol [19]. Using an existing protocol like AMQP was noted by the creators as 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 their needs.

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 LOGGER.error messages in Java.

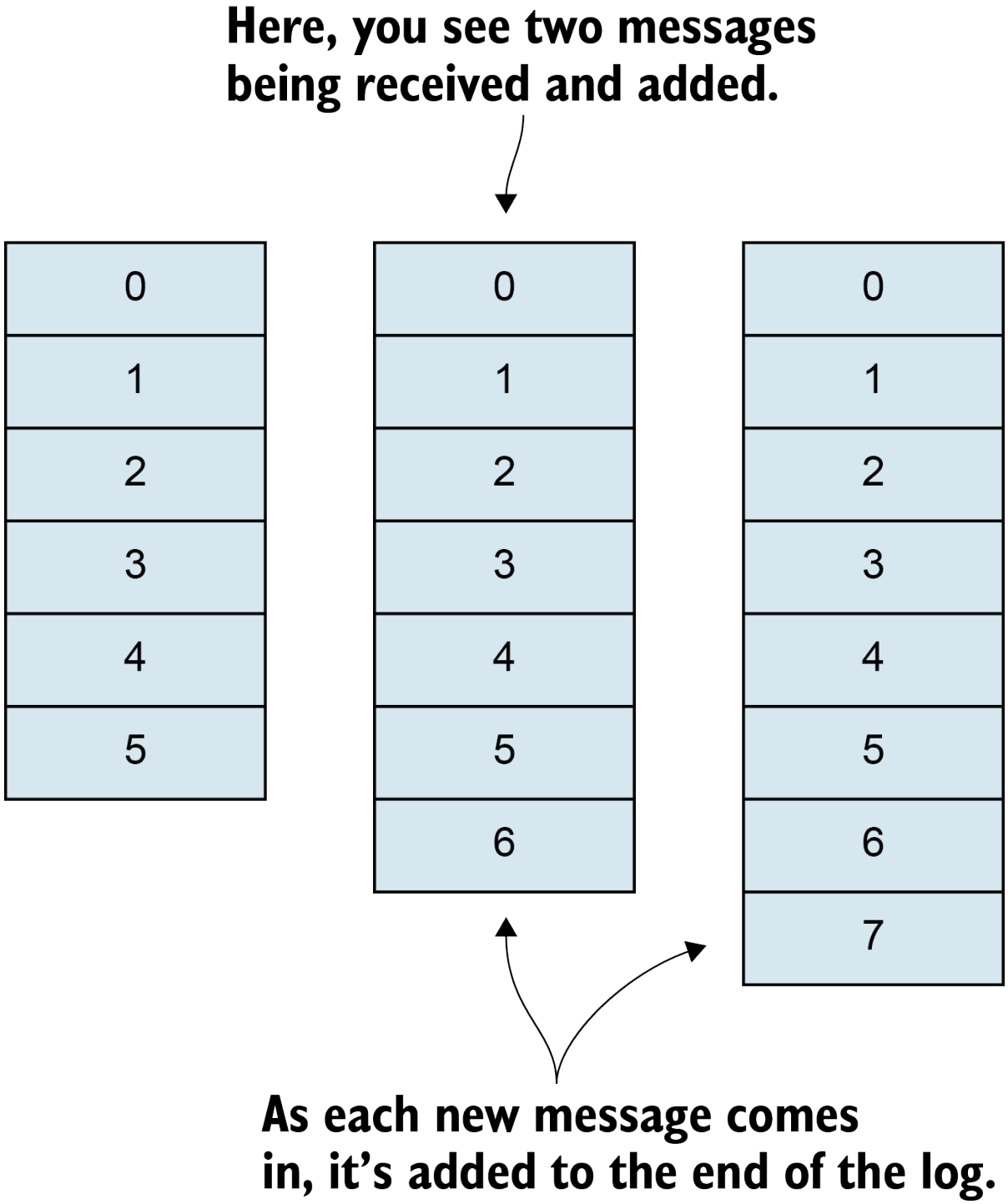

Figure 2.10 Commit Log [20]

Figure 2.10 shows how simple the concept of a commit log can be. While there are more mechanics that take place, like 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 other systems that might use something similar (like a write-ahead-log for a database). It is front-and-center, and its users will use 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. In most traditional systems, linear read and writes usually perform better than random operations that would require spinning hard-drive disks. 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 Kafka commit logs' data hits a configurable size or time retention period, the data is often moved into a permanent store like S3 or HDFS. However, it 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 100GB. [21] Kafka is made to keep its performance fast even while keeping 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 stand-alone products. We are going to look at some of these to see what options we have. The below packages are APIs found in the same source code repository as Kafka core except for ksqlDB.

2.4.1 Kafka Stream

Kafka Streams has grabbed a lot of attention compared to core Kafka itself. This API is found in the Kafka source code project's directory `streams` 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 YARN (Yet Another Resource Negotiator) to run your workloads. However, it still has very powerful features, including local state with fault-tolerance, one-at-a-time message processing, and exactly-once support. The more you move throughout this book, 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 make sure that creating streaming applications was as easy as possible and provides a fluent API (similar to Java 8's Stream API) (also referred to as a domain-specific language (DSL)). It takes the core parts of Kafka and builds on top of those smaller pieces by adding stateful processing and distributed joins, for example, without much more complexity or overhead. Micro-service designs are also being influenced by this API. Instead of data being isolated in various applications, it is rather pulled into applications that can use data independently. Figure 2.11 shows a before and after view of using Kafka to implement a micro-service system [22].

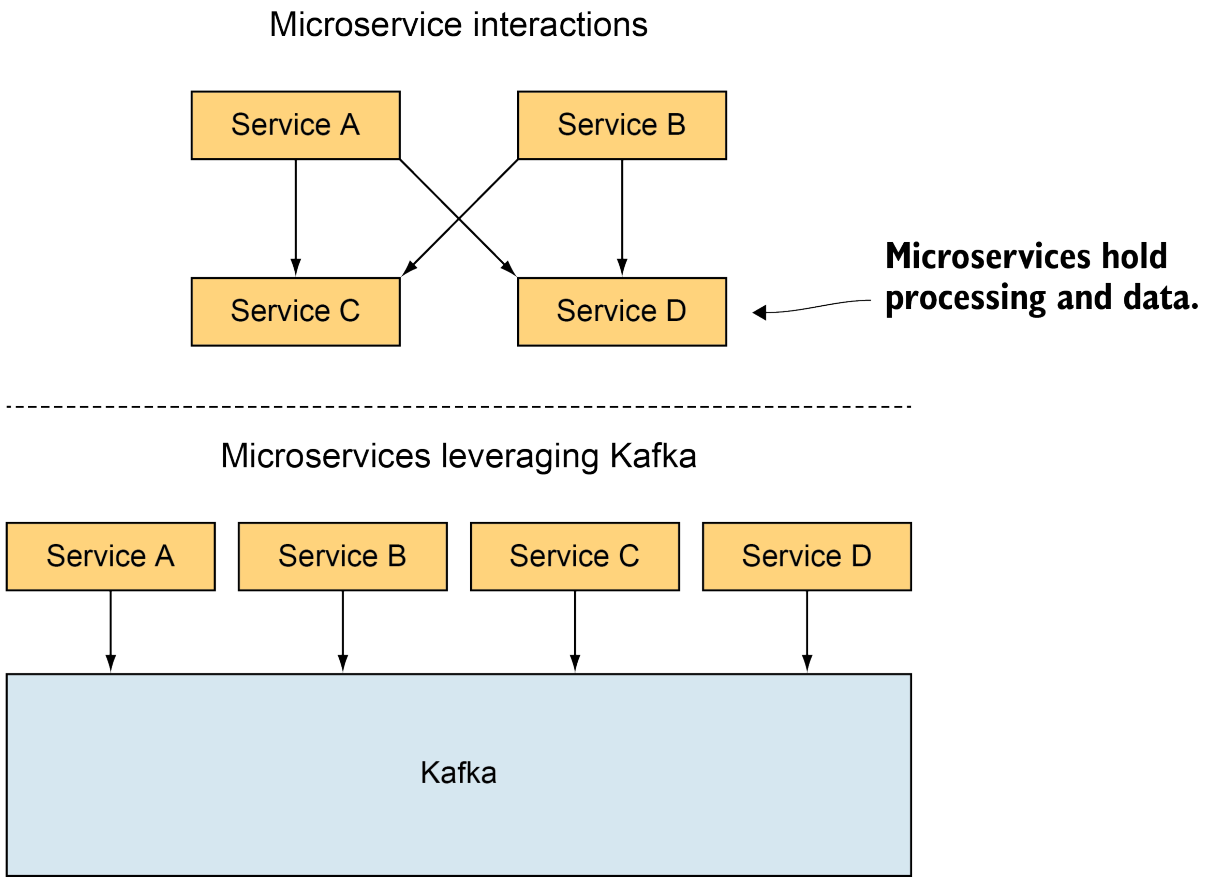

Figure 2.11 Micro-service design

While the top part of the figure without Kafka relies on each application talking directly to other applications at multiple interfaces, the bottom shows an approach leveraging 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.

2.4.2 Kafka Connect

Kafka Connect is found in the core Kafka folder `connect` and is also mostly written in Java. This framework was created to make integrations with other systems easier. In many ways, it can be thought to help replace other tools such as an Apache® incubator project Gobblin and Apache Flume. If one is 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™ (trademark of MySQL AB) 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 a different system. For example, if we wanted messages in some topic to be maintained longer-term, we would use a sink connector to consume those messages from the topic and place them somewhere like HDFS or S3. Figure 2.12 shows this data flow from the database to Connect and then finally to Kafka.

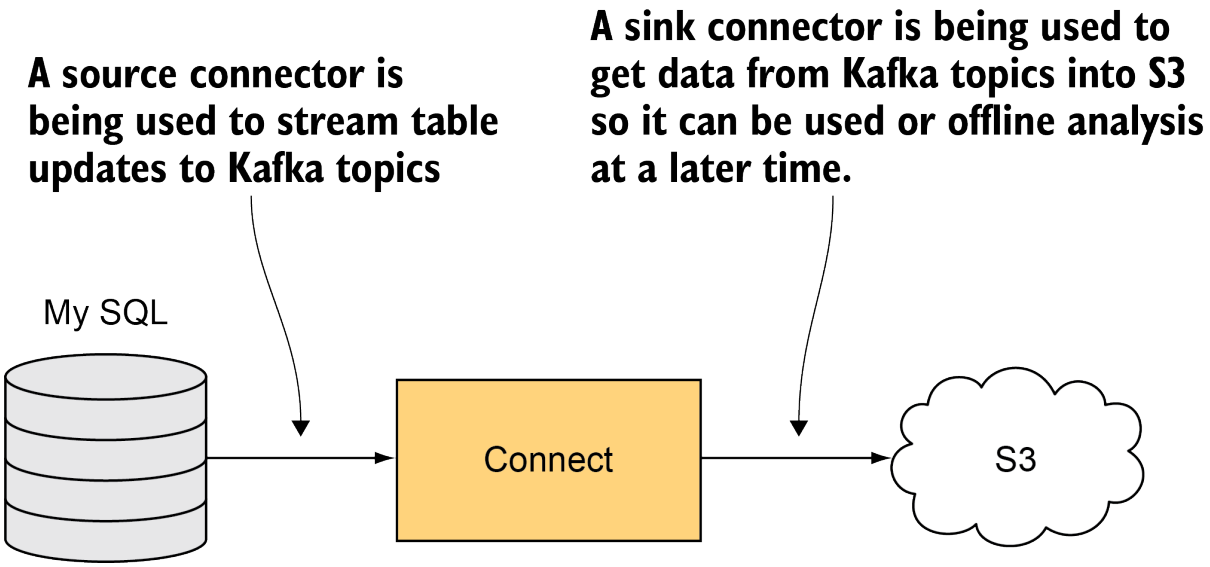

Figure 2.12 Connect Use Case

As a note, a direct replacement of Apache Flume features is not the intention or primary goal of Kafka Connect. Kafka does not have an agent per node setup and is designed to integrate well with stream processing frameworks and 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

With version 0.11.0, Kafka introduced the AdminClient API. 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` which is a different jar than the other APIs discussed. However, this interface is a great tool that will come in handy the more involved we become with Kafka's administration. This tool also uses a similar configuration that producers and consumers use. The source code can be found in `org/apache/kafka/clients/admin` package.

2.4.4 ksqlDB

In late 2017, a developer preview was released by Confluent of a new SQL engine for Kafka that was called KSQL before being renamed to ksqlDB. This allowed developers and data analysts who have used mostly SQL for data analysis to leverage streams by using the interface they have known for years. While 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 leverage the power of continuous flows of data. While the interface for data engineers will be a familiar SQL-like grammar, the idea that queries will be 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 What clients can I use for my language of choice?

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 languages such as Python®, Google Golang™ programming language, Ruby, and Microsoft® .NET®, just to name a few, that are created by various companies and other open-source contributors [23].

As with any code you plan to use in a production setting, you should make sure that you look into any independent clients' quality and if they support the full feature set you want to use. If you genuinely want to peek under the Kafka protocol's hood, there is a guide (kafka.apache.org/protocol.html) to help users understand the details. As a side note, taking a look at other open-source clients can help you develop your client or even help you learn a new language. For example, for Erlang fans, a couple of clients use bit syntax and pattern matching, which can help you tangentially in other parts of your application development.

Since using a client is the most likely way you will interact with Kafka in your applications, let's look at using the Java client. 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 2.8 Java Client Producer

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

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

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

    try (Producer<String, String> producer = new KafkaProducer<>(producerProperties)) {   ❹

      ProducerRecord<String, String> producerRecord =
          new ProducerRecord<>("kinaction_helloworld", null, "hello world again!");     ❺

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

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

❷ This property can take a list of Kafka brokers.

❸ The message's key and value have to be told what format they will be serializing.

❹ This creates a producer instance. Producers are thread-safe! Producers implement `Closable` interface, and will be closed automatically by Java runtume.

❺ This is what represents our message.

❻ Sending the record to the Kafka broker!

The above code is a simple producer. The first step to create a producer involved 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 config item, and its purpose may not be apparent at first glance. This list is a list of your Kafka brokers. A best practice is to include more than one server to ensure that if one server in the list had crashed or was in maintenance, your producer would still have something alive to talk to on start-up. This list does not have to be every server you have, though, as after it connects, it will be able to find out information about the rest of the cluster's brokers and not depend on that list. The `key.serializer` and `value.serializer` are also something to take note of when developing. 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.

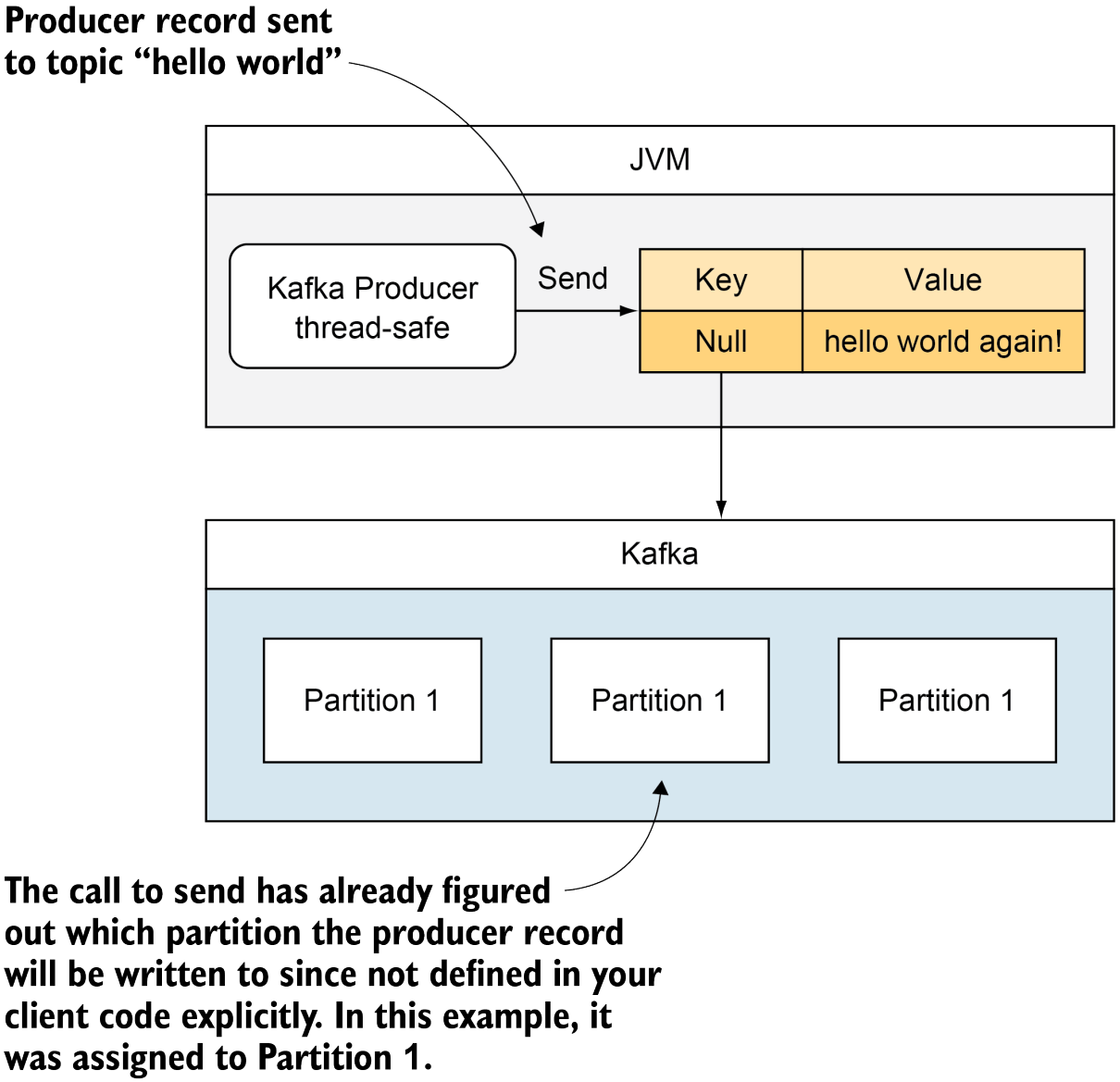

Figure 2.13 Producer Flow

Figure 2.13 helps display the flow that is happening when a producer sends a message. The created producer is thread-safe and takes in the configuration properties as an argument in the constructor we used. With this producer, we can now send messages. The `ProducerRecord` will contain the actual input that we wish to send. In our examples, "kinaction_helloworld" is the name of the topic we wish to send. 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 it can indeed be a null value and this makes our current example less complicated. 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 I want to make sure the message is different? We are working with the same topic with both producers, and since we have a new consumer, we should be retrieving the old message we produced before and our Java client-initiated message. Once our message is ready, we asynchronously send it using the producer. In this case, since we are only sending one

message, we are closing the producer to block until previously sent requests complete and shut down gracefully.

Before running these Java client examples, we'll need to make sure we have the entry in Listing 2.9 in `pom.xml`. We will be using Apache Maven™ in all of the examples of this book.

Listing 2.9 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 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 2.10 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 props = new Properties();    ❶
    props.put("bootstrap.servers",
      "localhost:9092,localhost:9093,localhost:9094");
    props.put("group.id", "helloconsumer");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("key.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer");

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

  private void consume(Properties props) {
    try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)) {
      consumer.subscribe(Collections.singletonList("kinaction_helloworld"));    ❷

      while (keepConsuming) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));    ❸
        for (ConsumerRecord<String, String> record : records) {    ❹
          log.info("[Consumer Record] offset = {}, key = {}, value = {}",
            record.offset(), record.key(), record.value());
        }
      }
    }
  }

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

1. Properties are set the same way as producers.
2. The consumer needs to tell Kafka what topics it is interested in.
3. We want to keep polling for new messages as they come in.
4. We are printing out each record that we consume to the console to see the result.

One thing that jumps out is that we do have an infinite loop in Listing 2.10. It seems weird to do that on purpose, but we are trying to handle an infinite stream of data. The consumer is very 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. We will need to take that into account as we scale past one consumer in later sections [24]. 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 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 could 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.

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

2.6 Stream Processing and Terminology

As we start to look at how to use Kafka, an important thing to note is that the terms used in this book are all written in the context of what it means in regards to Kafka. 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 we start to think of applying Kafka to our work use cases, we will be presented with the following terms and can hopefully use the descriptions below as a lens through which to view our processing mindset.

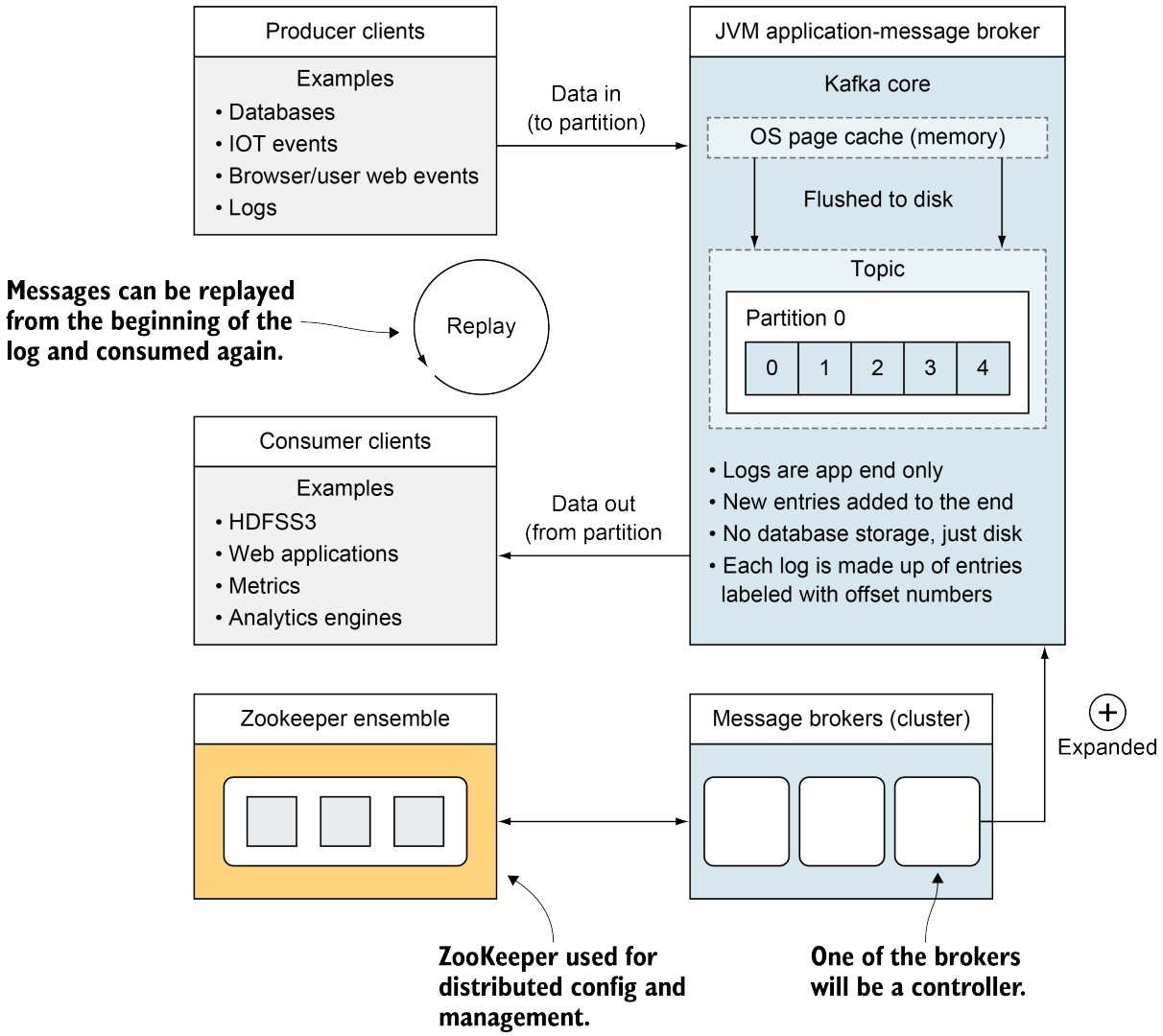

Figure 2.14 Kafka Context Overview

Figure 2.14 provides a very high-level view of the context 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 as the basis for storage. Once data is inside the Kafka ecosystem, consumers will 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. Since Kafka stores data on disk, the ability to replay data in case of an application failure is also part of our context and feature set. These attributes allow Kafka to become the foundation of powerful stream processing applications.

2.6.1 Streaming process

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 [25]. Also, your code should be processing this data all the time and not waiting for a request or time frame to run. As we saw, 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 river, the same principles apply. Sometimes there is a massive amount to transit and sometimes not that much, but it is continuously flowing between destinations.

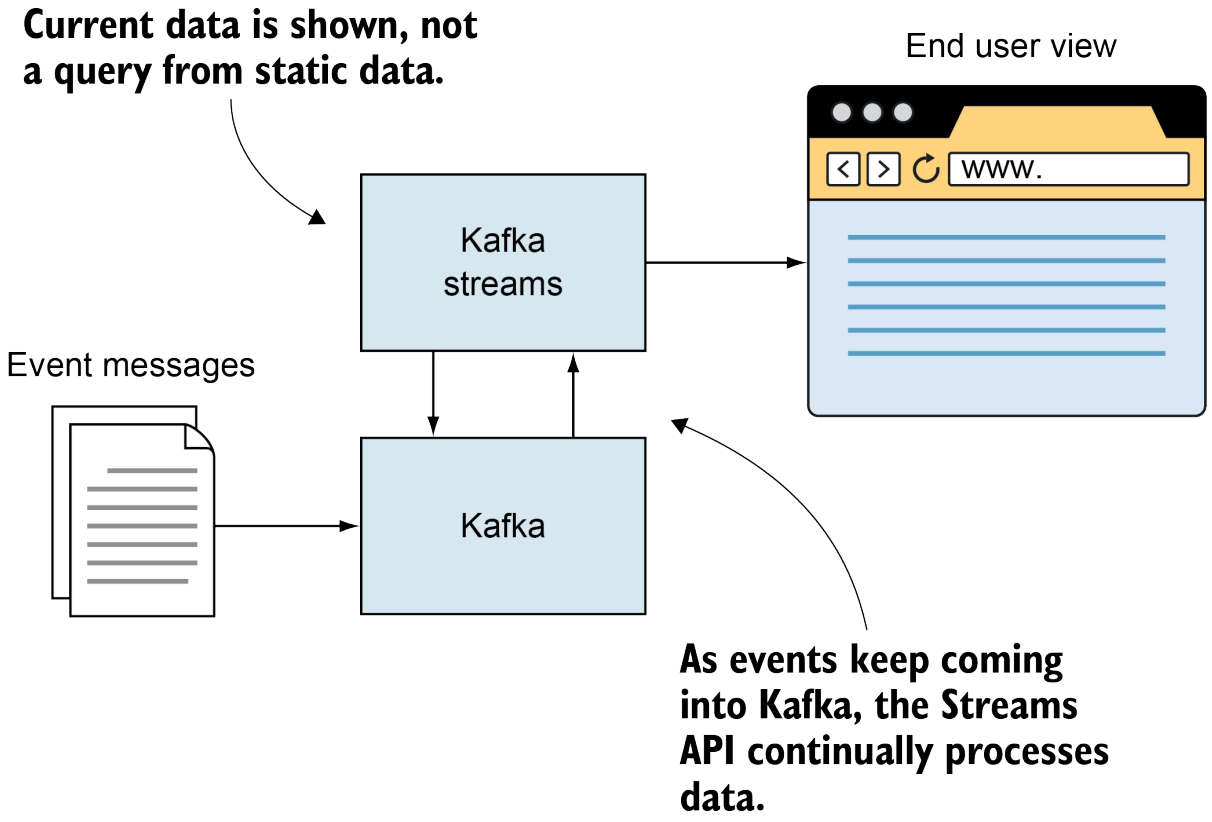

Figure 2.15 Stream Process

Figure 2.15 shows that Kafka Streams depends on core Kafka. While event messages continue to come into the cluster, a consumer application could 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 five minutes would be needed in our case for users to see the latest events.

2.6.2 What exactly once means in our context

One of the most exciting and maybe most discussed features in Kafka is its exactly-once semantics. A few from the Computer Science space still view exactly once delivery as impossible. This book will not discuss the theory behind those views; however, we will look at what these semantics mean for Kafka's everyday usage. Some important things to note: 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 since it won't always be the final endpoint for all data in every scenario.

2.7 Summary

- Messages represent your data in Kafka. Kafka's cluster of brokers handle this data and interact with outside systems and clients.
- Kafka's use of a commit log is an implementation detail. However, this detail helps in understanding the system overall. Messages appended to the end of a log frame the context of how data is stored and 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 a database, into Kafka can help expose data that was once siloed into a system that can provide 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.

3 Designing a Kafka project

This chapters covers

- Designing a real-world Kafka project
- Determining which data format to use for our project
- Identifying existing issues (like data silos) that impact usage of data
- Deciding when data transformation should take place
- Learning how Kafka Connect can help us start a data streaming journey

In our previous chapter, we started to see how we can work with Kafka from the command line and using a Java client. Now, we will expand on those first concepts and look at designing various solutions with Kafka. We will discuss some questions to consider while we lay out a strategy for the example project we'll be starting in this chapter. As we begin to develop our solutions, keep in mind that, like most projects, we might make minor changes along the way, and are just looking for a place to jump in and start developing.

After reading this chapter, we will be on our way to solving real-world use cases while producing a design to facilitate our further exploration of Kafka in the rest of this book. Let's start on this exciting journey!

3.1 Designing a Kafka project

While Kafka is being used in new companies and projects as they get started, that is not the case for all adopters. For those of us who have been in enterprise environments or worked with legacy systems (and anything over five years old is probably considered legacy these days), that is not a luxury we always have in reality. However, one benefit of dealing with existing architectures is that it gives us a list of pain points that we want to address. The contrast will also help us to highlight the shift in thinking about data in our work.

We will work on the project for a company that is ready to shift from their current way of doing data and apply this new hammer named Kafka.

3.1.1 Taking over an existing data architecture

Let's look at some background to give us the context of our fictional example inspired by Kafka's ever-growing usage. One topic mentioned in chapter 1, by Confluent (www.confluent.io/use-case/internet-of-things-iot/), and also in an excellent article by Janakiram MSV titled "Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform" includes Kafka's use with sensors [26]. Using the topic of sensors as a uses-case, we will dig into a made-up example project. Our new fictional consulting company has just won a contract to help re-architect a plant that works on e-bikes and manages them remotely. Sensors were placed throughout the bike which continuously provide events about the condition and status of the internal equipment they are monitoring. However, so many events are generated that the current system ignores most of the messages. We have been asked to help them unlock the potential in that data for their various applications to utilize. Besides this, our current data context includes traditional relational database systems that are large and clustered. With so many sensors and an existing database, how might we create our new Kafka-based architecture without impacting manufacturing?

3.1.2 Kafka Connect

One of the best ways to start our journey is probably not a big-bang approach: all our data does not have to move into Kafka at once. If we use a database today and want to kick the tires on streaming data, one of the easiest on-ramps is to start with Kafka Connect. It can handle production loads, but it does not have to out of the gate. We will take one database table and start our new architecture while letting the existing applications run for the time being. But first, let's get into some examples to get us familiar with Connect.

3.1.3 Connect Features

The purpose of Connect is to help move data into or out of Kafka without writing our own producers and consumers. Connect is a framework that is already part of Kafka that really can make it simple to use pieces that have previously been built to start your streaming journey. These pieces are called *connectors*, and they have been developed to work reliably with other data sources. Recall from Chapter 2, some of the producer and consumer Java client real-world code that we used as examples showed how Connect abstracts those concepts away from you by using them internally to Connect.

One of the easiest ways to start is by looking at how Connect can take a typical application log file and move it into a Kafka topic.

The easiest option to run and test Connect on your local machine is to run it in standalone mode.

Scaling can come later if we like what we can do in standalone mode!

In the folder where you installed Kafka, you should locate the following files: `connect-standalone.properties` and `connect-file-source.properties` under the `config` directory. Peeking inside the `connect-standalone.properties` file, you should see some configuration keys and values that should look familiar from some of the properties you used to make your own Java clients. Knowing the underlying producers and consumer clients can help you understand how Connect uses that same configuration to complete its work by listing items such as `bootstrap.servers` and `key.converter`.

We are taking data from one data source and into Kafka, and we will treat data as being sourced from that file. Using the file, `connect-file-source.properties`, included with your Kafka installation as an example template, let's create our file called `alert-source.properties` and place the text from Listing 3.1 inside as the contents of that file. This file defines the configuration that is needed to define the file name of interest, `alert.txt`, and that we want the data to be sent to the specific topic named `alert-connect`. With configuration and not code, we can get data into Kafka from any file! Since reading from a file is a common task, we can leverage the pre-built classes provided. In this case, the class is `FileStreamSource`. For Listing 3.1, let's pretend that we have an application that sends alerts to a text file.

Listing 3.1 Connect configuration for a file source

```
name=alert-source
connector.class=FileStreamSource      ❶
tasks.max=1     ❷
file=alert.txt      ❸
topic=alert-connect     ❹
```

❶ The class that we are delegating the work of interacting with our source file.

❷ For standalone mode, 1 is a valid value to test our setup.

❸ This is the file that will be monitored for changes.

❹ The topic property is the name of the topic where this data will be sent.

The value of the `topic` property is very significant. We will be using it later to verify that messages are pulled from a file into that specific `alert-connect` topic. The file name of `alert.txt` will be monitored for changes for new messages. We have chosen 1 for the value of `tasks.max` since we only really need one task for our connector and are not worried about parallelism.

IMPORTANT **If you were running ZooKeeper and Kafka locally, make sure that you have your own Kafka brokers still running as part of this exercise if you had shut them down after previous chapters!**

Now that we have done the needed configuration, we'll need to start Connect and send in our configuration. We can start the Connect process by invoking the shell script named `connect-standalone.sh` and include our custom configuration file as a parameter to that script. To start Connect, in a terminal, we'll run the following command and leave it running:

Listing 3.2 Starting Connect for a file source

```
bin/connect-standalone.sh config/connect-standalone.properties \
  alert-source.properties
```

Moving over to another terminal window, we will create a text file named `alert.txt` in the directory in which we started the Connect service. We can add a couple of lines of text to this file using a text editor.

Now we can use the console consumer command to verify that Connect is doing its job. Let's open another terminal, and consume from the `alert-connect` topic using Listing 3.3 as an example. Connect should have ingested the file contents and produced the data into Kafka.

Listing 3.3 Confirming file messages made it to Kafka

```
bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic alert-connect --from-beginning
```

Before moving to another connector type, let's quickly talk about the sink connector and how it can carry Kafka's messages back out to another file. Since the destination (or sink) for this data will be another file, we are interested in looking at the file `connect-file-sink.properties`.

Notice the small difference in the configuration in Listing 3.4 as the new outcome is writing to a file rather than reading from a file as we did before. `FileStreamSink` is declared to define this new role as a sink. The topic `alert-connect` will be the source of our data in this scenario. Placing the text from Listing 3.4 in a new file titled `alert-sink.properties` will set up our new configuration.

Listing 3.4 Starting Connect for a file source

```
name=alert-sink
connector.class=FileStreamSink    ❶
tasks.max=1                       ❷
file=alert-sink.txt               ❸
topics=alert-connect              ❹
```

❶ This is an available out of the box class to which we are delegating the work of interacting with our file.

❷ For standalone mode, 1 is a valid value to test our setup.

❸ This is the destination file for any messages that make it to our Kafka topic

❹ The topic property is the name of the topic that the data will come from.

If the Connect instance is still running in a terminal, we'll need to close that terminal or stop the process by pressing `Ctrl-C`. Then we'll restart with the file-source and file-sink property files. Listing 3.5 shows starting Connect again with both our custom alert source and sink properties. The end result should be data flowing from a file into Kafka and back out to a separate destination file.

Listing 3.5 Starting Connect for a file source and sink

```
bin/connect-standalone.sh config/connect-standalone.properties \
  alert-source.properties alert-sink.properties
```

To confirm that Connect is using our new sink, open the file we used in our configuration for the sink file, `alert-sink.txt`, to verify that we can see the messages that were in the source file, and had been sent to the Kafka topic.

3.1.4 Connect for our Invoices

Let's look at another requirement, dealing with our invoices. Connect allows those with in-depth knowledge of creating custom connectors and share them with others to help those of us that may not be the experts in those systems.

> **NOTE** **To discover connectors that the community and vendors have developed, check out Confluent Hub confluent.io/hub - an app store for connectors.**

Now that we have used a connector, it should be relatively simple to integrate a different connector since Connect has standardized the interaction with other systems.

To start using Connect in our manufacturing example, we will look at using an existing source connector that will stream table updates to a Kafka topic from a local database.

Again, our goal is not to change the entire data processing architecture at once. We are going to show how we would start bringing in updates from a database table-based application and develop our new application in parallel while letting the other system exist as-is. Our first step is to set up a database for our local examples. For ease of use and to get started quickly, we'll be using SQLite™. If you can run `sqlite3` in your terminal and get a prompt, then you are already set. Otherwise, use your favorite package manager or installer to get a version of SQLite that will work on your OS.

TIP	Confluent Open Source and CLI For ease of development, we are going to be using connectors from Confluent for the SQLite example. Check out the Commands.md in the source code of Chapter 3 to install instructions. The rest of the examples will reference commands in the Confluent installed directory only and NOT the Kafka install directory.

To create a database, we just run the following from the command line: `sqlite3 kafkatest.db`. In this database, we will run the following to create the `invoices` table and insert some test data. As we design our table, it is helpful to think of how we will capture changes into Kafka. Most use-cases will not require us to capture the entire database but just changes after the initial load. A timestamp, sequence number, or ID could help us determine what data has changed and needs to be sent to Kafka. In Listing 3.6, the id or modified columns could be our guide to Connect to let Kafka know which data has changed in the table.

Listing 3.6 Creating the Invoices Table

```
CREATE TABLE invoices(                          ❶
   id INT PRIMARY KEY     NOT NULL,             ❷
   title           TEXT    NOT NULL,
   details        CHAR(50),
   billedamt         REAL,
   modified     TIMESTAMP DEFAULT (STRFTIME('%s', 'now')) NOT NULL
);

INSERT INTO invoices (id,title,details,billedamt)  \
  VALUES (1, 'book', 'Franz Kafka', 500.00 );    ❸
```

❶ This will create a table called 'invoices' that we will use in our examples

❷ Having an id that increments will help Connect know which entries to capture

❸ This is an example to insert test data into our new table

By copying the pre-built `etc/kafka-connect-jdbc/source-quickstart-sqlite.properties` file to `kafkatest-sqlite.properties` and then after making slight changes to our database table name, we can see how additional inserts and updates to the rows will cause messages to be sent into Kafka.

IMPORTANT	Refer to the source code for chapter 3 to see more detailed setup instructions for finding the JDBC connector files in the Confluent installation directory. It is not part of the Apache Kafka distribution like the file connector.

Now that we have a new configuration, we need to start Connect passing it this property file. Listing 3.7 uses the generic command discussed at the Confluent JDBC Source Connector page [27].

Listing 3.7 Starting Connect for a database table source

```
bin/confluent local start
# OR
bin/connect-standalone etc/schema-registry/connect-avro-standalone.properties \
  etc/kafka-connect-jdbc/kafkatest-sqlite.properties   ❶
```

❶ Note that the schema-registry and sqlite files are from the Confluent install. You can launch Connect with the `confluent cli` tool or with the standalone connect script.

> **NOTE** **Kafka Connect provides APIs and documentation to build our connector but we are not going to be exploring them further in this book. If you're interested in exploring the topic of building your connectors, I recommend watching the excellent presentation of Randall Hauch (Apache Kafka committer and PMC) from Kafka Summit San Francisco 2018 located at www.confluent.io/kafka-summit-sf18/so-you-want-to-write-a-connector.**

While the power of Kafka Connect is great for moving our existing database table to Kafka, our sensors, which will not be database-backed, are going to require a different technique.

3.2 Sensor Event Design

Since there are no existing connectors for our start-of-the-art sensors, we can directly interact with their event system through custom producers. This ability to hook into and write our producers to send data into Kafka is the context in which we will look at the following requirements.

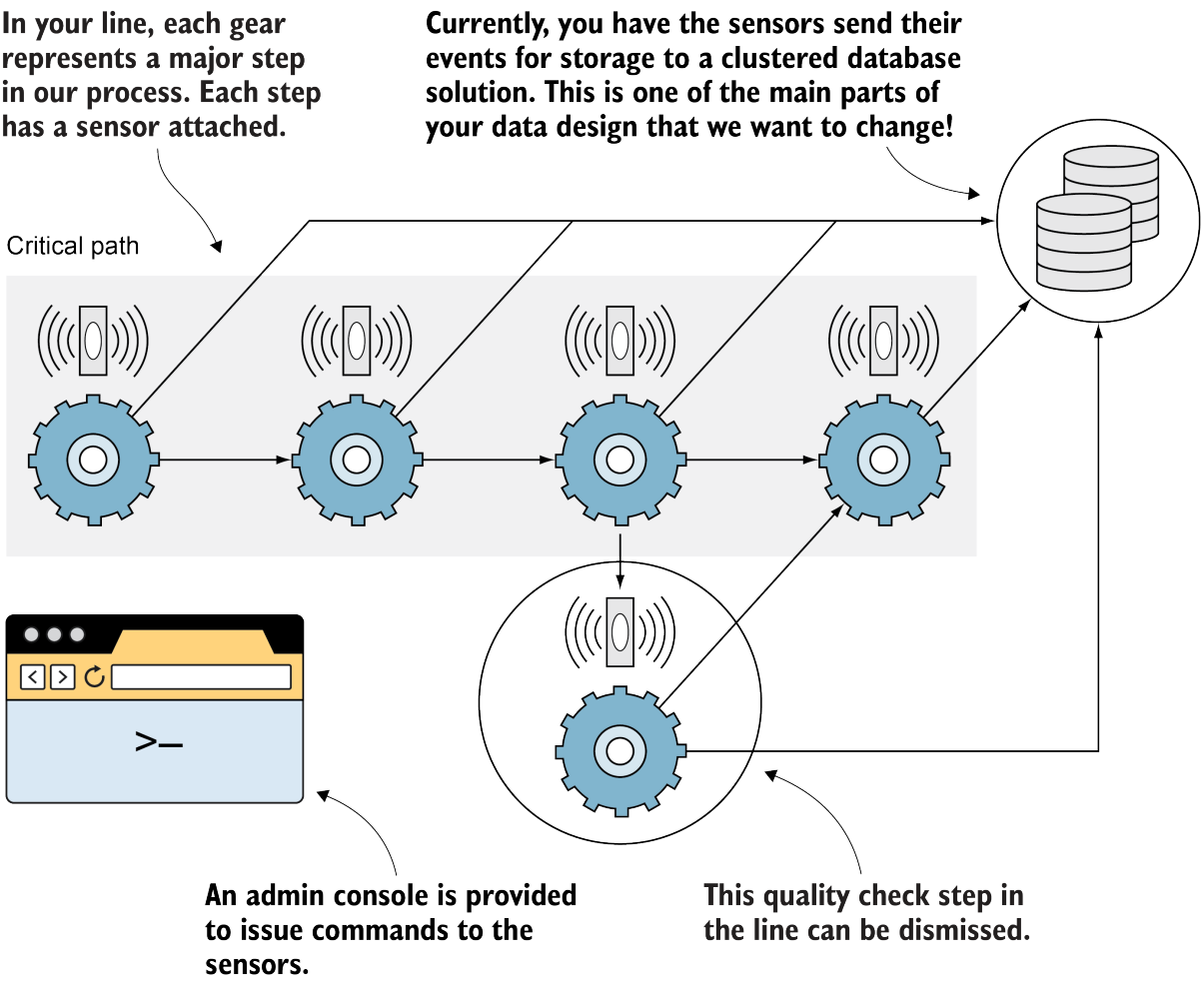

Figure 3.1 Factory Setup

Figure 3.1 shows that there is a critical path of stages that need to work together. One of the steps is an additional quality check which can be skipped, if down for maintenance or failure, to avoid processing delays. Sensors are attached to all of the bikes's internal steps and they send messages to the clustered database machines that exist in the current system. There is also an administration console used remotely to update and run commands against the sensors built into the system already.

3.2.1 Existing issues

DEALING WITH DATA SILOS

Let's start by discussing one of the issues that has come up in most of our previous use cases. The need for data to exist and be available to users is a challenging problem. In many projects, the data and the processing were owned by an application. If others wanted to use that data, they would need to talk to that application owner. And what are the chances that they provided the data in a format that could be easily processed? Or what if they did not provide the data at all?

The shift from most traditional data thinking is to make the data available to everyone in its raw

source. If you have access to the data as it comes in, you do not have to worry about the application API exposing it in their specific formats or after custom transformations have been done.

And what if the application providing the API parsed the original data wrong? To untangle that mess might take a while if we have to recreate the data from changes to the original data source.

RECOVERABILITY

One of the excellent perks of a distributed system like Kafka is that failure is an expected condition, planned for and handled. However, along with system blips, we also have the human element in developing applications. If an application had a defect or logic issue that destroyed our data, what would be our path to correct it? With Kafka, it can be as simple as starting to consume from a topic from the beginning as we did with the console consumer flag `--from-beginning` in Chapter 2.

The retention of the data makes it available for use again and again. The ability to reprocess data for corrections is powerful. If the original event was not available, it might be hard to retrofit the existing data.

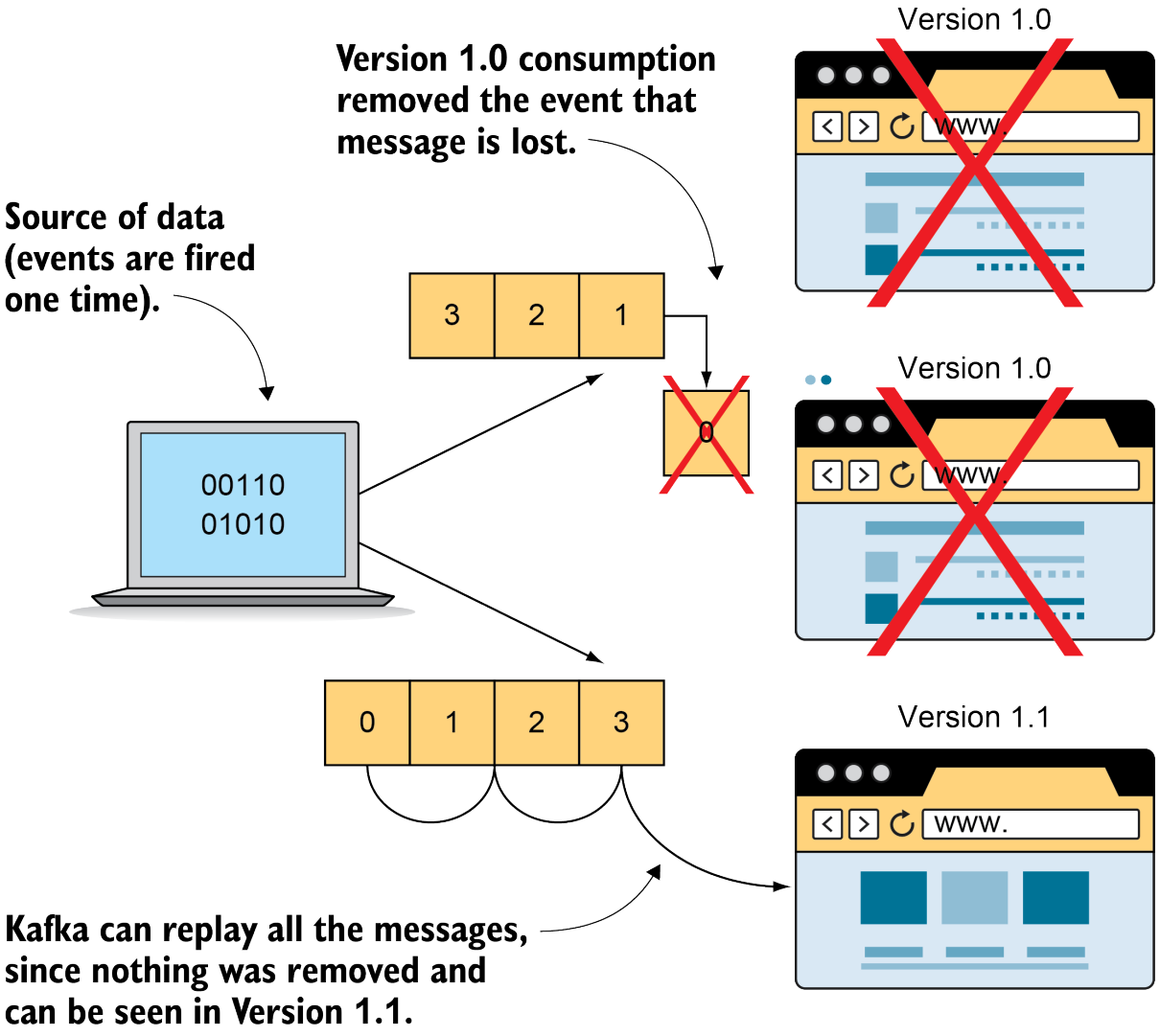

Figure 3.2 Developer Coding Mistake

Since events are only produced once from the sensor source for that specific instance, the message broker can play a crucial part in our consumption pattern. If the message in a queuing system is removed from the broker after a subscriber reads the message as in version 1.0 of the application in Figure 3.2, it is gone from the system. If a defect in an application's logic was found after the fact, analysis would be needed to see if data could be corrected using what was leftover from the processing of that original event since it will not be fired again. Kafka brokers allow for a different option. The application can replay those messages that it has already consumed with the new application logic in version 1.1. Messages can be replayed, and our new application code that fixed our logic mistake from version 1.0 can process all the events again. The chance to process our events again makes it easier to enhance our applications without data loss or corruption.

The replay of data can also show you how a value changed over time. It might be beneficial to draw a parallel between replaying the Kafka topic and the idea of a write-ahead log (WAL). With a WAL, you can tell what a value used to be and the changes that happened over time since

modifications to values are written in the log before they are applied. WALs are commonly found in database systems and help a system recover if an action fails during a transaction. If you followed the events from the beginning to the end, you would see how data got from its initial value to its current value.

WHEN SHOULD DATA BE TRANSFORMED

One of the most common tasks for engineers that have worked with data for any amount of time is one in which data is taken from one location, transformed, and then loaded into a different system. Extract, transform, and load (ETL) is a common term for this work and might seem like second nature when approaching another data system. However, usually, transformation is something that can wait for a different stage in the process. Whether data is coming from a database or a log event our preference is to get the data into Kafka first. Then the data will be available in its purest form. Each step before it is stored in Kafka is an opportunity for the data to be altered or injected with various formatting or programming logic errors.

One great reason to transform the data is if it is important for your records to have keys (like unique ids) that didn't exist before or need some processing to create initially. Just keep in mind that hardware, software, and logic can and will fail in distributed computing, so it's always great to get data into Kafka first, which gives you the ability to replay data if any of those failures occur. Otherwise, you are at the mercy of the tools from which you extracted the data to attempt to gather that data again.

3.2.2 Why Kafka is a right fit

Does Kafka even make sense in this situation? Of course, this is a book about Kafka, right? However, let's quickly try to pinpoint a couple of compelling reasons to give Kafka a try.

One thing that has been made clear by our clients is that the current database is getting expensive to scale up in a vertical way. By vertical scaling, we mean increasing things like CPU, RAM, and disk in an existing machine. To scale dynamically, we would be looking at adding more servers to our environment. With the ability to horizontally scale-out our cluster, we hope to get more overall benefits for our buck. While the servers we will run our brokers on will not be the cheapest machines money can buy, 32 GB or 64 GB of RAM on these servers will handle production loads.

The other item that probably jumped out at you is that we have events being produced continuously. This should sound very similar to the definition of stream processing we talked about earlier. The constant feed of data won't have a defined end time or stopping point. Our system should be ready to handle messages constantly.

Another interesting point to note is that our messages are usually under ten kB. The small size of the messages will be under the default 1 MB size, requiring some configuration changes. The

smaller the message size and the amount of memory we can offer to page cache should put us in good shape to keep our performance healthy.

During this scenario review, some security-minded developers might have noticed no built-in disk encryption for the brokers (data at rest). However, that isn't a requirement of the current system, and the data in motion options like Transport Layer Security (TLS) will help for protection during transit. We also discussed that we would focus on getting our system up and running and then worry about adding security at a later point in our implementation.

3.2.3 Thought starters on our design

One thing to note is what features are available for specific versions. While we will use a recent version (at the time of this writing 2.7.1) for our examples, some developers might not have control over the current broker and client versions they are using due to existing infrastructure. It is good to keep in mind when some of the features and APIs we might use made their debut. Table 3.1 gives a highlight of some of the past major features and is not inclusive of all versions.

Table 3.1 Past Kafka Version History Milestones

Kafka Version	Feature
2.0.0	ACLS with prefix support, host-name verification default for SSL
1.0.0	Java 9 Support, JBOD disk failure improvements
0.11.0.0	Admin API
0.10.2.0	Improved client compatibility
0.10.1.0	Time-based Search
0.10.0.0	Kafka Streams, Timestamps, Rack Awareness
0.9.0.0	Various Security features (ACLS, SSL), Kafka Connect, 'New Consumer' Client

Another thing to note since we are focused on clients in the next few chapters is the feature: improved client compatibility. Broker versions since 0.10.0 can work with newer client versions. This is important because we can now try new versions of clients, by upgrading them first. The brokers can remain at their version until we decide that we want to upgrade them. This will come in handy if you are running against a cluster that already exists as you work through this material.

Now that we have decided to give Kafka a try, it might be time to decide how we want our data to exist. The following questions are intended to make us think about how we want to process our data. These preferences will impact various parts of our design, but our main focus is just on figuring out the structure. We will cover the implementation in later chapters. This list is not meant to be complete but is a good starting point in planning our design.

Is it okay to lose any messages in the system? For example, is one missed event about a mortgage payment going to ruin your customer's day and their trust in your business? Or is it a minor issue, like your social media account RSS feed missing a post? While the latter is unfortunate, would it be the end of your customer's world?

Does your data need to be grouped in any way? Are the events correlated with other events that are coming in? For example, are we going to be taking in address changes? In that case, we might want to associate the various address changes with the customer whose address is changing. Grouping events upfront might also prevent the need for applications to coordinate messages from multiple consumers while reading from the topic.

Do you need data delivered in exact order? What if a message gets delivered in an order other than when it occurred? For example, if you get an order canceled notice before the actual order. The product might end up shipping out due to order alone. The customer service impact are probably good enough reasons to say that the ordering is indeed essential in that case. Or course, not everything will need this order. If you are just looking at total SEO data for your business, the order is not as important as making sure that you can get a total at the end.

Do you only want the last value of a specific item? Or is the history of that item important? Do you care about the history of how your data has evolved? One way to think about this is how data is updated in a traditional relational database table. It is mutated in place, i.e., the older value is gone and the newer value replaces it. The history of what that value looked like a day ago (or even a month ago) is lost.

How many consumers are we going to have? Will they all be independent of each other, or will they need to maintain some sort of order when reading the messages? If you are going to have a bunch of data that you want to consume as quickly as possible, it will inform and help shape how we break up our messages on the tail end of our processing.

Now that we have a couple of questions to ask as our context let's try to apply them to our actual requirements.

3.2.4 User data requirements

Our new architecture will need to provide a couple of specific key features.

In general, we want the ability to capture messages, even if the consuming service is down. If one of the consumer applications is down in our remote plant, we want to make sure that it can process the events later without dropping messages entirely. When the application is out of maintenance or comes back up after a failure, we want it to still have the data it needs.

We want the statuses from our sensors as either working or broken - a sort of alert. We want to make sure we see if any part of our bike could lead to total failure.

Along with the preceding information, we also want to maintain a history of the sensors' alert status. This data will be used to determine if we can find trends and predict failures from sensor data before actual events lead to broken equipment.

We also want to keep an audit log of any users that pushed updates or queries directly against the sensors. For compliance reasons, we want to know who did what administration actions on the sensors themselves.

3.2.5 High-Level Plan for applying our questions

Let’s take a closer focus at our requirement to create an audit log. Overall, it seems like everything that comes in from the management API will need to be captured. We want to make sure that only users with access were able to perform actions against the sensors. We should not lose messages as our audit would not be complete without all the events. In this case, we do not need any grouping key. Each event can be treated as independent. The order does not matter inside our audit topic, as each message will have a timestamp in the data itself. Our primary concern is that all the data is there to process.

As a side note, Kafka itself does allow messages to be sorted by time, and your message payload can include time as well. However, this specific use-case does not warrant this usage.

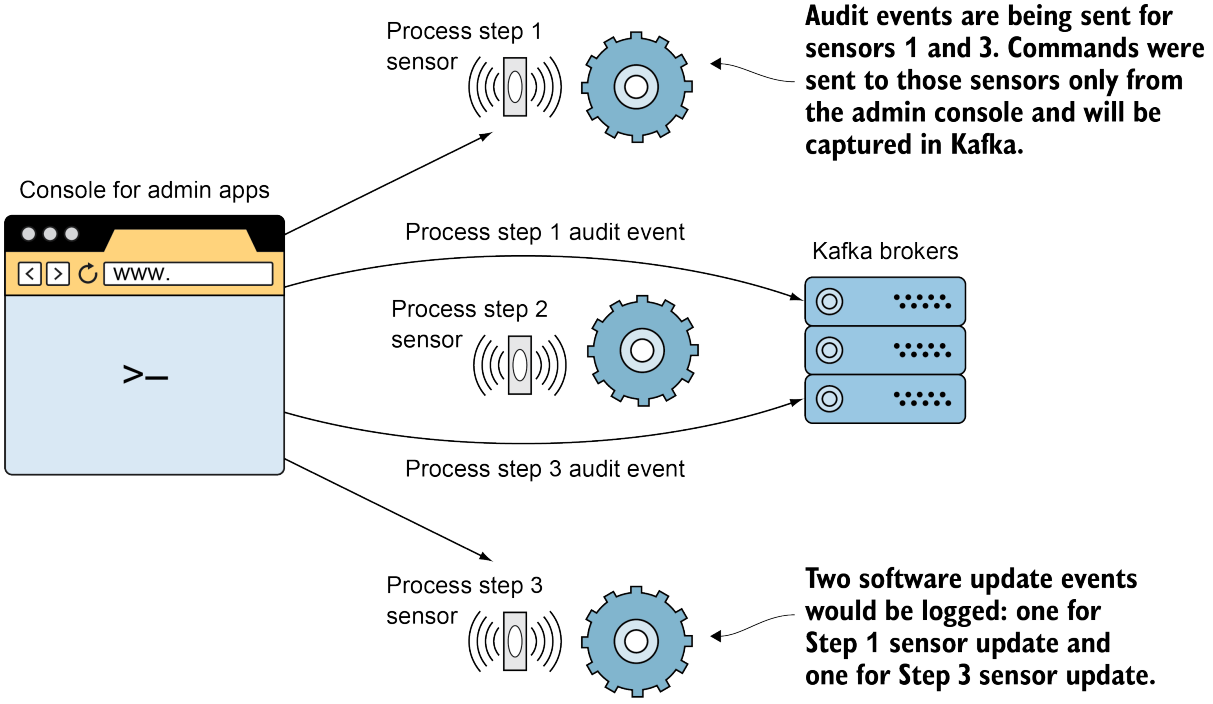

Figure 3.3 Audit Use Case

Figure 3.3 shows how a user would generate two audit events from a web administration console by sending a command to sensor 1 and sensor 3. Both commands should end up as separate events in Kafka.

Table 3.2 Audit Checklist

Kafka Feature	Concern?
Message Loss	X
Grouping	
Ordering	
Last value only	
Independent Consumer	X

To make the things we've discussed a little clearer, I have created a rough checklist of things that we should consider regarding data for each requirement in Table 3.2. This at a glance view will help us when determining the configuration options we want to use for our producer clients. In this audit producer, we are concerned with making sure that no data is lost and that consuming applications do not have any worries about data being ordered or coordinated.

The alert trend of statuses requirement deals with the the alert of each process in the bike's system, with a goal of spotting trends. So, it might be helpful to group this data using a key. We have not addressed the term 'key' in-depth, but it can be thought of as a way to group related events. We will likely use the bikes parts stage name where the sensor is installed, since it will be unique from any other stage name. We will want to look across the key at all of the events, for a

given stage, that have been produced to spot these trends over time. By using the same key for each sensor, we should be able to consume these events easily. Since alert statuses are sent every 5 seconds, we are not concerned about missing a message since the next one should arrive again shortly. Remember, we are concerned with the equipment's alert making the line work, not the sensor's alert. If a sensor sends a 'Needs Maintenance' message every couple of days, that is the type of information we want to have to spot trends in equipment failing.

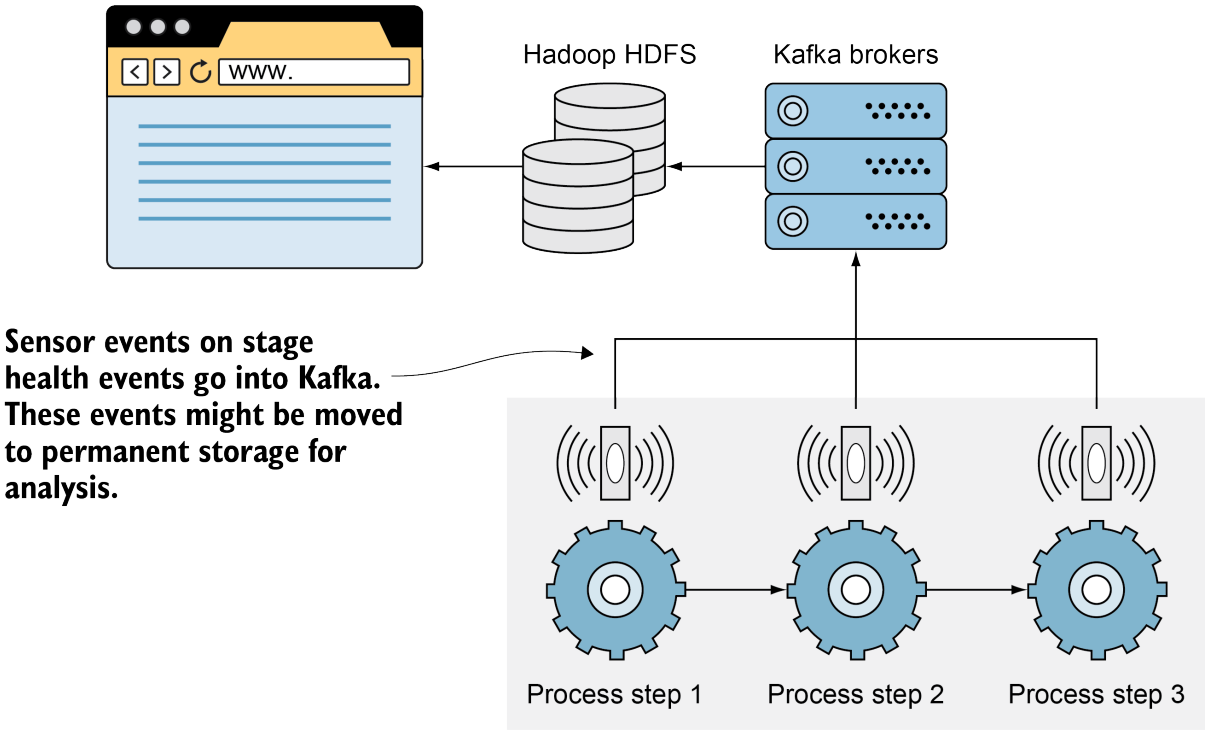

Figure 3.4 Alert Trending Use Case

Figure 3.4 shows that there is a sensor watching each stage in the process. Those equipment alert events go into Kafka. While not an immediate concern to our system, Kafka enables us to pull that data into other data storage or processing systems like Hadoop.

Table 3.3 highlights that the goal is to group the stage alert results by stage and that we are not concerned about losing a message from time-to-time.

Table 3.3 Alert Trending Checklist

Kafka Feature	Concern?
Message Loss	
Grouping	X
Ordering	
Last value only	
Independent Consumer	X

As for alerting on statuses, we will also want to group by a key, which will be the process stage.

However, we do not care about past states of the sensor, but rather the current status. In other words, the current status is all we care about and need for our requirements. The new status will replace the old, and we do not need to maintain a history. The word replace here is not entirely correct (or not what we are used to thinking). Internally, Kafka will add the new event that it receives to the end of its log file, like any other message it receives. After all, the log is immutable and can only be appended to the end. So how does Kafka make what appears to be an update happen? It uses a process called log compaction, which we will dig into in the next chapter. Another difference we will have with this requirement is the usage of a consumer assigned to specific alert partitions. Critical alerts will be processed first due to an uptime service level agreement (SLA) in which those events need to be handled quickly. Figure 3.5 shows an example of how critical alerts could be sent to Kafka and then consumed to populate an operator's display to get attention quickly.

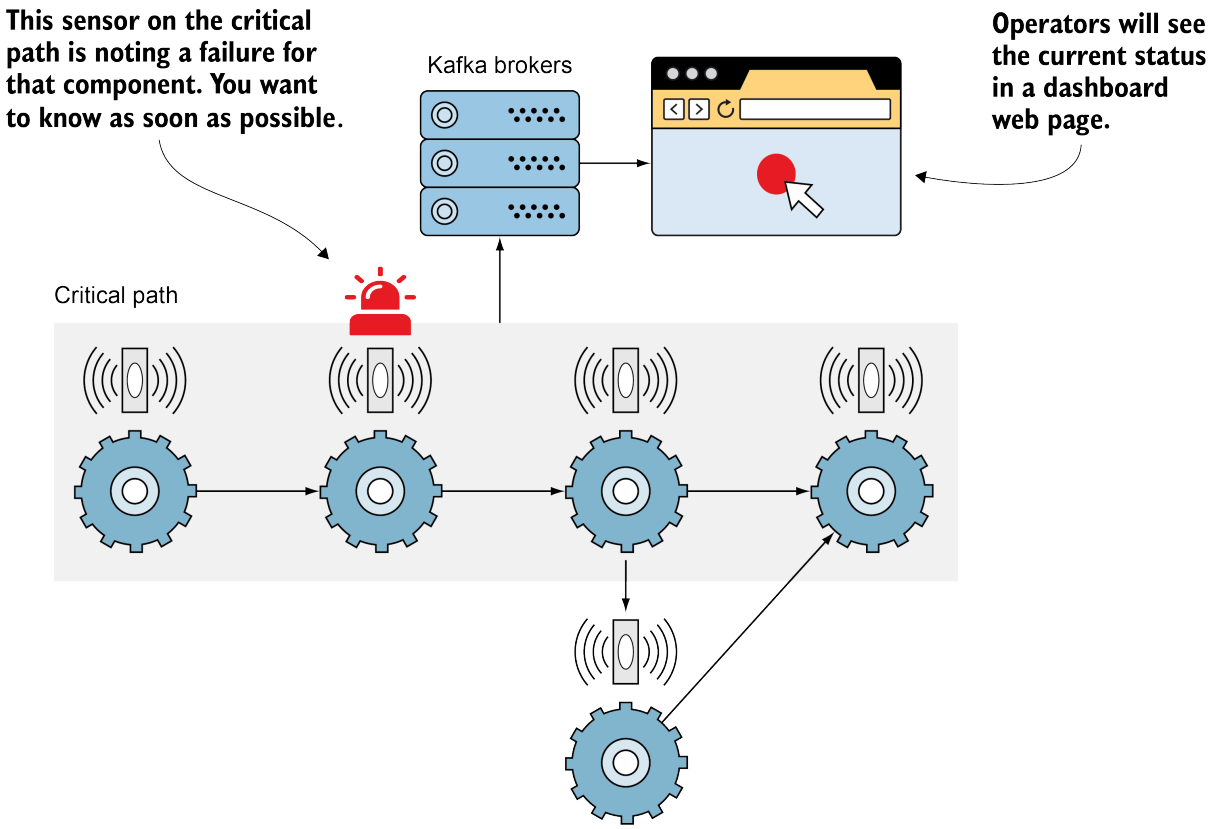

Figure 3.5 Alert Use Case

Table 3.4 reinforces the idea that we want to group an alert to the stage it was created in and know the latest status only.

Table 3.4 Alert Checklist

Kafka Feature	Concern?
Message Loss	
Grouping	X
Ordering	
Last value only	X
Independent Consumer	

Taking the time to plan out our data requirements will not only help us clarify our application requirements but hopefully validate the use of Kafka in our design.

3.2.6 Reviewing our blueprint

One of the last things to think about is how we want to keep these groups of data organized. Logically, the groups of data can be thought of in the following manner:

- audit data
- alert trend data
- alert data

For those of you already jumping ahead, let's keep in mind that we might use our alert trend data as a starting point for our alerts topic: i.e., you can use one topic as the starting point to populate another topic. However, to start in our design, we will write each event type from the sensors to their logical topic to make our first attempt uncomplicated and easy to follow. In other words, all audit events will end up on an audit topic, all alert trend events will end up on a alert trend topic, and our alert events on an alert topic. This 1-to-1 mapping will make it easier to focus on the requirement at hand for the time being.

3.3 Data Format

One of the easiest things to skip, but critical to cover in our design, is the format of our data. XML, JSON, or comma-separated values (CSV) are pretty standard formats that help define some sort of structure to our data. However, even with a clear syntax format, there can be information missing in your data. What is the meaning of the first column or the third one? What is the data type of the field in column 2 of a CSV file? The knowledge of how to parse or analyze our data can be hidden in applications that repeatedly pull the data from its storage location. Schemas are a means of providing some of this needed information in a way that can be used by our code, or other applications that may need the same data.

If you look at the Kafka documentation, you may have noticed references to another serialization system called Apache Avro™. Avro provides schema definition support as well as storage of schemas in Avro files. Let's take a closer look at why this format is commonly used in Kafka.

3.3.1 Schemas for Context

One of the significant gains of using Kafka is that the producers and consumers are not tied directly to each other. Further, Kafka does not care about the content of the data or do any validation by default. Kafka understands how to quickly move bytes and lets us have the freedom to put whatever data you need into the system. However, there is likely a need for each process or application to understand the context of what that data means and what format is in use. By using a schema, we provide a way for our applications to understand the structure and intent of the data. The definition doesn't have to be posted in a README for others in the organization to determine data types or try to reverse-engineer from data dumps. Listing 3.8 shows an example of a schema defined as JSON. Fields can be defined with details such as the name, type, and any default values. Looking at the field `daysOverDue`, the schema tells us that the days a book is overdue will be an 'int' in this case with a default value of 0. Knowing the fact that this will be numeric and not text, such as "one week", helps to create a clear contract for the producers and consumers of the data.

Listing 3.8 Avro schema example

```
{
    "type" : "record",                    ❶
    "name" : "libraryCheckout",
    "namespace" : "org.kafkainaction",
    "fields" : [{"name" : "materialName",
                 "type" : "string",
                 "default" : ""},

                {"name" : "daysOverDue",    ❷
                 "type" : "int",    ❸
                 "default" : 0},    ❹

                 {"name" : "checkoutDate",
                  "type" : "int",
                  "logicalType": "date",
                  "default" : "-1"},

                  {"name" : "borrower",
                   "type" : {
                         "type" : "record",
                         "name" : "borrowerDetails",
                         "fields" : [
                            {"name" : "cardNumber",
                             "type" : "string",
                             "default" : "NONE"},
                            {"name" : "residentZipCode",
                             "type" : "string",
                             "default" : "NONE"}
                          ]},
                          "default" : {}
                }
    ]
}
```

❶ JSON-defined Avro schema

❷ Direct mapping to a field name

❸ Field with name, type, and default define

❹ Default value provided

By looking at another example of the Avro schema in Listing 3.8, we can see that questions such as do we parse the resident's zip code as a number or a string (in this case, string) are easily answered by a developer looking at the schema. Applications could automatically use this information to generate data objects for this data, which helps to avoid parsing data type errors.

For relational database users, you might say how is this different than using a table that has a schema defined? After all, a schema is usually associated with one topic, just like a database schema is associated with one table. However, as we will discuss, schemas can be used by tools like Avro to handle data that evolves. Most of us have dealt with alter statements or tools like Liquibase™ to work around these changes. With schemas, we are starting with the knowledge that our data will probably change upfront. Do we need a schema when we are first starting with our data designs? One of the main concerns is that if our system's scale keeps getting larger, will you be able to control the correctness of data? The more consumers you have could lead to a burden on testing that you would need to do. Besides the growth in numbers alone, you might not even be able to know of all of the consumers of that data.

3.3.2 Usage of Avro

Now that we have discussed some of the advantages of using a schema, why would we look at Avro? First of all, Avro always is serialized with its schema. While not a schema itself, Avro supports schemas when reading and writing data and can apply rules to handle schemas that can change over time. Also, if you have ever seen JSON, it is pretty easy to understand Avro. Besides the data itself, the schema language itself is defined in JSON as well. The ease of readability does not have the same storage impacts of JSON, however. A binary representation is used for efficient storage. The interesting point is that Avro data is serialized with its schema. If the schema changes, you can still process data [28]. The old data will use the schema that existed as part of its data. On the other hand, any new formats will use the schema present in their data. Figure 3.6 shows how even though Kafka can work with Avro, the brokers do not treat Avro data formats differently. Clients are the ones who gain the benefit of dealing with Avro.

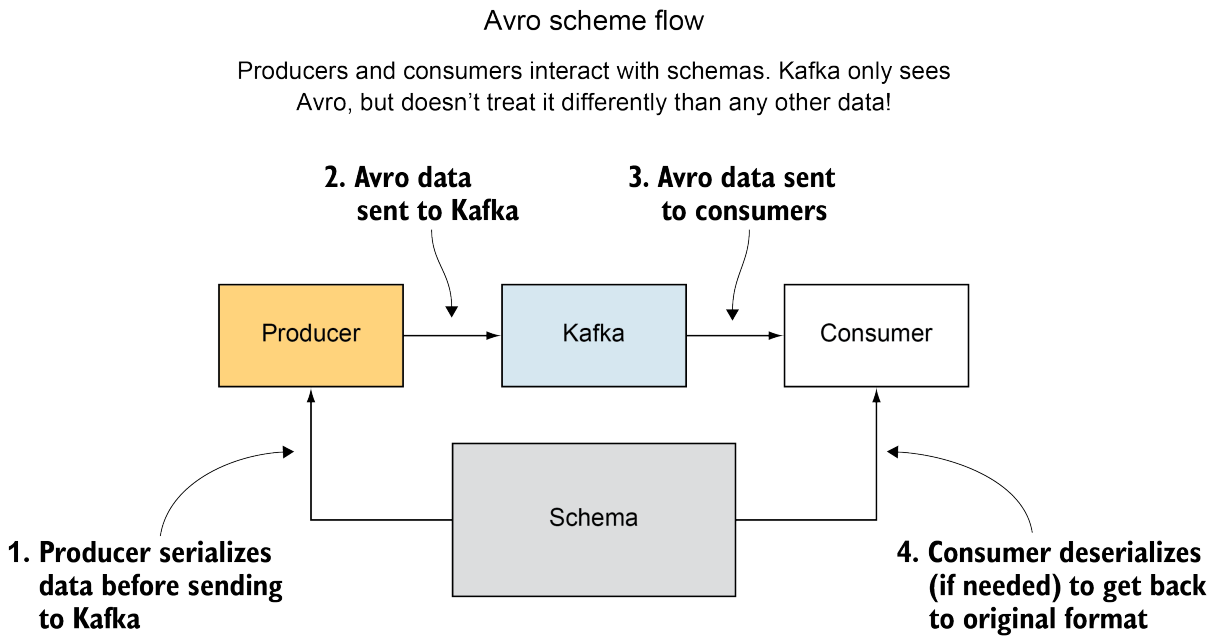

Figure 3.6 Avro Flow

Another benefit of looking at Avro is the popularity of its usage. I first saw its usage on various Hadoop efforts, and it can be used in many other applications. Bindings exist for many programming languages and should not be hard to find in general. Those who have past bad experiences and prefer to avoid generated code, can use Avro dynamically without code generation.

Let's get started with how we will use Avro by adding it to our `pom.xml`, as shown in Listing 3.9 [29]. If you are not used to the pom.xml or Maven, you can find this file at our project's root.

Listing 3.9 Add Avro to pom.xml

```
<dependency>
  <groupId>org.apache.avro</groupId>   ❶
  <artifactId>avro</artifactId>
  <version>${avro.version}</version>
</dependency>
```

❶ This entry will be added as a dependency to the project's pom.xml file

Since we are already modifying the pom file, let's go ahead and include a plugin that will generate the Java source code for our schema definitions. As a side note, you can also generate the sources from a standalone java jar titled `avro-tools` if you do not want to use a Maven plugin. For those who do not prefer code generation in their source code projects, it is not a hard requirement [30]. Listing 3.10 shows adding the avro-maven-plugin plugin to our pom.xml suggested by the Apache Avro Getting Started with Java documentation site [31]. This entry also

lets Maven know that we want to generate source code for the Avro files found in the source directory we list and output the generated code to the specified output directory. You can change the source and output locations to match your specific project structure.

Listing 3.10 Adding Avro Maven plugin to pom.xml

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>                                    ❶
  <version>${avro.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>                                           ❷
      <goals>
        <goal>schema</goal>                                                     ❸
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>  ❹
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>       ❺
      </configuration>
    </execution>
  </executions>
</plugin>
```

❶ Artifact id needed in our pom.xml as a plugin

❷ Maven phase configuration

❸ Maven goal configuration

❹ Directory scanned for Avro files

❺ Generated Java code placement location

Let's start defining our schema by thinking about the data types we will be using, beginning with our alert status scenario. To start, we'll create a new file named alert.avsc with a text editor. Listing 3.11 shows a schema definition with the namespace field that defines the generated package we want the source code to be created in. Alert will be the name of the Java class we will interact with after the generation of source code from this file.

Listing 3.11 Alert Schema: alert.avsc

```
{
  "namespace": "org.kafkainaction",     ❶
  "type": "record",
  "name": "Alert",      ❷
  "fields": [     ❸
    {
      "name": "sensor_id",
      "type": "long",
      "doc": "The unique id that identifies the sensor"
    },
    {
      "name": "time",
      "type": "long",
      "doc": "Time the alert was generated as UTC milliseconds from the epoch"
    },
    {
      "name": "status",
      "type": {
        "type": "enum",
        "name": "AlertStatus",
        "symbols": [
          "Critical",
          "Major",
          "Minor",
          "Warning"
        ]
      },
      "doc": "The allowed values that our sensors will use to emit current status"
    }
  ]
}
```

❶ Namespace determines the generated package

❷ Alert will be the name of the created Java class

❸ The fields defining data types and documentation notes

Listing 3.11 shows a definition of alerts. One thing to note is that "doc" is not a required part of the definition. However, there is certainly value in adding details that will help future producer or consumer developers understand what the data means in context. The hope is to stop others from inferring our data's meaning and be more explicit about the content. For example, the field "time" always seems to bring developer anxiety when seen. Is it stored in a string format, time-zone information included, or does it include leap seconds? The doc field can provide that information. The namespace provided turns into the Java package for the generated Java class. The various field definitions include the name as well as a type.

Now that we have the schema defined let's run the Maven build to see what we are working with. `mvn generate-sources` or `mvn install` can generate the sources in our project. This should give us a couple of generated classes: `org.kafkainaction.Alert.java` and `org.kafkainaction.AlertStatus.java` we can now use in our examples.

While we have been focusing on Avro itself, the remaining part of the setup is related to the

changes we need to make in our producer and consumer clients to use the schema we have created.

While we could always create our own serializer for Avro, we already have an excellent example provided by Confluent. Access to those existing classes is accomplished by adding the `kafka-avro-serializer` dependency to our build [32].

Listing 3.12 Adding Kafka serializer to pom.xml

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>     ❶
    <version>${confluent.version}</version>
</dependency>
```

❶ This entry will also be added as a dependency in the project's pom.xml file

Listing 3.12 shows the pom.xml entry we would add. This is needed to avoid having to create our own Avro serializer and deserializer for the keys and values of our events.

NOTE **Confluent Avro Maven Repository**

If you are using Maven to follow along: Make sure that you add the following to your pom.xml repository. This information is needed to let Maven know where to get specific dependencies [33].

```
<repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
</repository>
```

With the build setup and our Avro object ready to use, let's take our example producer from the last chapter `HelloWorldProducer`, and slightly modify the class to use Avro. Listing 3.13 shows the pertinent changes to the producer class-not including imports. Notice the use of `io.confluent.kafka.serializers.KafkaAvroSerializer` as the value of the property `value.serializer`. This will handle the Alert object that we create and send to our new `avrotest` topic. Before, we could use a string serializer, but with Avro, we need to define a specific value serializer to tell the client how to deal with our data. The use of Alert, rather than a string, shows how we can utilize types in our applications as long as we can serialize them. This example also makes use of the Schema Registry. We will cover more details of the Schema Registry in chapter 11, but at this point, it is helpful to know that the property `schema.registry.url` points to the URL of our registry. This registry can have a versioned history of schemas and helps us manage schema evolution.

Listing 3.13 Producer using Avro Serialization

```
public class HelloWorldProducer {

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

  public static void main(String[] args) {
    Properties producerProperties = new Properties();
    producerProperties.put("bootstrap.servers",
                           "localhost:9092,localhost:9093,localhost:9094");
    producerProperties.put("key.serializer",
                           "org.apache.kafka.common.serialization.LongSerializer");
    producerProperties.put("value.serializer",
                           "io.confluent.kafka.serializers.KafkaAvroSerializer");     ❶
    producerProperties.put("schema.registry.url", "http://localhost:8081");     ❷

    try (Producer<Long, Alert> producer = new KafkaProducer<>(producerProperties)) {
      Alert alert = new Alert(12345L, Instant.now().toEpochMilli(), Critical);    ❸

      log.info("Alert -> {}", alert);

      ProducerRecord<Long, Alert> producerRecord =
          new ProducerRecord<>("avrotest",
                               alert.getSensorId(),
                               alert);    ❹

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

❶ KafkaAvroSerializer for our custom Alert value.

❷ Pointing to the URL of the registry.

❸ Creating an alert.

❹ Type of the generic is now `Alert`.

The differences are pretty minor. The types change for our Producer and ProducerRecord definitions, and we change the configuration settings for the `value.serializer`.

Now that we have produced messages using Alert, the other changes would be on the consumption side of the messages. For a consumer to get the values produced to our new topic, it will have to use a value deserializer, in this case, `KafkaAvroDeserializer`. This deserializer works to get back the value that was serialized by the producer. This code can also reference the same Alert class generated in the project from the custom Avro file `alert.avsc`. Listing 3.14 shows the significant changes for the consumer class `HelloWorldConsumer`.

Listing 3.14 Consumer using Avro Serialization

```
public class HelloWorldConsumer {

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

  private volatile boolean keepConsuming = true;

  public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "helloconsumer");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("key.deserializer",
              "org.apache.kafka.common.serialization.LongDeserializer");
    props.put("value.deserializer",
              "io.confluent.kafka.serializers.KafkaAvroDeserializer");      (1)
    props.put("schema.registry.url", "http://localhost:8081");

    HelloWorldConsumer helloWorldConsumer = new HelloWorldConsumer();
    helloWorldConsumer.consume(props);

    Runtime.getRuntime().addShutdownHook(new Thread(helloWorldConsumer::shutdown));
  }

  private void consume(Properties props) {
    try (KafkaConsumer<Long, Alert> consumer = new KafkaConsumer<>(props)) {
      consumer.subscribe(Collections.singletonList("avrotest"));      (2)

      while (keepConsuming) {
        ConsumerRecords<Long, Alert> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<Long, Alert> record : records) {      (3)
          log.info("[Consumer Record] offset = {}, key = {}, value = {}",
                   record.offset(),
                   record.key(),
                   record.value());
        }
      }
    }
  }

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

(1) KafkaAvroDeserializer defined due to Alert usage

(2) KafkaConsumer typed to handle Alert values

(3) ConsumerRecord updated to handle Alert values

As with the producer, the consumer client did not require many changes due to the power of updating the configuration deserializer and Avro!

Now that we have some ideas about the 'what' we want to accomplish and our data format, we are well equipped to tackle the 'how' in our next chapter. While the task of sending data into Kafka is straightforward, there are various configuration-driven behaviors that we can use to help us satisfy our specific requirements.

3.4 Summary

- Designing a Kafka solution first involves understanding our data. These details include how we need to handle data loss, ordering of messages, and latency in our use-cases.
- The need to group data will determine whether we will key the messages in Kafka.
- Leveraging Avro schemas definitions can not only help us generate code but can also help us handle data changes in the future. These schemas can be used with our own custom Kafka clients.
- Kafka Connect provides existing connectors to write to and from various data sources. To implement a solution moving data into or out of Kafka, searching the existing source and sink connectors available can help us quickly get started with the Kafka ecosystem.

4 Producers: sourcing data

This chapters covers

- Exploring the producer write path and how it works
- Creating custom producer serializers and partitioners
- Examining various configuration options that we can apply to solve a company's requirements

In the previous chapter, we started to look at some of the requirements that an organization might have regarding their data. Some of the design decisions that we made have practical impacts on how we will need to send data into Kafka. Let's enter the world of an event-streaming platform through the portal gate of Kafka Producer.

After reading this chapter, you will be on your way to solving fundamental requirements by producing data in a couple of different ways. The producer, despite its importance, is only one part of this system. In fact, some producer configuration options can be changed or set at the broker or topic level. We will discover those options as we get further along. However, getting data into Kafka is our first concern in this chapter.

4.1 Introducing the Producer

The producer will be the way we push data into the Kafka system.

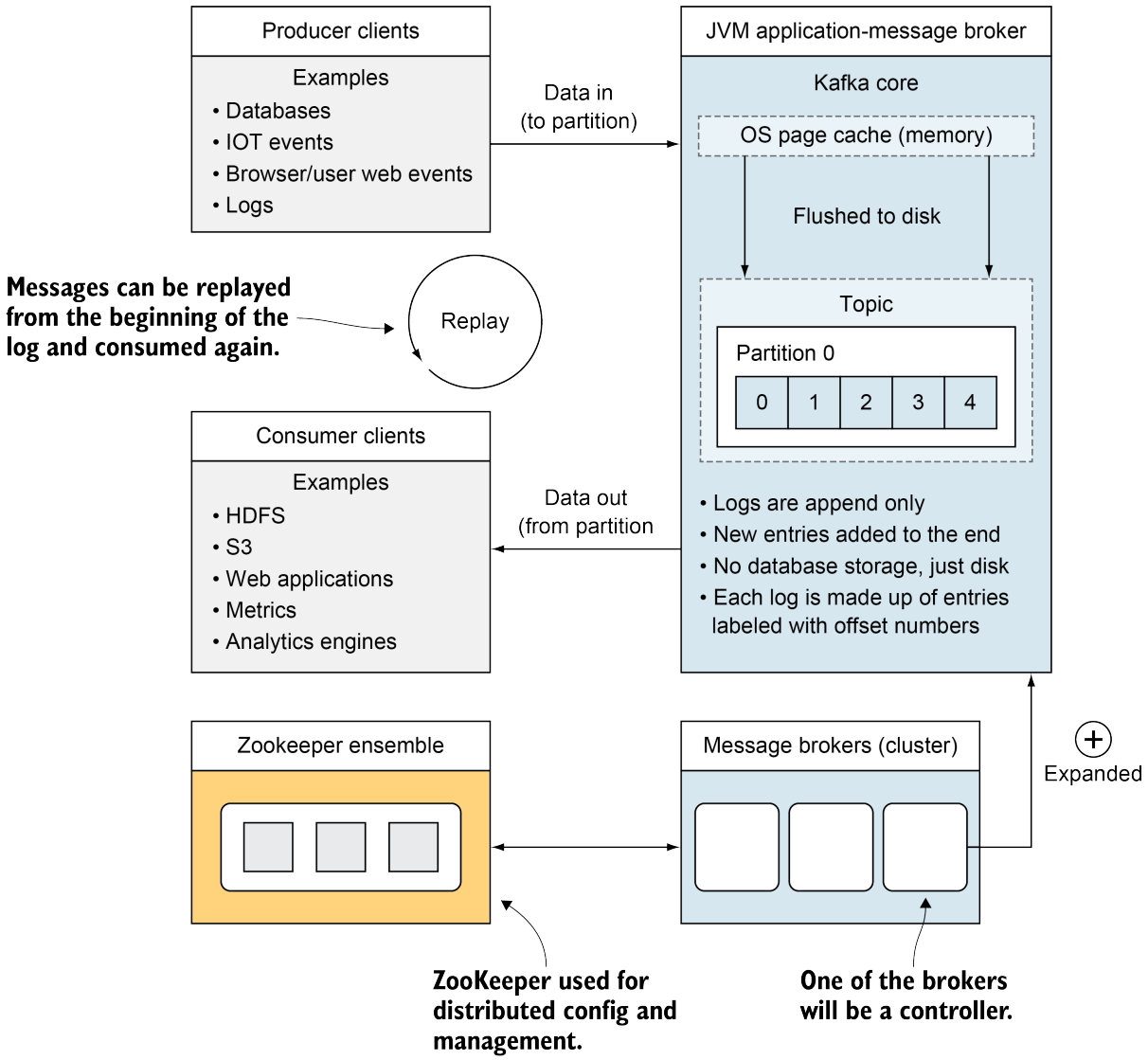

Figure 4.1 Kafka Producers

For a refresher from the last chapter, let's look at where producers fit in the overall context of Kafka. Looking at Figure 4.1, let's focus on the top left of the figure, which shows examples of data being produced into Kafka. This data could be anything from database records, IoT events, web events of users, or logs, to name a few. To make it more concrete, let's imagine a practical example that we might have written for one of our projects already. Let's look at an application that takes user feedback on how a website is working for customers. Right now, the user submits a form on the website that generates an email to a support account or chatbot. Every now and then, one of our support staff checks that inbox to see what suggestions or issues that customers have encountered. Looking to the future, we want a way to keep this information coming to us, but maybe 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 responses rather than just reactive email responses to customers. The benefit of flexibility comes out of having the event in Kafka for any consuming applications to leverage.

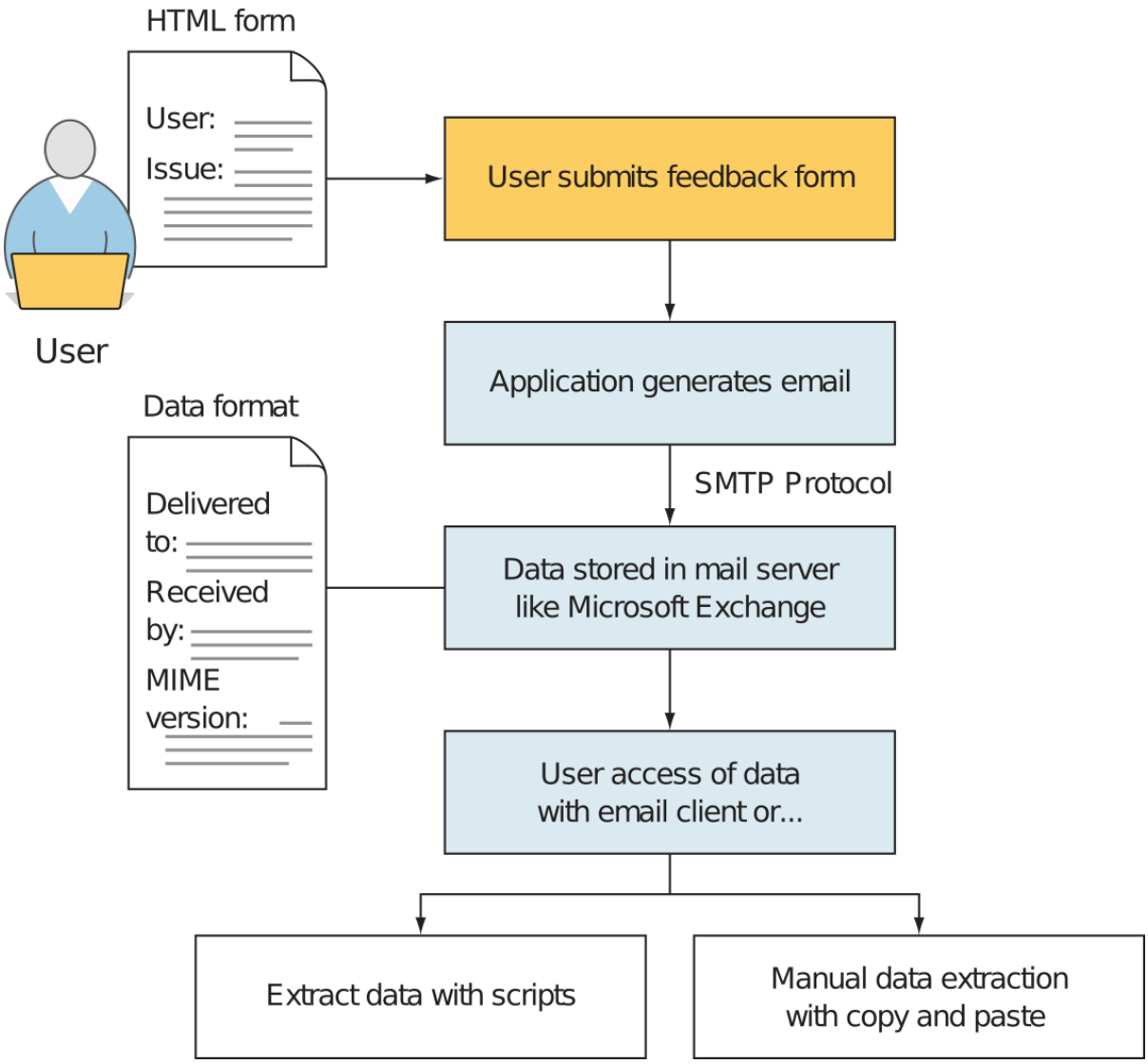

Figure 4.2 Sending data in email

Let's look at what using email as part of our data pipeline impacts. Looking at Figure 4.2 of our email use-case, it might be helpful to focus on the format that the data is stored in once a user submits a form about feedback on a website. A traditional email will use the SMTP protocol, and we will see that reflected in how the email event itself is presented and sometimes stored. Email clients (like Microsoft® Outlook®) can be used to retrieve the data quickly, but how else can we pull data out of that system for other uses rather than just reading email? Copy and paste are common manual steps, of course, as well as email parsing scripts. Parsing scripts would include using a tool or programming language and libraries or frameworks to get the parsing correct. In comparison, while Kafka uses its own protocol, it does not reflect that in the message data. We should be able to write the data in whatever format we choose.

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

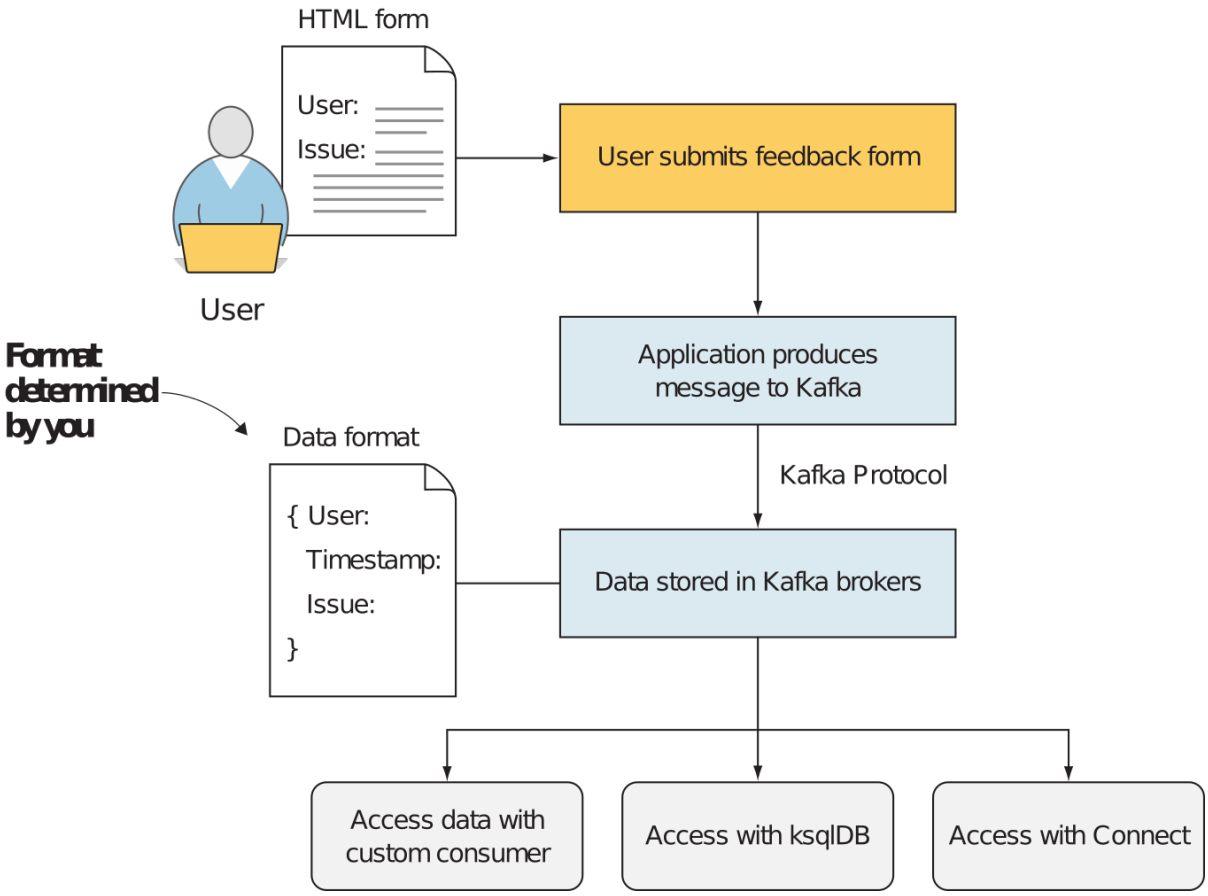

Figure 4.3 Sending data to Kafka

One other usage pattern that comes to mind is treating notifications of customer issues or website outages as temporary alerts that can be deleted 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 have been reported by customers? Does the site always slow to a crawl after sale coupon codes go out in mass marketing emails? Or perhaps this data could help us find features that our users are missing from our site? Do 40% of our user-reported emails involve having trouble finding the 'Privacy' settings on their account? Having this data present in a topic that can be replayed or read by applications with different purposes can add more value than a support or bot email back to the customer that is automated and deleted. Also, if we did have retention needs, it would be controlled by the teams running our email infrastructure vs. a configuration setting we could control with Kafka. Looking at Figure 4.3, notice that the application still has an HTML form but writes to a Kafka topic and not to an email server. We can extract the information that is important for us in whatever format we need, and it can be used in many different ways. Consuming applications can use schemes to know how 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 since we control the retention of those events.

Now that we have seen an example of why we might use a producer let's quickly look at the

write path for a producer interacting with the Kafka brokers.

4.1.1 Key Producer Write Path

While it took relatively few lines of code to send a message as seen in Chapter 2, the Java client producer write process is doing various tasks behind the scenes.

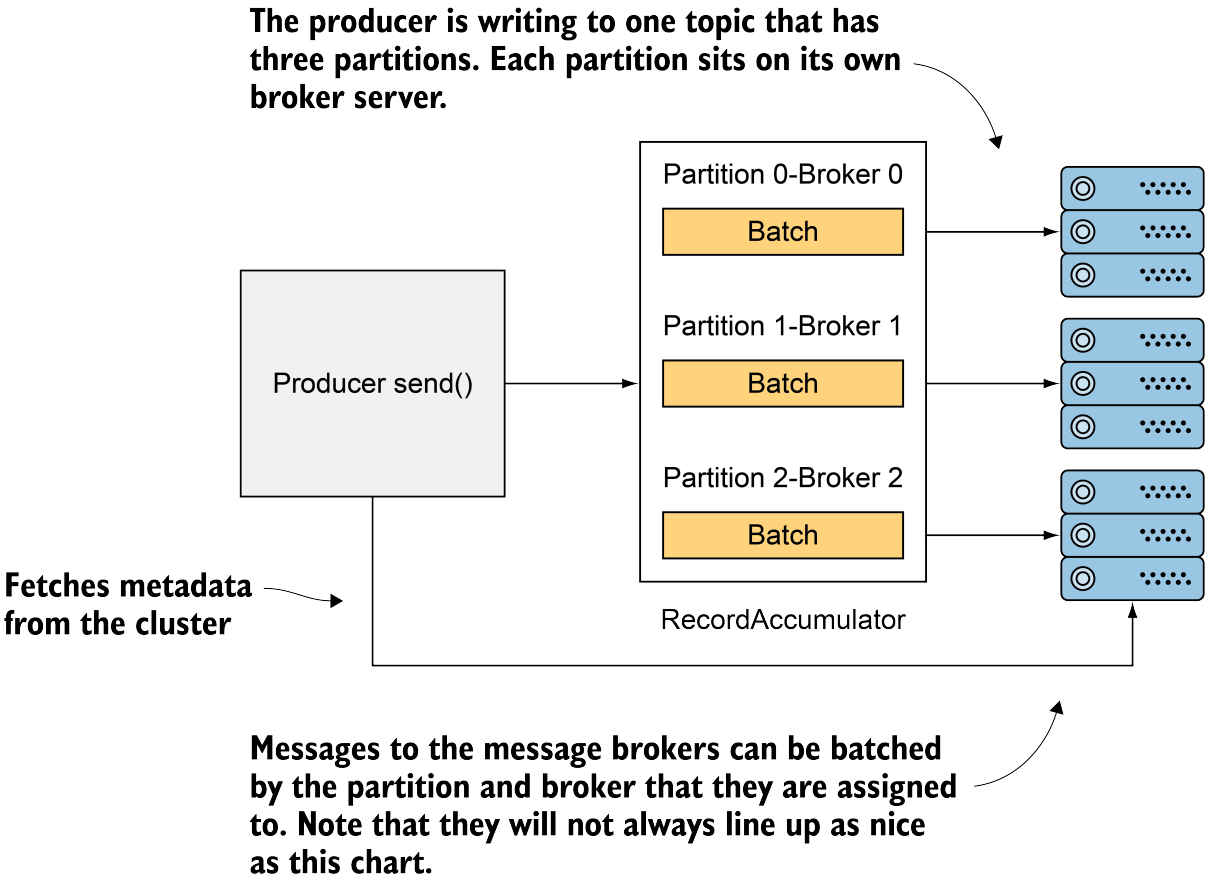

Figure 4.4 Producer Write Path

Looking at Figure 4.4 shown, calling `send()` in our code puts some impressive machinery into motion. The `Sender` class provides the plumbing to work with the `RecordAccumulator` to send requests to the Kafka brokers from a producer [34]. The sender's job includes fetching metadata about the cluster itself. Since producers only write to the replica leader of the partition they are assigned, the metadata helps the producer find out which actual broker to write to since the producer record might only have been given a topic name without any other details. This is nice because the end-user of the producer does not have to make a separate call to get that information. The end-user needs to have at least one running broker to connect to, and the Java client library can figure out the rest.

Continuing with the write path flow, we can see that the record accumulator's job is to *accumulate* the messages into batches. These batches are grouped by broker and partition. Why did I just mention the word batch? Isn't that a bad word when discussing stream processing?

Collecting messages is really about improving throughput. If each message was sent one at a time, we would see significantly slower throughput. Another option for improving throughput is to compress messages. If compression for the messages is used, along with batching, the full batch is compressed. This likely would allow for a higher compression ratio than one message at a time.

IMPORTANT	If one message in the batch fails during delivery, then the entire batch fails for that partition.

Since this distributed system is designed to account for transient errors, like a network blip, the logic for retries is built-in already. However, if the ordering of the messages is essential, then besides setting the retries to a non-zero number, we will 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** to provide the best situation for making sure your producer's messages arrive in the order you intend [35].

Another option to be aware of is using an *idempotent* producer. In effect, the desire is for a producer to only write a message once. The term *idempotent* is referring to how sending the same message multiple times will only result in the message being written once. These settings can also be set with the configuration property `enable.idempotence=true` as well as the same `acks` and `retries` values mentioned in the previous paragraph. The value for `max.in.flight.requests.per.connection` can be a number up to 5 [36]. Despite being able to have more than one in-flight request, the safest method to keep the order of your messages (in addition to the exactly-once delivery) is to set it at 1. Another item to note is that the idempotent delivery is for the lifetime of a producer. If we have another producer sourcing the same data or even a restart of that producer code, we might see duplicate messages on your topic.

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 as data will not be overwritten, but handled by the broker itself and appended to the broker's log [37].

Now it is time to look at how to enable compression and the values like `max.in.flight.requests.per.connection` in code. Let's take a look at how the producer uses these configurations to meet our requirements.

4.2 Important Options

One of the things that was interesting when I started working with sending data into Kafka was the ease of setting options. Specifically, using the Java clients that we will focus on in this book.

If you have worked with other queue or messaging systems, it seems like the setup can include

things like providing remote and local queues lists, manager hostnames, starting connections, connection factories, sessions, and more. While far from being set up free, the producer will work 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 will fetch metadata about brokers and partitions which it will use for all subsequent writes.

As mentioned earlier, Kafka allows you to change key behaviors just by changing some configuration values. Recent versions of the producer have over 50 properties that we could choose to set. 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 kafka.apache.org/27/javadoc/org/apache/kafka/clients/producer/ProducerConfig.html). However, in our examples, we will use the property names themselves for clarity.

4.2.1 Producer Options

Table 4.1 Important Producer Configuration

Key	Purpose
acks	Number of replica acknowledgments producer requires before success is considered
bootstrap.servers	One or many Kafka brokers to connect for startup
value.serializer	The class that is being used for serialization of the value
key.serializer	The class that is being used for serialization of the key
compression.type	The compression type (if any) used to compress messages

Table 4.1 lists some of the most crucial producer configurations that will be helpful in our specific examples. The Kafka documentation has a helpful way of letting us know which options might have the most impact. Look for the **IMPORTANCE** label of High in the documentation listed at kafka.apache.org/documentation/\#producerconfigs. Let's go through some of the configuration options listed that we have not discussed in-depth before and share what they mean in more context.

4.2.2 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. But, how does Kafka know where that topic partition resides? Recall that topics are made up of partitions. However, we do not have to know the details of those partitions when we are sending messages. One of the required configuration options is `bootstrap.servers`. Figure 4.5 shows an example of a producer that has only Broker 0 in its list of bootstrap servers.

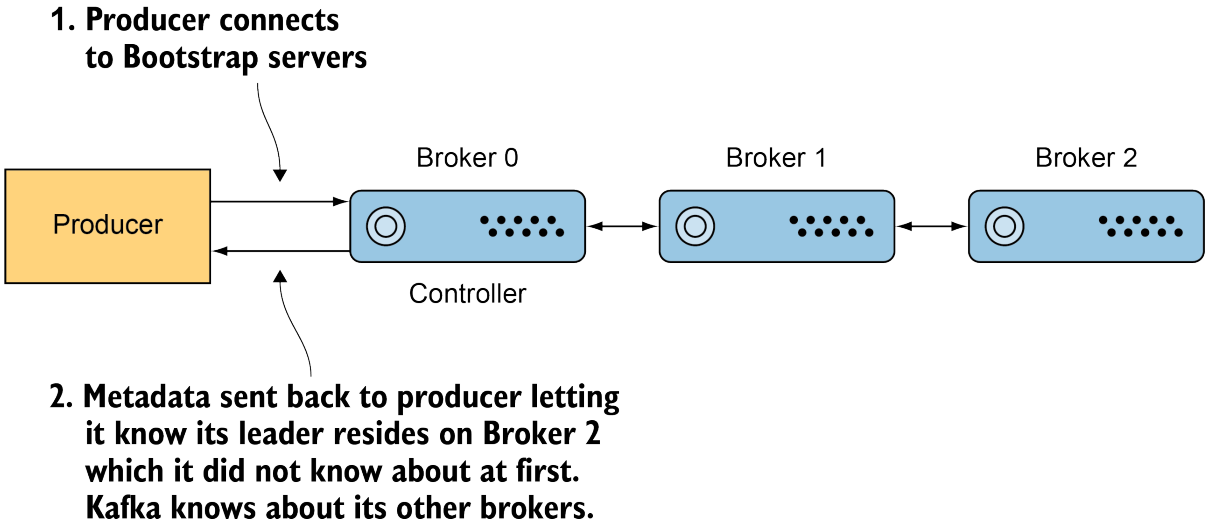

Figure 4.5 Bootstrap Servers

This property can take a list of Kafka brokers or just one, as in Figure 4.5. By connecting to this one broker, the client is able to discover the metadata it needs, which includes data about other brokers in the cluster as well. It is a good idea to send a list rather than one in case one of the servers is down or under maintenance and unavailable [38]. 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 those details we did not have to provide above, such as where the leader replica for the partition resides on disk. Producer clients can also overcome a failure of the partition leader they are writing to by using the information about the cluster to find the new leader. You might notice that ZooKeeper's information is not part of the configuration needed. Any metadata the producer needs will be handled without the producer client needing to provide ZooKeeper cluster details.

4.2.3 How to go Fast (or 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, but we also have the option of handling success or failure asynchronously with callbacks or `Futures`. 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 will apply to our scenarios 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 complete request. The valid values for this property are 'all,' -1, 1, and 0 [39]. Figure 4.6 shows how a message with ack=0 will behave. Setting this value to 0 will probably get us the lowest latency, but at the cost of durability. Guarantees are not made if any broker received the message, and retries are not attempted. 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 event or hover event. If it is lost, there is no real business impact.

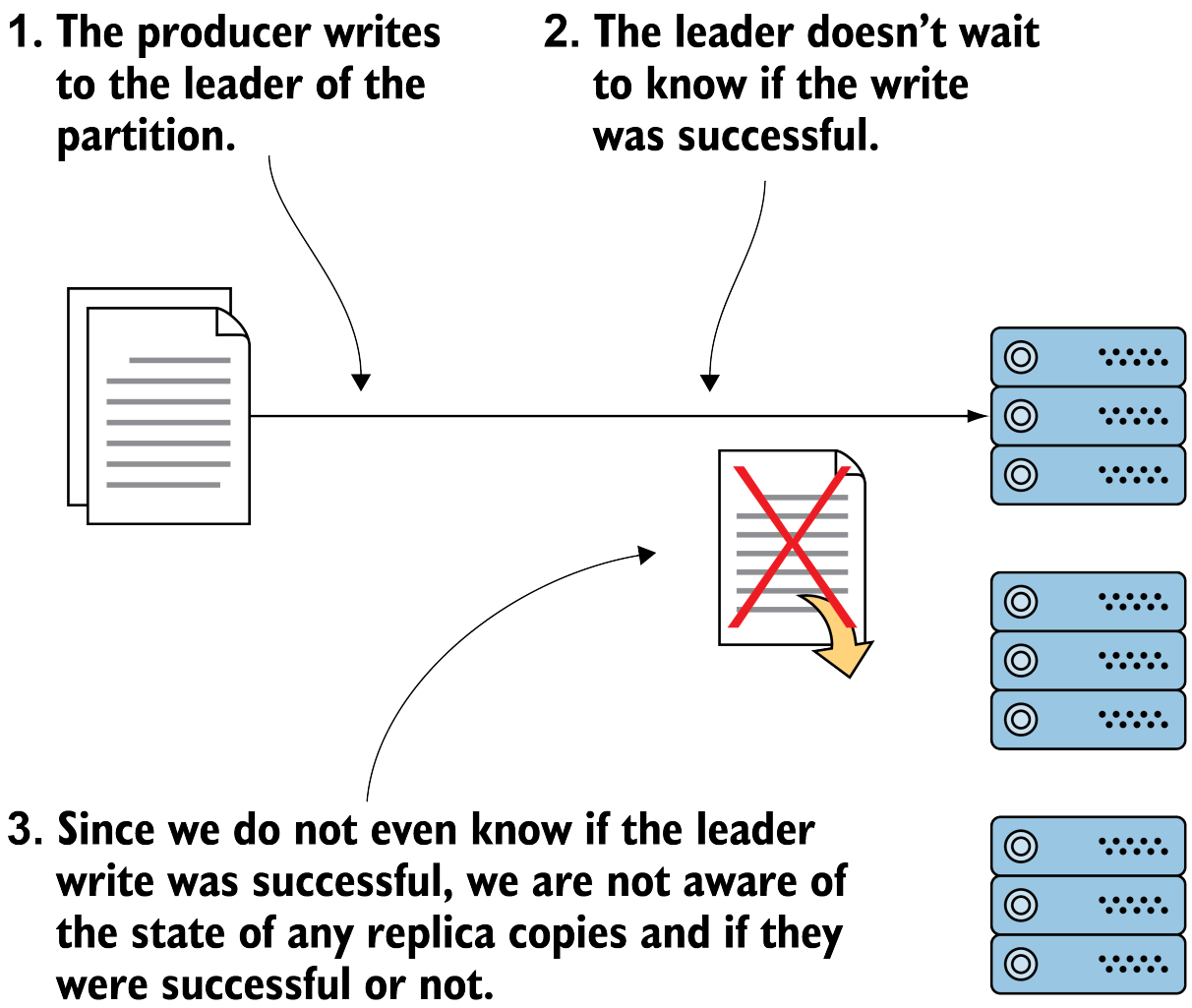

Figure 4.6 Property acks=0

In essence, the event was sent from the producer and forgotten. The message might have never been written to the leader replica of the partition. If the message did make it to the leader replica, the producer does not know if any follower replica copies were successful.

Figure 4.7 shows the impact of setting the `acks` value to 1 and asking for an acknowledgment. An acknowledgment would involve the receiver of the message, the leader replica of the specific partition, sending confirmation back to the producer. The producer client would wait for that acknowledgment. Setting the value to 1 will mean that, at least, the leader replica has received the message. However, the followers might not have copied the message before a failure brings down the leader. If that situation occurs before a copy was made, then the message would never appear on the replica followers for that partition.

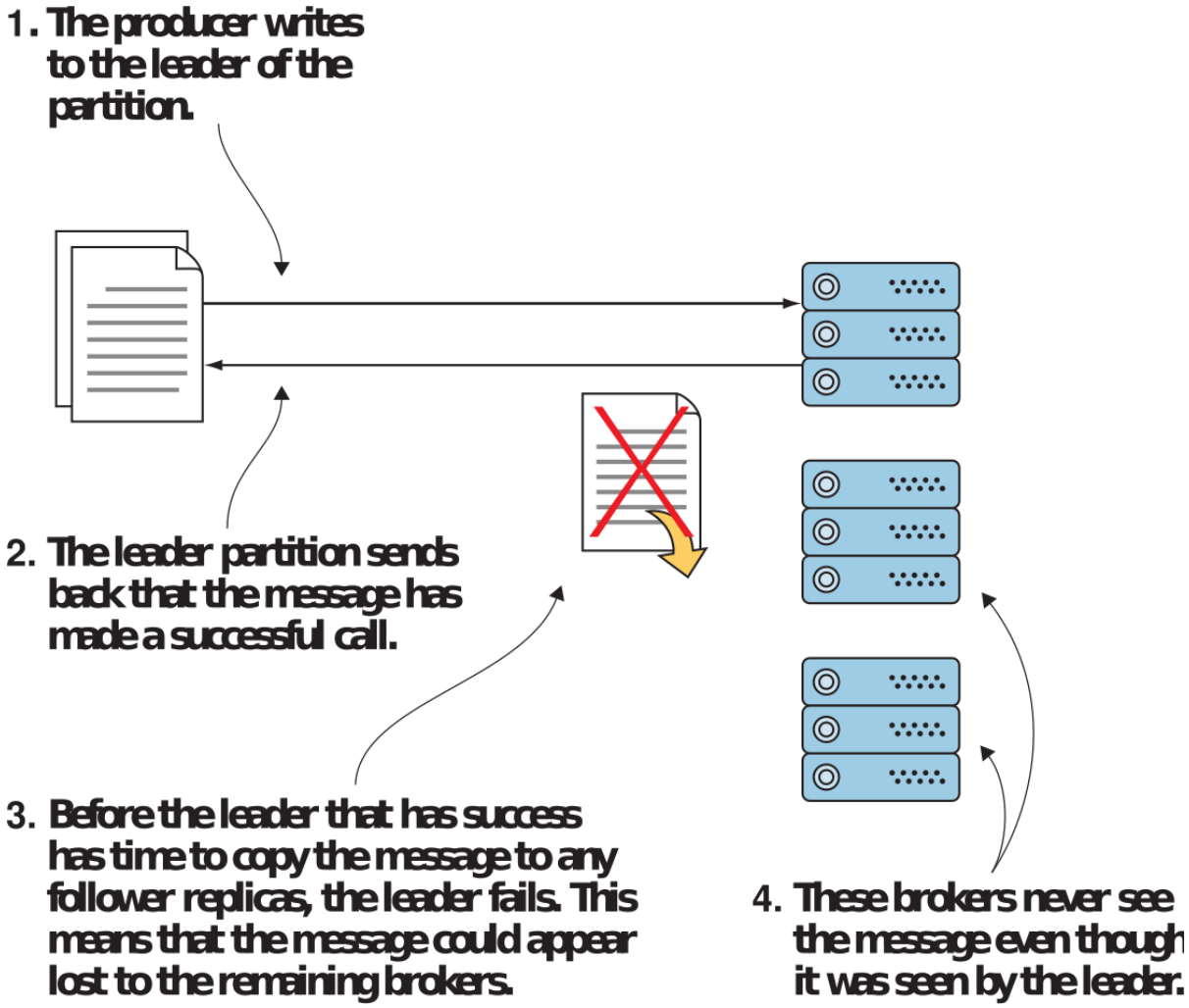

Figure 4.7 Property acks=1

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 could make it appear as if the message never made it to the cluster.

Values all or -1 are the strongest available option. Figure 4.8 shows how this value means that a partition leader replica waits on the entire list of its in-sync replicas (ISRs) to acknowledge it has received the message. While this is better for durability, it is easy to see that it won't the quickest due to the dependencies it has on other brokers. In many cases, it is worth paying the performance price in order to prevent data loss. With many brokers in a cluster, it is important to be aware of the number of brokers the leader will have to wait on. The broker that takes the longest to reply will be the determining factor for how long it takes for the producer to receive a success message.

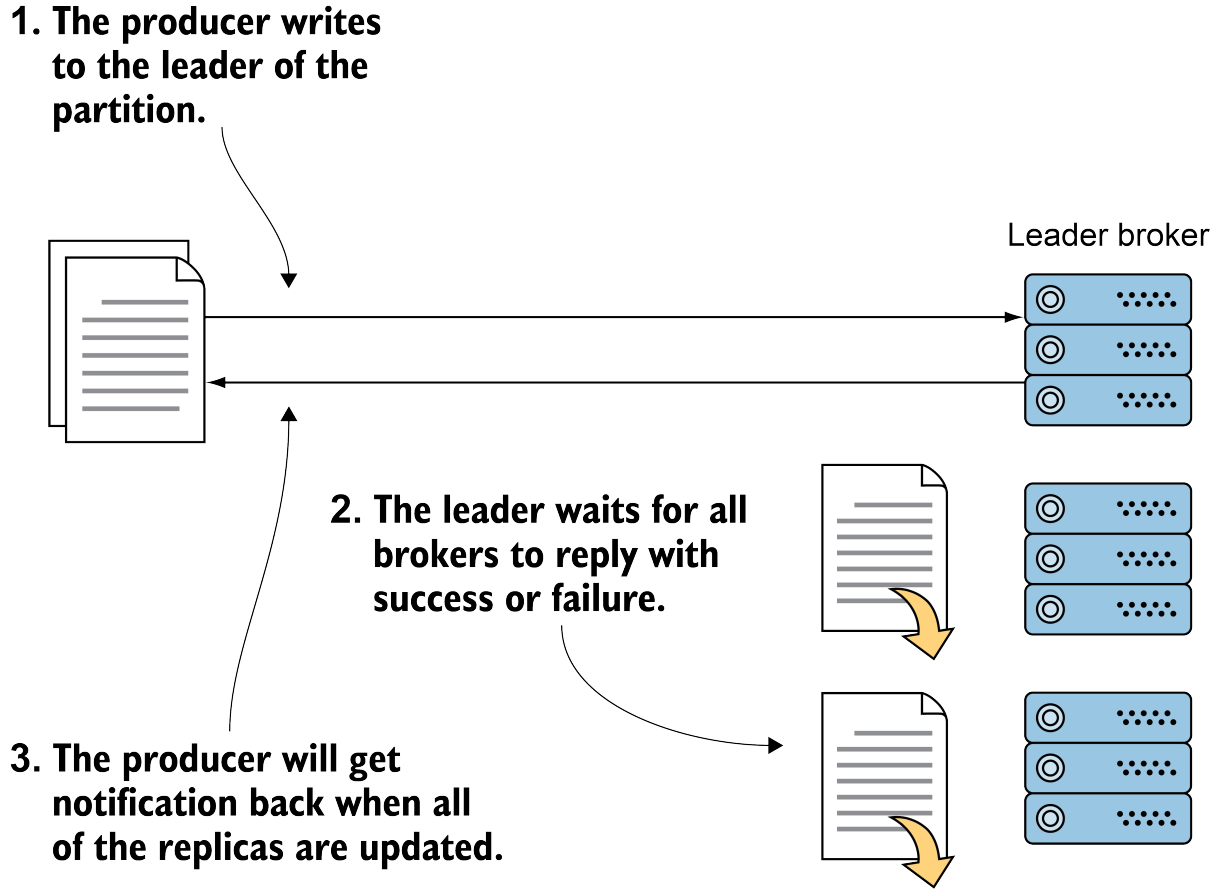

Figure 4.8 Property acks=all

Figure 4.8 shows that the producer will not get an acknowledgment of success until after all replicas for a partition are successful.

NOTE **What Committed Means**

A message will not be seen by consumers until it is considered to be committed. However, this status is not necessarily related to the `acks` setting given in the producer configuration. We will discuss more when the messages become available for consumption later. This involves the data being received by all the replicas in its in-sync replica list [40].

4.2.4 Timestamps

Recent versions of the producer record will have 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 will be used. The actual time that is used in the message can stay this value or can be a broker timestamp that occurs when the message is logged. Setting the topic configuration `message.timestamp.type` to `CreateTime` will use the time set by the client, whereas setting it to `LogAppendTime` will use the broker time [41]. Why would you want to use 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) took 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 actual event time is not business or ordering 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 will be ordered in the log by offsets and not by timestamp. While reading timestamped data is often thought of as a consumer client concern, it is also a producer concern since the producer takes the first steps to ensure message ordering. As discussed earlier, this is also why `max.in.flight.requests.per.connection` is important to consider whether you want to allow retries or many inflight requests at a time. If a retry happens and other requests succeeded on their first attempt, earlier messages might be added after those later ones. Figure 4.9 is an example of when a message can get out of order. Even though message one was sent first, it does not make it into the log in an ordered manner since retries were enabled.

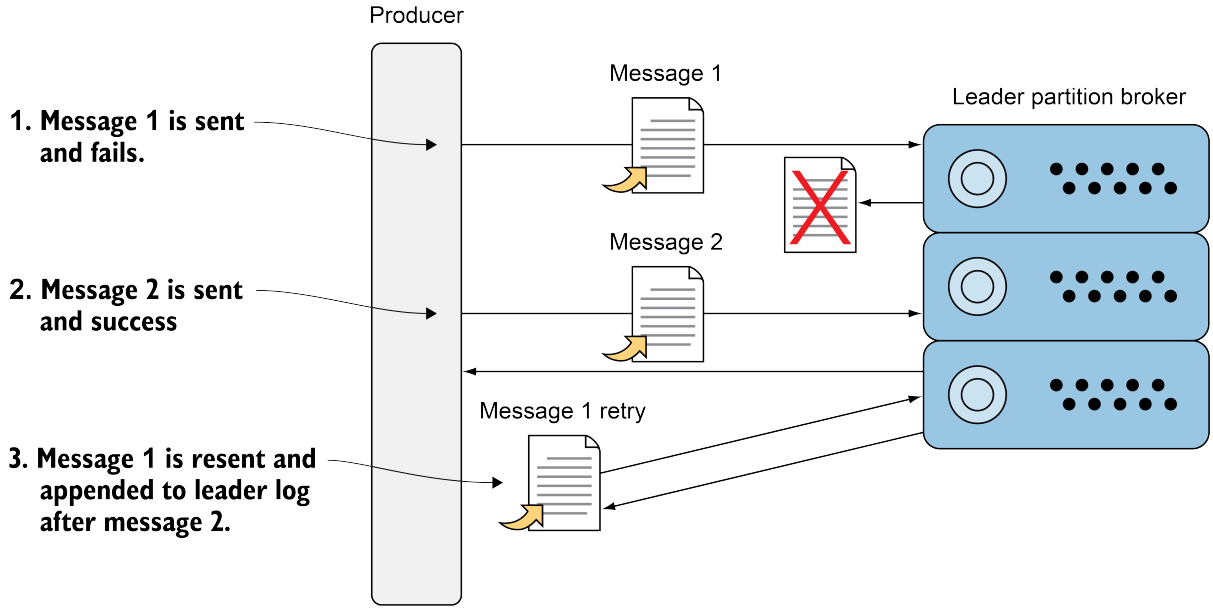

Figure 4.9 Retry Impact on Order

As a reminder, with Kafka versions before 0.10, timestamp information will not be available as

that feature was not included in earlier releases. We can still include a timestamp, but we would need to store it in the value of the message itself.

4.2.5 Adding compression to our messages

One of the topics that we discussed briefly above, when talking about message batches, was compression. Google Snappy, GNU gzip, lz4, zstd (developed at Facebook™), and none are all options that can be used and are set with the configuration key `compression.type` [42]. If the size of data being sent is a concern, as when data needs to be mirrored across data centers, compression is a feature you might want to look into. Depending on the shape of your data, you may want to use this as a way to help save space on your data storage as well. We use `snappy` in our examples for Apache Flume configuration in this book.

Compression is done at the batch level. Size and rate of the data should be considered when deciding whether or not to use compression. Small messages being compressed do not necessarily make sense in all cases. In addition, if you have low traffic, compression might not provide significant benefits.

4.2.6 Custom Serializer

So far, we have used a couple of prebuilt serializers. For plain text messages, our producer has been using serializers that are called `StringSerializer`. And when we started to talk about Avro in Chapter 3, we reached for the class: `io.confluent.kafka.serializers.KafkaAvroSerializer`. But what if we have a specific format to produce? This often happens when trying to work with custom objects. For our context, serialization is used to translate data into a format that can be transmitted, stored, and then retrieved to achieve a clone of our original data.

Listing 4.1 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) {     (1)
    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;
  }
}
```

(1) This bean will hold the id, level, and message of the alerts.

Listing 4.1 shows code that we have used to create a bean named Alert to hold the information we want to send. Those familiar with Java will notice that it 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 leverage it in making a simple custom `Serializer` called `AlertKeySerde`.

Listing 4.2 Custom Serializer

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

  public byte[] serialize(String topic, Alert key) {      ❶
    if (key == null) {
      return null;
    }
    return key.getStageId().getBytes(StandardCharsets.UTF_8);     ❷
  }

  public Alert deserialize(String topic, byte[] value) {      ❸
    //We will leave this part for later
    return null;
  }

  //...
}
```

❶ The topic, as well as the Alert object itself, is sent into our method.

❷ Our end goal is to convert objects to bytes.

❸ The rest of the interface methods do not need any logic at this point.

In Listing 4.2, we are planning on using this custom class only as the key serializer class at the moment and 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. We should be mindful of our intended serializers, and the configuration values for both. The code implements the `Serializer` interface and pulls out just the field `stageId` to use as our key for our message. This example is straightforward since the focus is on the technique of using a Serde. Other options of `Serdes` that are often used are JSON and Avro implementations.

NOTE **What is Serde?**

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

Another thing to keep in mind is that this will involve the consumers knowing how to deserialize the values; however, they 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.

4.2.7 Producer Interceptor

One of the other options when using the producer is creating producer interceptors. They were introduced in KIP-42 whose main goal was to help support measurement and monitoring [43]. It is true that you can modify the message content, and since the interceptor will run before a partition gets assigned, it is important to note any key changes would make to the initial message. If an interceptor changed a key, for example, consumer clients might not know what to expect if the key changes the partition assignment and ordering that they intended. In comparison to using a Kafka Streams workflow to filter or aggregate data or even creating different topics entirely for specifically modified data, the usage of these interceptors might not be our first choice. Kafka also reports various internal metrics using Oracle JMX™ that can be used if monitoring is the main concern.

If you do create an interceptor, remember to set the producer config `interceptor.classes`. At the current time, there are no default interceptors that run in the life cycle. In Chapter 9, a use-case is shown for tracing messages from producer clients to the consumer client with interceptors adding a trace id.

4.2.8 Our own partition code

So far, in our examples of writing to Kafka, the data has been directed to a topic, and no additional metadata has been provided from the client. Since the topics are made up of partitions that sit on the brokers, Kafka has provided a default way to decide where to send a message to a specific partition. The default for a message with no key (which has been used in the examples so far) was a round-robin assignment strategy prior to Apache Kafka version 2.4. Versions after 2.4 changed no key assignments to use a sticky partition strategy in which messages were batched to the same partition until the batch was full, and a new partition was randomly assigned. If a key is present, then the key is used to create a hashed value. That hash is then used to determine a specific partition. However, sometimes we have some specific ways we want our data to be partitioned. One way to take control of this is to write our own custom partitioner class.

The client has the ability to control what partition it writes, too, by configuring a custom partitioner. This can be one way to load balance the data over the partitions. This might come into the picture if we have specific keys that we want to be treated in a special way. Let’s look at an example for our sensor use-case.

One example we can think about is alert levels from our sensor monitoring service that was discussed in Chapter 3. Some sensors' information might be more important than others, i.e., they might be on the critical path of our e-bike that could 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 would place the different levels in different partitions. Our consumer clients could always

make sure to read the critical alerts before processing the others and have their own service-level agreements (SLA) based on the alert level. If our consumers were keeping up with the messages being logged, it probably would not be a huge concern. However, this example shows that we could change the partition assignment with a custom class to make sure that our critical alerts are directed to the first partition (note that other alert levels could end up on partition 0 as well due to our logic, but that criticals will always end up there). The logic found mirrors an example of the `DefaultPartitioner` used in Kafka itself [44].

Listing 4.3 Custom Partitioner for alert levels

```
public class AlertLevelPartitioner implements Partitioner {

  public int partition(final String topic,   ❶
                       final Object objectKey,
                       final byte[] keyBytes,
                       final Object value,
                       final byte[] valueBytes,
                       final Cluster cluster) {

    final List<PartitionInfo> partitionMetaList =
        cluster.availablePartitionsForTopic(topic);
    final int criticalPartition = 0;

    final String key = ((Alert) objectKey).getAlertLevel();   ❷

    return key.contains("CRITICAL") ?
           criticalPartition : Math.abs(key.hashCode()) % partitionMetaList.size();     ❸
  }
  //...
}
```

❶ Custom class `AlertLevelPartitioner` needs to implement the partition method for its core logic.

❷ We are casting the value to a String to check the value.

❸ Critical alerts should end up on partition 0. Other alerts locations will be based on the key's hashcode.

Listing 4.3 shows an example of a custom partitioner. By implementing the `Partitioner` interface, we can use the `partition` method to send back the specific partition we would have our producer write to. In this case, we are looking at the value of the key to make sure that any `CRITICAL` events make it to a specific place - partition 0. In addition to the class itself being created, Listing 4.4 shows how the configuration key `partitioner.class` will need to be set for our producer to use the custom class we created. The configuration that powers Kafka is used to leverage our new class.

Listing 4.4 Custom Partitioner Class Config

```
Properties props = new Properties();
//...
props.put("partitioner.class",    ❶
          "org.kafkainaction.partitioner.AlertLevelPartitioner");
```

❶ Producer config needs to be updated 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. Custom code can be used to accomplish the specific logic of our business needs.

4.3 Generating data for our requirements

Let's try to use this information we gathered about how producers work on our own solutions.

Let's start with the audit checklist that we designed in Chapter 3 for use with Kafka in an e-bike factory. As noted in Chapter 3, we wanted to make sure that we did not lose any audit messages when operators completed 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. Listing 4.5 shows how we would start our producer config. We are trying to make sure that we are safe for message acknowledgment by setting `acks=all`, for example.

Listing 4.5 Audit Producer Config

```
Properties producerProperties = new Properties();     ❶
producerProperties.put( "bootstrap.servers",
                        "localhost:9092,localhost:9093,localhost:9094");
producerProperties.put("acks", "all");      ❷
producerProperties.put("retries", "3");     ❸
producerProperties.put("max.in.flight.requests.per.connection", "1");
```

❶ We start with creating properties for our configuration as before.

❷ We are using `acks=all` to get the strongest guarantee we can.

❸ We are letting the client do retries for us in cases of failure, so we don't have to implement our own failure logic.

Notice that we did not have to touch anything, but the configuration we send to the producer for the concern of message loss? This is a small but powerful feature that has a significant impact on if a message is going to arrive or not.

Since we do not have to correlate (group) any events together, we are not using a key for these messages. However, there is an essential piece of code that we do want to change in order to wait for the result before moving on to other codes. This means waiting for the response to complete synchronously and is shown in Listing 4.6. The `get` method is how we are waiting for the result to come back before moving on in the code.

Listing 4.6 Synchronous wait

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

❶ Unlike the other examples we have seen, we are doing a get to wait on the response from the send call.

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 over speed.

Another goal of our design for the factory was to capture the alert trend status of our stages and track their alerts over time. Since 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, since each stage id will be unique, it makes sense that we can use that id as a key. Listing 4.7 shows the `key.serializer` property being set as well as sending a `CRITICAL` alert.

Listing 4.7 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 producerProperties = new Properties();
    producerProperties.put("bootstrap.servers",
                           "localhost:9092,localhost:9093,localhost:9094");
    producerProperties.put("key.serializer",
                           "org.kafkainaction.serde.AlertKeySerde");     ❶
    producerProperties.put("value.serializer",
                           "org.apache.kafka.common.serialization.StringSerializer");

    try (Producer<Alert, String> producer = new KafkaProducer<>(producerProperties)) {
      Alert alert = new Alert(0, "Stage 0", "CRITICAL", "Stage 0 stopped");
      ProducerRecord<Alert, String> producerRecord =
          new ProducerRecord<>("kinaction_alerttrend", alert, alert.getAlertMessage());      ❷

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

❶ We need to tell our producer client how to serialize our custom Alert object into a key.

❷ Instead of null for the 2nd parameter, we put the actual object we wish to used to help populate the key.

If we use the default hashing of the key partition assigner, 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) will be 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 we will go with this for the time being.

Our last requirement was to have any alerts quickly processed to let operators know about any critical outages. We do want to group by the stage id in this case as well. One reason is that we can tell if that sensor was 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. Listing 4.8 shows the configuration of the producer to add the `partitioner.class` value to use our custom partitioner. The intention is for us to have the data available in a specific partition so those that process the data would have access to the critical alerts specifically and could go after other alerts (in other partitions) when those were handled.

Listing 4.8 Alert Producer

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

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

    try (Producer<Alert, String> producer = new KafkaProducer<>(producerProperties)) {
      Alert alert = new Alert(1, "Stage 1", "CRITICAL", "Stage 1 stopped");
      ProducerRecord<Alert, String>
          producerRecord = new ProducerRecord<>("kinaction_alert", alert,
                                                alert.getAlertMessage());     ❸

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

❶ We are reusing our custom Alert key serializer.

❷ We are using the property `partitioner.class` to set our custom partitioner class we created above.

❸ This is the first time we have used a callback to handle the completion or failure of an asynchronous send call.

One addition we see above is how we are adding a callback to run on completion. While we said that we are 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 our application-related errors. Listing 4.9 shows an example of implementing a `Callback` interface. The callback would log a message only if an error occurred.

Listing 4.9 Alert Callback

```
public class AlertCallback implements Callback {      ❶

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

  public void onCompletion(RecordMetadata metadata, Exception exception) {      ❷
    if (exception != null) {
      log.error("Error sending message:", exception);
    } else {
      log.info("Message sent: offset = {}, topic = {}, timestamp = {}",
               metadata.offset(), metadata.topic(), metadata.timestamp());
    }
  }
}
```

❶ We need to implement the Kafka interface `Callback`.

❷ The completion can have success or failure. Since we are only interested in the failures, we will print those out to our console.

While we will focus on small sample examples in most of our material, I think that it is helpful to look at how a producer is used in a real project as well. As mentioned earlier, Apache Flume can be used alongside Kafka to provide various data features. When Kafka is used as a `sink`, Flume places data into Kafka. While you might or might not be familiar with Flume, we are not interested in its feature-set but just seeing how it leverages Kafka producer code in a real situation. In the following examples, we will be referencing Flume version 1.8 located at github.com/apache/flume/tree/flume-1.8 if you want to reference more of the complete source code. Let's look at a snippet of configuration that would be used by a Flume agent in Listing 4.10.

Listing 4.10 Flume Sink Configuration

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

Topic, acks, bootstrap servers: Don't those configuration properties seem familiar from Listing 4.10? In our previous code, we declared the configuration inside our code as properties. However, this is showing an example of an application that chose to externalize the configuration values which is something we could do on our projects. The KafkaSink source code from Apache Flume, found at (raw.githubusercontent.com/apache/flume/flume-1.8/flume-ng-sinks/flume-ng-kafka-sink/src/main/java/org/apache/flume/sink/kafka/KafkaSink.java) shows an example of taking data and placing it inside Kafka with producer code. Listing 4.11 is

a small sample of a producer using a similar idea to take a configuration file like in Listing 4.10 and loading those values into a producer instance.

Listing 4.11 Reading Kafka Producer Configuration from a file

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

try (Producer<String, String> producer = new KafkaProducer<>(props)) {
      @SuppressWarnings("unused")
  ProducerRecord<String, String> producerRecord = new   ❶
    ProducerRecord<>(topic, null, "event");
}

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

  Properties props = new Properties();
  try (Stream<String>  lines = Files.lines(path)) {   ❷
      lines.forEachOrdered(line -> determineProperty(line, props));
  } catch (IOException e) {
    System.out.println("Error: " + e);
  }
  return props;
}

private static void determineProperty(String line, Properties props) {   ❸
  if (line.contains("bootstrap")) {
    props.put("bootstrap.servers", line.split("=")[1]);
  } else if (line.contains("acks")) {
      props.put("acks", line.split("=")[1]);
  } else if (line.contains("compression.type")) {
    props.put("compression.type", line.split("=")[1]);
  } else if (line.contains("topic")) {
    props.put("topic", line.split("=")[1]);
  }
  ...
}
```

❶ Our familiar `producer.send` with a callback.

❷ Reading an external file for configuration

❸ Parsing configuration properties and setting values.

While some code is omitted, the core Kafka producer pieces might be starting to look familiar in Listing 4.11. Setting configuration and the producer `send` method should all look like the code we have written 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` compares to the Kafka Sink callback class `SinkCallback` located in the source code at raw.githubusercontent.com/apache/flume/flume-1.8/flume-ng-sinks/flume-ng-kafka-sink/src/main/java/org/apache/flume/sink/kafka/KafkaSink.java . Both examples uses the `RecordMetadata` object to learn more information about successful calls. This information can help us learn more about where the producer message was written to 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 be successful. However, I think that if you want to know what is going on under the covers or need to do 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 1.0 and the Java producer client you are using is at 0.10 (using these versions as an example since they were the first to handle this mismatch!), the broker will handle this upgrade in the message version. However, because you can, it does not mean you should do it in all cases. Some of the things that make Kafka fast can be impacted by choices such as this. For example, instead of messages being read from the OS's page cache, the messages will have to be decompressed and converted to the newer message format. The trade-off is that the messages now have to be processed with the JVM heap being involved. Overhead is not zero, and garbage collection might become an issue for larger messages (or other various message combinations). To dig into more of the bidirectional version compatibility, take a peek at KIP-35 (cwiki.apache.org/confluence/display/KAFKA/KIP-35+-+Retrieving+protocol+version) and KIP-97 (cwiki.apache.org/confluence/display/KAFKA/KIP-97%3A+Improved+Kafka+Client+RPC+Compatibility+Policy). Just be aware of the cost of the version changes and watch for any impacts to your applications
when versions change.

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 we can start to pull this data back out so our other applications can consume it. Now that we have some ideas about the 'how' we get data into Kafka, 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 will be a vital part of this discovery, and as with producers, there are various configuration-driven behaviors that we can use to help us satisfy our different requirements for consumption.

4.4 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 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.

5 Consumers: unlocking data

This chapters covers

- Exploring the consumer read path and how it works.
- Leveraging consumer groups to coordinate reading data from topics.
- Learning about what offsets are and how to use them.
- Examining various configuration options that change consumer behavior.

In our previous chapter, we started working with writing data into our Kafka system. However, as you know, that is only one part of the story. Consumers are the way that we get data from Kafka and to other systems or applications that use that data to provide value. Since consumers are clients that exist outside of brokers, they can be written in various programming languages just like producer clients. Please take note that when we are looking at how things work below, I will try to lean towards the defaults of the Java consumer client.

After reading this chapter, we will be on our way to solving our previous business requirements by consuming data in a couple of different ways.

5.1 Introducing the Consumer

The consumer client is the program that subscribes to the topic or topics that it is interested in (you are not limited to only one topic at a time). As with producer clients, the actual consumer processes can run on separate machines and are not required to run on a specific server. In fact, most consumer clients in production settings will be on separate hosts. As long as the clients can connect to the Kafka brokers, they are good to read messages. Figure 5.1 reintroduces the broad context of Kafka and shows consumers running outside the brokers and working to get data outside of Kafka.

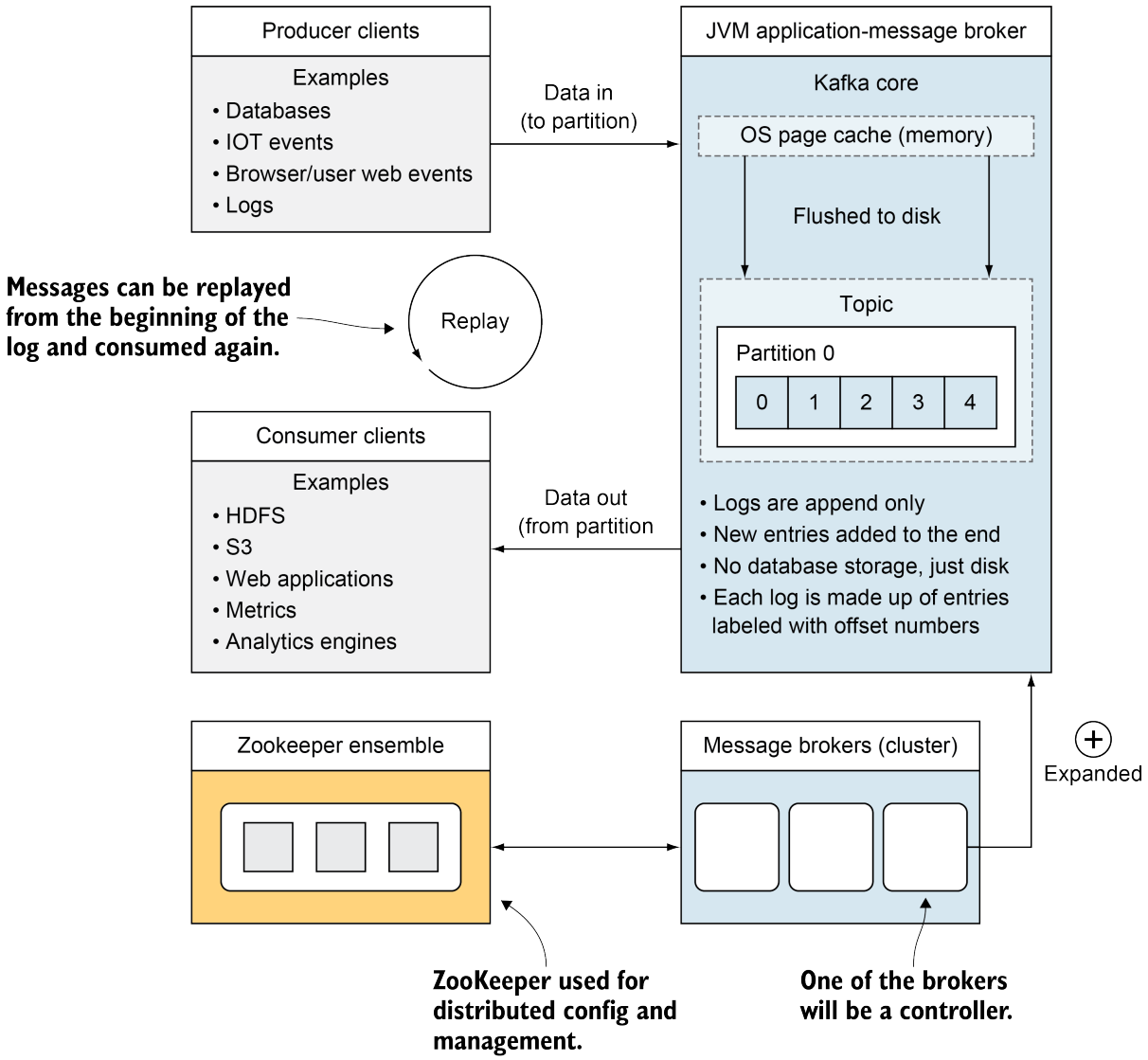

Figure 5.1 Kafka Consumer clients in Context

Why is it important to know that the consumer is subscribing to topics (pulling messages) and not being pushed to instead? The power of processing control shifts to the consumer in this situation. Figure 5.1 shows where consumer clients fit into the overall Kafka ecosystem. Clients are responsible for reading data from topics and making it available to application (like metrics dashboards or analytics engines) or storing it in other systems like HDFS or S3. Consumers themselves control the rate of consumption. If brokers drove the rate by pushing, one issue would be the rate of produced messages. If producers were hitting the brokers with data faster than the consumers could read the data, it would be overwhelming for the consumers to keep up. This pace could cause your consumer's resources to be used up and fail. With the consumer being in the driver's seat, if a failure occurred, and the consumer applications come back up, it will start pulling when it is up. No need to always have the consumers up and running to handle (or miss) notifications. While you can develop applications that are capable of handling this constant data flow or even a build-up of back-pressure due to volume, you need to know that you are not a listener to the brokers, you are the one pulling that data to your consumer. For those

readers that have used Kafka before, you might know that there are reasons that you probably will not want to have your consumers down for extended periods. When we discuss more details about topics, we will look at how data might be removed from Kafka due to size or time limits that users can define.

Let's take a look at reading from a topic with one consumer as we did in chapter 2. For this instance, we have an application that brings us back to the origin of Kafka talked about earlier in chapter 2 as well, LinkedIn's use-case of user activity events. Let's say that we have a specific formula that uses the time a user spends on the page as well as the number of interactions they had on the page that is sent as a value to a topic to project future click rates with a new promotion. Imagine we run the consumer and process all of the messages on the topic and are happy with our application of the formula (in this case multiplying by a magic number).

Listing 5.1 shows an example of looking at the records from the topic `webclicks` and printing out a value based on data from each event. Note that this is not a complete code listing but is meant to highlight specific consumer lines. Listing 5.1 has many similarities to the producer code that was written in chapter 4. Properties are used to determine the behavior of the consumer. Notice the class of deserializer that we will use to convert the bytes in Kafka back to the original format. This usage of deserializers for the key and values which is different than having serializers for producers. This will vary depending on the topic we are consuming. A consumer can subscribe to multiple topics, but in this instance we are only interested in the `webclicks` topic.

Also, an infinite loop is used to poll the topic partitions that our consumer is assigned in order to process messages. Usually this sort of loop can cause errors, especially for beginner programmers! Why an infinite loop? Part of the streaming mindset is events as a continuous stream and this is reflected in the logic. Notice that this example uses 500 for the value of the poll timeout which is in milliseconds. This timeout is for how long the call will block but will return immediately when records are ready for delivery. This value will be something that you will tune and adjust based on the needs of your applications.

Listing 5.1 Web Click Consumer

```
public class WebClickConsumer {

  final static Logger log = LoggerFactory.getLogger(WebClickConsumer.class);
  private volatile boolean keepConsuming = true;

  public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers",
              "localhost:9092,localhost:9093,,localhost:9094");
    props.put("group.id", "webconsumer");     (1)
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("key.deserializer",
              "org.apache.kafka.common.serialization.StringDeserializer");      (2)
    props.put("value.deserializer",
              "org.apache.kafka.common.serialization.StringDeserializer");

    WebClickConsumer webClickConsumer = new WebClickConsumer();
    webClickConsumer.consume(props);

    Runtime.getRuntime().addShutdownHook(new Thread(webClickConsumer::shutdown));
  }

  private void consume(Properties props) {
    try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)) {     (3)
      consumer.subscribe(Arrays.asList("webclicks"));     (4)

      while (keepConsuming) {     (5)
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));
        for (ConsumerRecord<String, String> record : records) {
          log.info("[Consumer Record] offset = {}, key = {}, value = {}",
                   record.offset(), record.key(), record.value());
          log.info("value = {}", Double.parseDouble(record.value()) * 1.543);
        }
      }
    }
  }

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

(1) `group.id` defined and will be discussed with consumer groups.

(2) Deserializers for the key and values defined.

(3) Properties are passed into the KafkaConsumer constructor.

(4) Subscribe to one topic: `webclicks`.

(5) The logic polls in an infinite loop for records that will come from the topic.

After generating a value for every message in the topic in Listing 5.1, we find out that our modeling formula wasn't correct! So what should we do now? Attempt to recalculate the data we have from our end-results (assuming the correction would be harder than in this example) and then apply a new formula? This is where we could use our knowledge of consumer behavior in Kafka to rather replay the messages we already processed. By having that raw data retained, we do not have to worry about trying to recreate that original data. Developer mistakes, application

logic mistakes, and even dependent application failures have a chance to be corrected because the data is not removed from our topics once it is consumed.

This also explains how time travel, in a way, is possible with Kafka. The consumer is controlling where to start reading data: from the beginning (or the earliest event available), from a location after the last read your client had previously committed, or just reading from the present and ignoring any earlier messages. The developer has a lot of control in determining how to consume the data they are interested in.

5.2 Important Consumer Options

You will notice a couple of properties that are related to the ones that were important for the producer clients as well. For one, ee always need to know the brokers we can attempt to connect to on the startup of the client. One minor gotcha is to make sure that you are using the deserializers for the key and values that match the serializers you produced the message with. For example, if you produce using a StringSerializer but try to consume using the AvroDeSerializer, you will likely get an exception that you will need to fix. The Kafka documentation has a helpful way of letting us know which options might have the most impact. Look for the IMPORTANCE label of High in the documentation listed at kafka.apache.org/documentation/#consumerconfigs. Table 5.1 lists some of the important configuration values to know as you start writing your own consumers [45]. One way to deal with all of the consumer configuration key names is to use the constants provided in the java class `ConsumerConfig` when developing (See kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/ConsumerConfig.html). However, in our examples, we will use the property names themselves for clarity.

Listing 5.1 already showed off four of these in action.

Table 5.1 Consumer Configuration

Key	Purpose
bootstrap.servers	One or many Kafka brokers to connect on startup
value.deserializer	The class that is being used for deserialization of the value
key.deserializer	The class that is being used for deserialization of the key
group.id	A name that will be used to join a consumer group
client.id	An id used for being able to identify a client (will use in chapter 10)
heartbeat.interval.ms	Interval for consumer's pings to the group coordinator

The values for these configurations will determine how our consumer interacts with the brokers as well as other consumers.

5.2.1 Understanding Tracking Offsets

One of the items that we have only talked about in passing so far is the concept of offsets. Offsets used as an index position in the log that the consumer sends to the broker to let it know what messages it wants to consume and from where. If you think back to our console consumer example, we used the flag `--from-beginning`. This sets the consumer's configuration parameter `auto.offset.reset` to `earliest` behind the scenes. With that configuration, you should see all the records on that topic for the partitions you are connected to-even if they were sent before you started up the console consumer. The top part of Figure 5.2 shows reading from the start of the log every time you run in this mode.

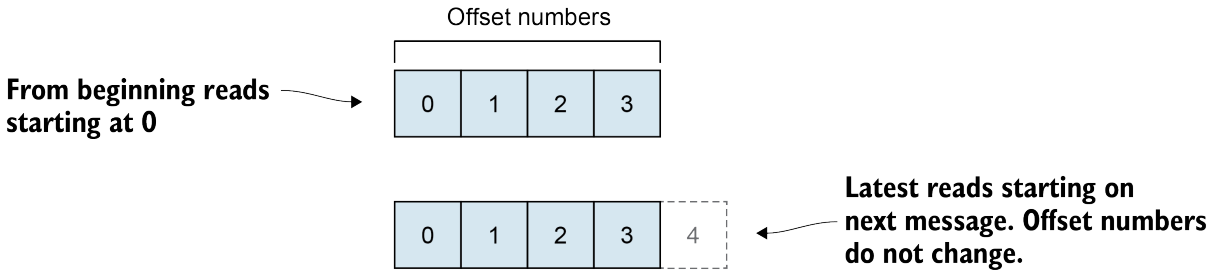

Figure 5.2 Kafka Offsets [46]

If you didn't add that option, the default is `latest`. Figure 5.2 shows this mode as well. In this case, you will not see any messages from the producer unless you send them after you start the consumer. It is saying disregard processing the messages that already are in the topic partition that your consumer is reading from, I only want to process what comes in after my consumer client starts polling the topic. You can think of it as an infinite array that has once had an index starting at 0. However, there are no updates allowed for an index. Any changes will have to be appended to the end of the log.

Note that offsets are always increasing for each partition. Once a topic partition has seen offset 0, even if that message is removed at a later point, the offset number will not be used again. Some of you might have run into the issue of numbers that keep increasing until they hit the upper bound of a data type. However, the offset value is a `long` data type (64-bit in Java), so the risk of overflowing (remember each partition has its own offset sequence) is considered negligible.

For a message written to a topic, what would the coordinates be to find the message? First, we would find the partition, within the topic, that it was written to, and then we would find the index-based offset. As Figure 5.3 shows, consumers usually read from the consumer's partition leader replica. Note, that this consumer leader replica could be different than any producer's leader replica due to changes in leadership over time, however, they are generally similar in concept.

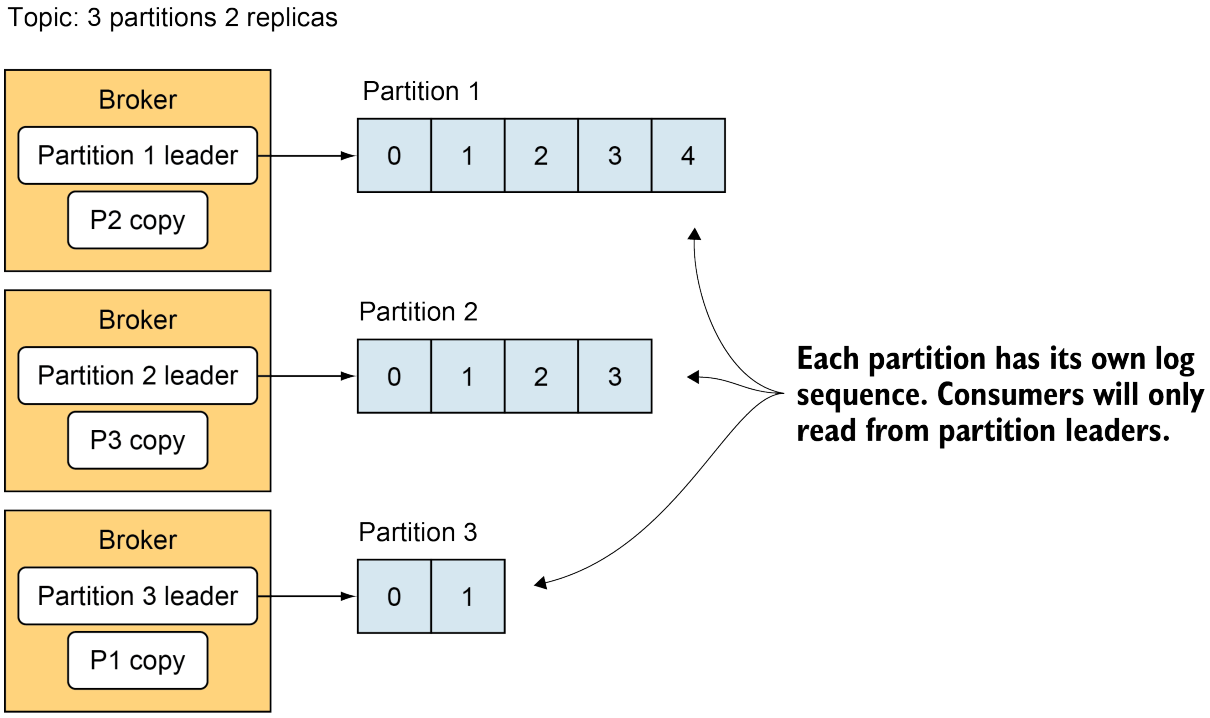

Figure 5.3 Partition Leaders

Follower replicas are used in the case of failure, after a new replica assumes leadership, but are not actively serving consumer fetch requests. Also, note that when we talk about partitions, it is okay to have the same offset number across partitions. The ability to tell messages apart needs to include the context of which partition we are talking about within a topic as well as the offset.

As a side note, if you do need to fetch from a follower replica due to an issue like network latency concerns - like having a cluster that stretches across data-centers, KIP-392 introduced this ability in version 2.4.0 [47]. As you are starting out with your first clusters, I would start with the default behavior and only reach for this feature as it becomes a real impact. If you do not have your cluster across different physical sites, you likely will not need this feature at the current time.

Partitions play a very important role in how we can process messages. While the topic is a logical name for what your consumers are interested in, they will read from the leader replicas of the partitions that they are assigned to. But how do consumers figure out what partition to connect to? And not just what partition, but where the leader exists for that partition? For each group of consumers, a specific broker will take on the role of being a group coordinator [48]. The consumer client will talk to this coordinator in order to get a partition assignment along with other details it needs in order to consume messages.

The number of partitions also comes into play when talking about consumption. Some consumers will not get any work to do with more consumers than partitions. An example would be four consumers and only three partitions. Why might you be okay with that? In some

instances, you might want to make sure that a similar rate of consumption will occur if a consumer dies unexpectedly. The group coordinator is in charge of not only assigning which consumers read which partitions at the beginning of the group startup, but also when consumers are added or fail and exit the group. To be clear, in the instance where there are more partitions than consumers, consumers will handle more than one partition if needed.

Figure 5.4 shows a generic view of how four consumers will read all of the data on the brokers where the subscribed topic has partition leader replicas spread evenly with one on each of the three brokers. Note that in this figure, the data is shown to be roughly the same size which might not always be the case. One consumer does not take traffic since each partition leader replica is handled by one consumer only.

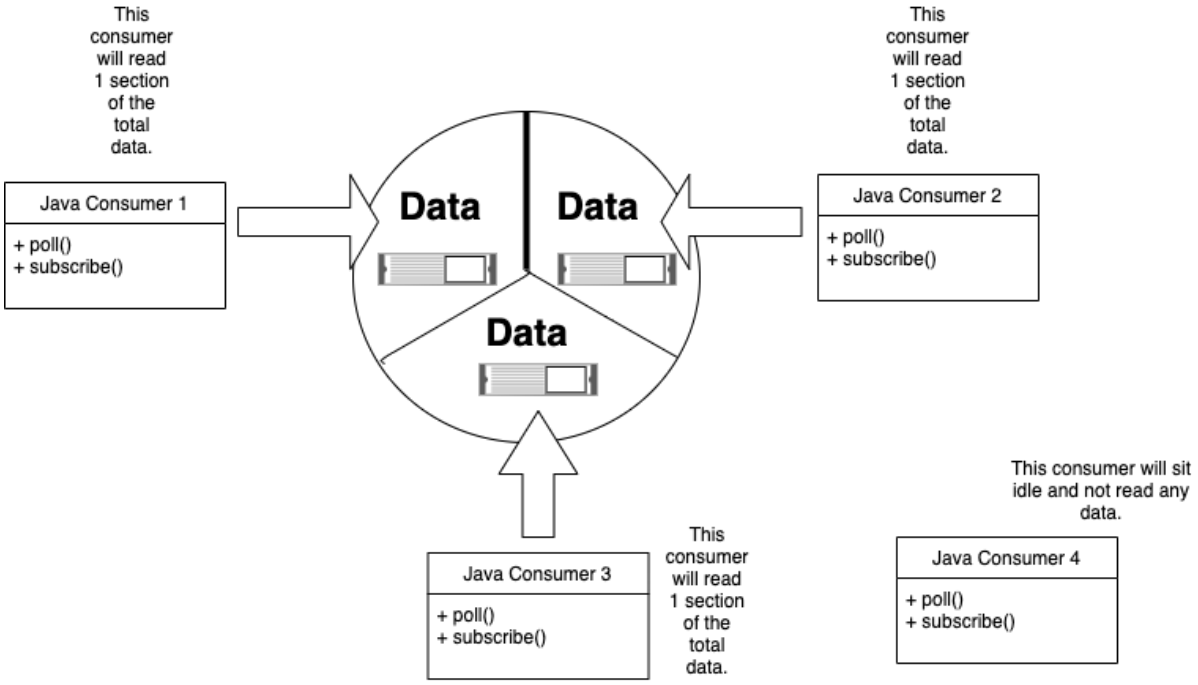

Figure 5.4 Idle Kafka Consumer

Since the number of partitions determines the amount of parallel consumers you can have, some might ask why you don't always choose a large number like having 500 partitions. This quest for higher throughput is not free [49]. This is why you will need to choose what best matches the shape of your data flow. One key consideration is that many partitions might increase end-to-end latency. If milliseconds count in your application, you might not be able to wait until a partition is replicated between brokers. This replication is key to having in-sync replicas and is done before a message is available to be delivered to a consumer. You would also need to make sure that you watch the memory usage of your consumers. If you do not have a 1-to-1 mapping of partitions to a consumers, each consumer's memory requirements may increase as it is assigned more partitions.

We have addressed the consumer running in an infinite loop, but what if you do want to stop a

consumer, what is the correct way? One common use case would be updating your consumer client code with new logic. For example, you might now want your consumers to interact with a different REST endpoint than they do today due to vendor changes. So far, you already saw the infinite loop of processing that you had just used Ctrl-C to end your processing or stopped the process on the terminal. However, the proper way includes calling a `close` method on the consumer. This proper shutdown method allows the group coordinator to receive notification about membership of the group being changed due to the client leaving rather than determining this from timeouts or errors from that consumer as it disappears. Listing 5.2 shows a consumer that would be run on a thread and controlled by a different class for shutdown. When the code in Listing 5.2 is started, the thread will run with a consumer instance. By calling the public method `shutdown`, a different class can flip the boolean and stop our consumer from polling for new records. The stopping variable will be our guard condition to decide whether or not to continue processing. Calling the `wakeup` method also causes a `WakeupException` to be thrown that leads to the finally block closing the consumer resource correctly [50].

Listing 5.2 Stopping a Consumer

```
public class StopConsumer implements Runnable {
    private final KafkaConsumer<String, String> consumer;
    private final AtomicBoolean stopping = new AtomicBoolean(false);
    ...

   public StopConsumer(KafkaConsumer<String, String> consumer) {
     this.consumer = consumer;
   }

    public void run() {
        try {
            consumer.subscribe(Arrays.asList("webclicks"));
            while (!stopping.get()) {      ❶
                ConsumerRecords<String, String> records =
                  consumer.poll(Duration.ofMillis(100));
                ...
            }
        } catch (WakeupException e) {      ❷
            if (!stopping.get()) throw e;
        } finally {
            consumer.close();      ❸
        }
    }

    public void shutdown() {      ❹
        stopping.set(true);
        consumer.wakeup();
    }
}
```

❶ Variable stopping determines whether to continue processing.

❷ The WakeupException will be triggered by the client shutdow hook.

❸ Close properly stops the client and informs the broker of shutdown.

❹ This method can be called from a different thread to stop the client in the proper way.

If your consumer process does not shutdown correctly, Kafka should still be informed of a down consumer by using a heartbeat. If no heartbeat occurs within a specific time, the coordinator will consider the client to be gone and reassign that client's partitions to the remaining active consumers [51].

If you run across older documentation for Kafka, you might notice consumer client configuration in regards to ZooKeeper. However, unless one is using a very old consumer client, Kafka does not have consumers rely directly on Zookeeper. While consumers used to store the offsets that they had consumed to at that point to Zookeeper, now the offsets are often stored inside a Kafka internal topic. As a side note, consumer clients do not have to store their offsets in either of these locations, but this will likely be the case. Just know that if you want to manage your own offset storage you can: either by storing it in a local file, an S3 bucket in AWS, or a database. One of the advantages of moving away from ZooKeeper storage was to reduce the clients' dependency on ZooKeeper.

5.3 Consumer Groups

Why is the concept of consumer groups so important? Probably the most important reason is that scaling is impacted by either adding more or less consumers to a group. Consumers that are not part of the same group do not share the same coordination of offset knowledge. Listing 5.3 shows an example of a group that would be named `testgroup`. If you instead had made up a new group.id (like a random GUID) it means that you are going to be starting as a new consumer with no stored offsets and with no other consumers in your group [52]. If you join an existing group (or one that had offsets stored already), your consumer will be able to share work with others or even be able to resume where it left off reading from any previous runs.

Listing 5.3 Consumer configuration for consumer group

```
Properties props = new Properties();
props.put("group.id", "testgroup");   ❶
```

❶ Your consumer behavior with other consumers determined by `group.id`.

It is often the case that you will have many consumers reading from the same topic. An important detail to decide on if you need a new group id is whether your consumers are working as part of one application or as separate logic flows. Why is this important? Think of two use cases for data that came from a human resources system. One team is wondering about the number of hires from specific states, and the other team is more interested in the data for the impact on travel budgets for interviews. So would anyone on the first team care about what the other team was doing or would either of them want to consume only a portion of the messages? Likely not! So how can we keep this separation? The answer is by making each application have its own specific `group.id`. Each consumer that uses the same `group.id` as another consumer

will be considered to be working together to consume the partitions and offsets of the topic as one logical application.

5.4 The Need for Offsets

While we are going through our usage patterns so far, we have not talked too much about how we keep track of what each client has read. Let's briefly talk about how some message brokers handle messages in other systems. In some systems, consumers do not track what they have read, they pull the message, and then it will not exist on a queue anymore after it has been acknowledged. This works well for a single message that needs to have exactly one application process it. Some systems will use topics in order to publish the message to all those that are subscribers. And often, future subscribers will have missed this message entirely since they were not actively part of that receiver list when the event happened.

Figure 5.5 shows non-Kafka message broker scenarios including how messages are often removed after consumption. It also shows a second pattern where a message might come from the original source and then be replicated to other queues. In systems where the message would be consumed and not available for more than one consumer, this is needed for separate applications to each get a copy.

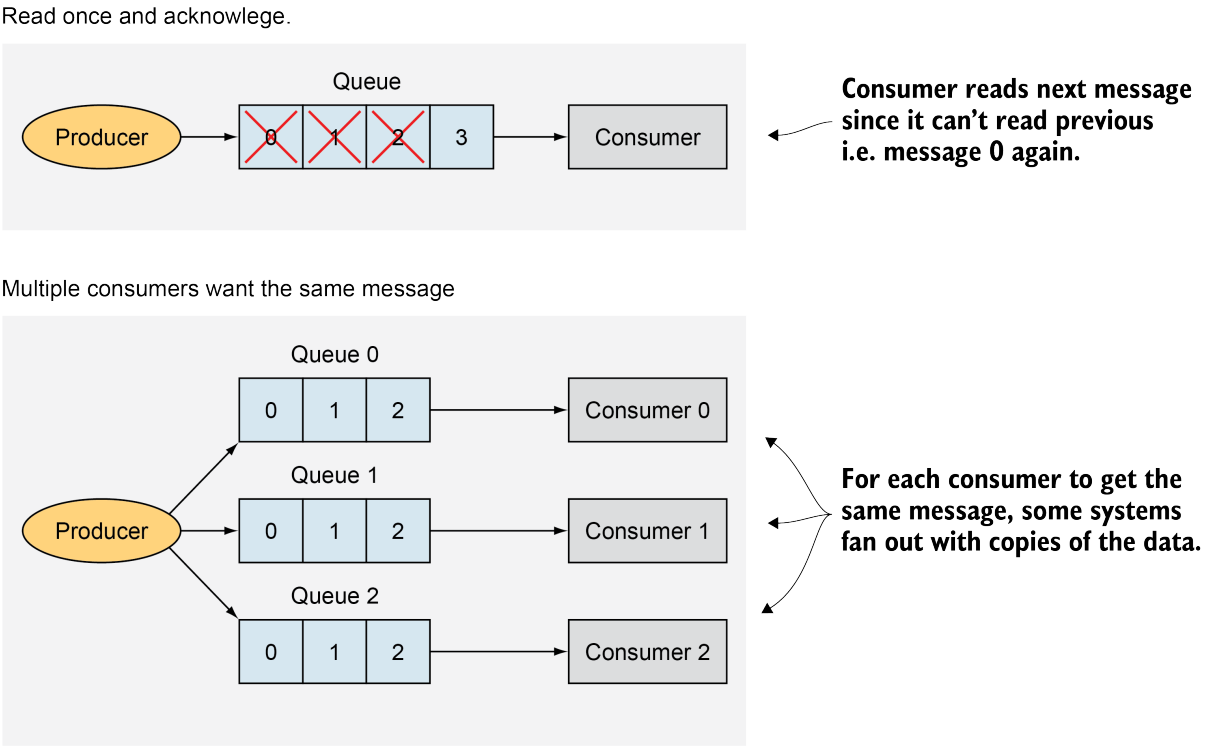

Figure 5.5 Other Broker Scenarios

You can imagine the copies grow as an event becomes a popular source of information. Rather than have entire copies of the queue (besides those for replication/failover), Kafka can serve multiple applications from the same partition leader replica.

Kafka, which we have mentioned in the first chapter, is not limited to only having one consumer. And even if a consuming application does not exist when the message was first created on the topic, as long as Kafka has retained that message in its log, then it can still process that data. Since messages are not removed from other consumers or delivered once, the consumer client would need a way to keep track of where it had read so far in the topic. In addition, since many applications could be reading the same topic, it is important that the offsets and partitions are tied-back and specific to a certain consumer group. The key coordinates to let your consumer clients work together is a unique blend of the following: group, topic, and partition number.

5.4.1 Group Coordinator

There is usually one broker that takes over the important job of working with offset commits for a specific consumer group. The offset commit can be thought of the last message consumed by a specific member of that consumer group. These duties are handled by the `GroupCoordinator` which helps with being an offset manager [53]. The group coordinator works with the consumer clients to keep track of where that specific group has read from inside the topic. These coordinates of a partition of a topic and group id make it specific to an offset value.

Looking at Figure 5.6, notice that we can use the offset commits as coordinates to figure out where to read from next. For example, a consumer that is part of the marketing consumer group and is assigned partition 0 would be ready to read offset 3 next.

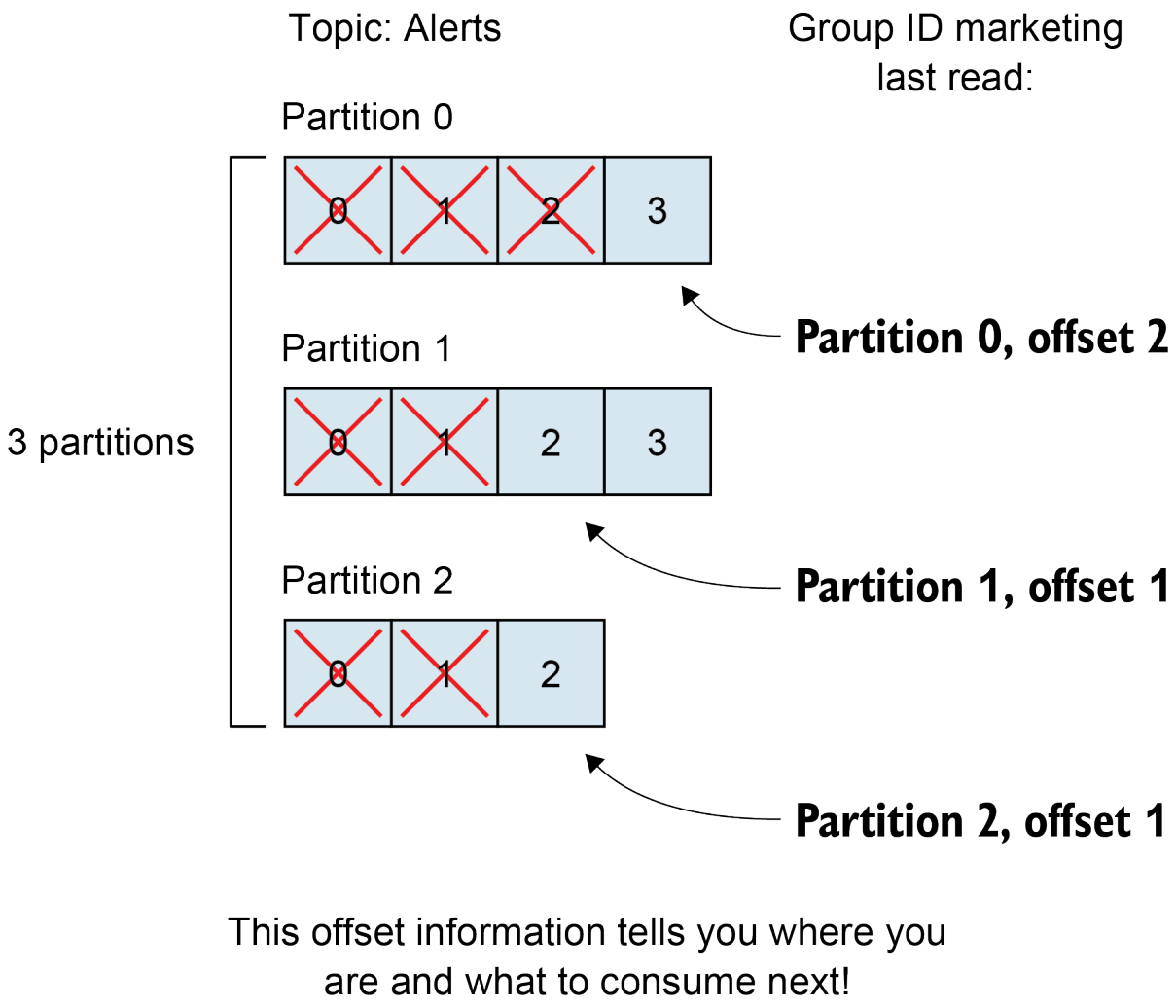

Figure 5.6 Coordinates

Figure 5.7 shows a scenario where the same partitions of interest exist on 3 separate brokers for 2 different consumer groups, marketing and ad sales. The consumers in each group will get their own copy of the data from the partitions on each broker. They do not work together unless they are part of the same group. Correct group membership is important for each group to have their metadata managed accurately.

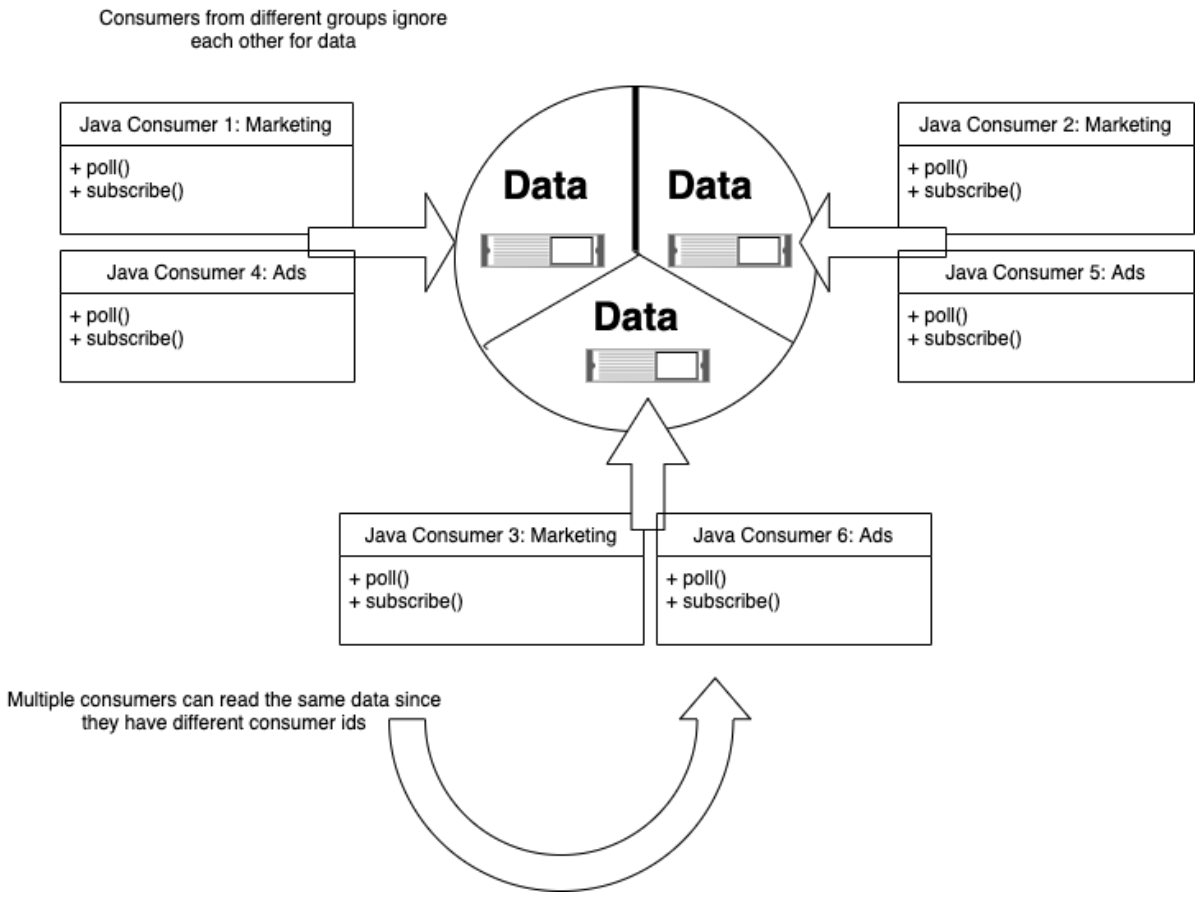

Figure 5.7 Consumers in Separate Groups [54]

As a general rule, only one consumer (per consumer group) can read one partition. In other words, while a partition might be read by many consumers, it will only be read by one consumer from each group at a point in time. Figure 5.8 highlights how one consumer will read two partitions leader replicas where the second consumer will only read data from a third partition leader [55]. A single partition replica will not be divided or shared between more than one consumer.

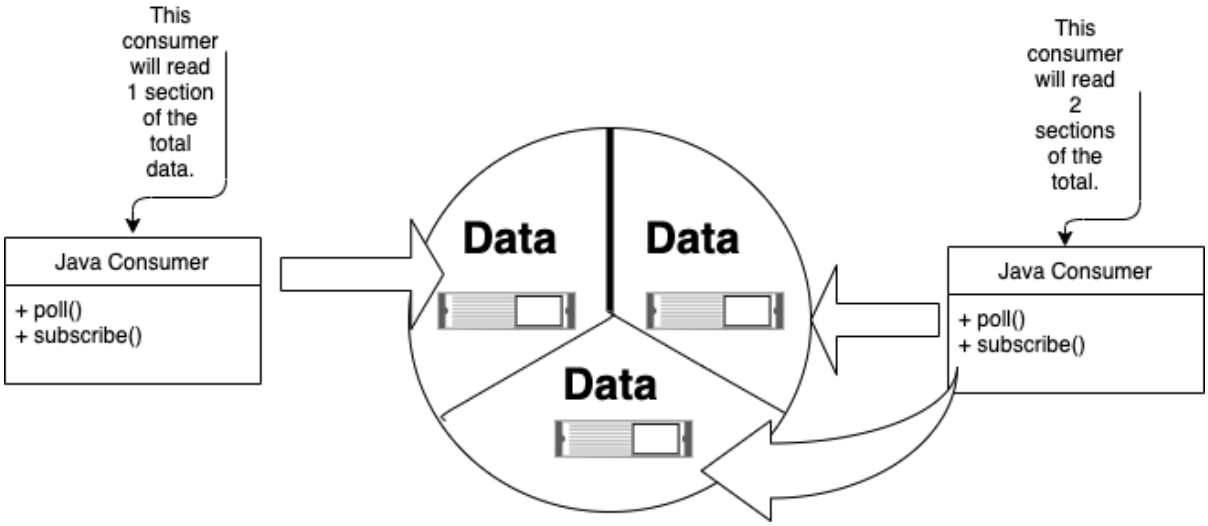

Figure 5.8 Kafka Consumers in a Group

One of the neat things about being a part of a consumer group is that when a consumer fails or leaves a group, the partitions that it was reading are re-assigned. An existing consumer will take the place of reading a partition that was once being read by the consumer that dropped out of the group.

One way a consumer can drop out of a group membership is by failing to send a heartbeat to the GroupCoordinator. This heartbeat is the way that the consumer communicates with the coordinator to let it know it is still replying in a timely fashion and working away. Failure by a consumer client to send these heartbeats can happen in a couple of ways [56]. One way for this is to stop the consumer client by either termination of the process or failure due to a fatal exception. If the client isn't running, it will not be sending messages back to the group coordinator. Another common issue is the time that your client code could be using to process the last batch of messages (long running code that takes seconds or minutes) before another poll loop.

Consumer groups also help when new consumer clients are being added [57]. Take an example of one consumer client reading from a topic made up of 2 partitions. Adding a second consumer client should result in each consumer processing one partition. Since one single partition will not be shared among consumer clients, one major detail to note is what happens when there are more consumer clients than the total number of partitions. If more consumers are present, then those consumers without assignments will be sitting idle. However, one valid reason to have idle consumers standing by is to make sure a consumer is ready to handle a partition in case of a different consumer failing.

5.4.2 Partition Assignment Strategy

The property `partition.assignment.strategy` is what determines which partitions are assigned to each consumer. Range is the default value but RoundRobin is also provided as is Sticky. Partition assignment will automatically change to account for differences in partitions or consumer updates [58].

- Range: This assigner uses a single topic to find the number of partitions (ordered by number) and then divides by the number of consumers. If the division is not even, then the first consumers (using alphabetical order) will get the remaining partitions. Note, if you have a topic and partition numbers that often cause partitions to be assigned unevenly, you might see some consumers taking a larger number of partitions over time. Make sure that you see a spread of partitions that your consumers can handle and consider switching the assignment strategy if some consumer clients used up all their resources while others are fine. Figure 5.9 shows how the first of 3 clients will grab 3 out of 8 total partitions and will end up with more partitions than the last client.
- Round Robin: This strategy is most easily shown by a consumer group that all subscribe to all the same topics. The partitions will be uniformly distributed in that the largest difference between assignments should be one partition. Figure 5.9 shows an example of 3 clients part of the same consumer group assigned in a round-robin fashion for one topic made of 8 partitions. Like dealing cards around a table, the first consumer gets the first partition, the second consumer the second, and so on until the partitions run out.

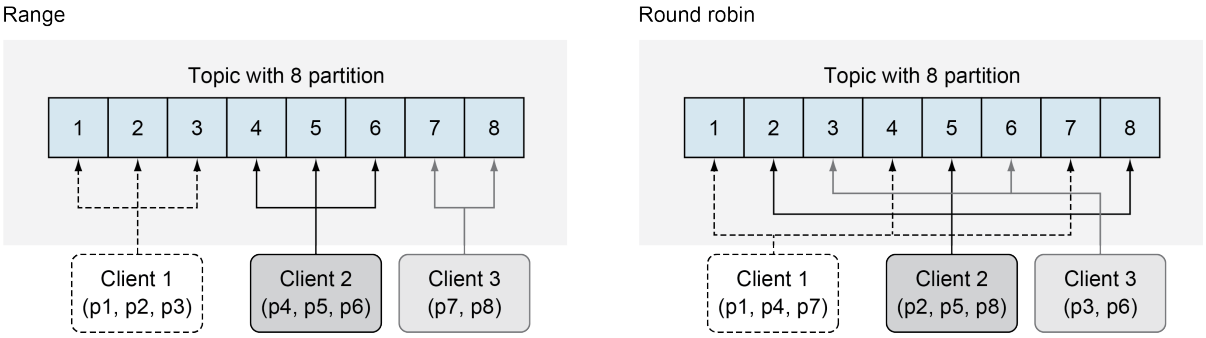

Figure 5.9 Partition Assignment

- Sticky: This is a strategy that was added in version 0.11.0 [59]. The driver is to not only try and distribute topic partitions as evenly as possible but also to have partitions stay with their existing consumer clients when possible. Allowing partition assignments to stay with the same consumers could help speed up a rebalance which could occur during any consumer group changes. Allowing certain consumers to stick to their existing partition assignment rather than make potentially every consumer change assignments is the goal.

You can also create your own strategy. `AbstractPartitionAssignor` is an abstract class that does some of the work to help implement your own logic. The exercise of implementing your assignor is left for the reader to explore if interested.

5.4.3 Manual Partition Assignment

If the assignment of partitions is important to your applications, then you will want to look at how to specify specific partitions in your client code. To implement manual assignment, you would call `assign` where you had previously called `subscribe`. Listing 5.4 shows an example of assigning a consumer directly to the first and second partitions of a topic. TopicPartition objects are used to help your code tell Kafka what specific partitions you are interested in for a topic. Passing the TopicPartition objects created to the `assign` method takes the place of allowing a consumer to be at the discretion of a group coordinator.

Listing 5.4 Manual Partition Assignment

```
//...
      consumer.assign(List.of(new TopicPartition(TOPIC_NAME, 1),      ❶
                              new TopicPartition(TOPIC_NAME, 2)));    ❷

      while (keepConsuming) {
        var records = consumer.poll(Duration.ofMillis(100));

        for (ConsumerRecord<String, String> record : records) {
          log.info("offset = {}, key = {}, value = {}", record.offset(), record.key(),
              record.value());
          log.info("value = {}", Integer.getInteger(record.value()) * 1.543);
        }
      }
//...
```

❶ The `assign` method receives a list fo topic and partition combinations.

❷ `TopicPartition` objects help define the topic and partition combination you are interested in.

Due to this assignment decision, it is important to note that each consumer acts on its own and does not coordinate with other consumers in a group. Sharing a group id with another consumer does not give this consumer any benefits like we have seen in examples when a consumer fails. In addition, if you add a partition that your consumer should know about, the consumer client will have to assign itself to see those messages from the new partition, your client won't learn automatically about and consume any new partitions. Make sure that you know the responsibilities of managing your own consumers when you reach for this feature.

5.5 Auto or Manual Commit of Offsets

One of the important things to think about is your needs for assuring that you have read messages from your topic. Is it okay to possibly miss a few? Or do you need each message confirmed as read? The real decision comes down to your requirements and what you are willing to trade-off. Are you okay with sacrificing some speed in order to have a safer method of seeing each message? These choices are discussed below.

One option is to use is `enable.auto.commit=true` which is the default for consumer clients [60].

Offsets are committed on your behalf on a recurring interval: `auto.commit.interval.ms`. One of the nicest parts of this option is that you do not make any other calls to commit the offsets that you have consumed. At-least-once delivery is related to automatic commits since Kafka brokers will resend messages again if they were never not automatically acknowledged due to a consumer client failure. But what sort of trouble could we get into? If we are processing messages that we got from our last poll, say in a separate application thread, the automatic commit offset could be marked as being read even if you do not actually have everything done with those specific offsets. What if we had a message fail in our processing that we would need to retry? With our next poll, we could be getting the next set of offsets after what was already committed as being consumed. It is possible and easy to lose messages that look like they have been consumed despite not being processed by your consumer logic.

When looking at what you commit, notice that timing might not be perfect. If you do not call a commit method on a consumer with metadata specifically noting your specific offset to commit, you might have some undefined behavior based on the timing of polls, expired timers, or even your own threading logic. If you need to be sure to commit a record at a specific time as you process it, or a specific offset in general, you should make sure that you send the offset metadata into the commit method.

Let's explore this topic more and talk about using manual commits enabled by: `enable.auto.commit=false`. This method can be used to exercise the most control over when your application has actually consumed a message and committed it. At-least-once delivery guarantees can be achieved with this pattern. Let's talk about an example in which a message causes a file to be created in Hadoop: like in a specific HDFS location. As you get a message, let's stay that you poll a message at offset 100. During processing, the consumer stops because of an error. Since the code never actually committed offset 100, the next time a consumer of that same group starts reading from that partition, it will get the message at offset 100 again. So by delivering the message twice, the client was able to complete the task without missing the message. On the flip side, you did get the message twice! If for some reason your processing actually worked and you achieved a successful write, your code will have to handle the fact that you might have duplicates.

Let's look at some of the code that we would use to manually commit our offsets. As we did with a producer when we sent a message, we can also commit offsets in a synchronous manner or asynchronous. Listing 5.5 shows a synchronous commit. Looking at the example Listing 5.5 for commitSync, it is important to note that the commit is taking place in a manner that will block until a success or failure occurs.

Listing 5.5 Synchronous Commit

```
while (keepConsuming) {
        var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          log.info("offset = {}, key = {}, value = {}",
                   record.offset(), record.key(), record.value());
        }
        consumer.commitSync();      ❶
}
```

❶ The commitSync method is a blocking waiting for a success or fail

Also, note that the commit is taking place outside of the loop that goes through all of the returned records. Since no parameters are passed in, the call to commitSync will not commit each record one by one, as the looping logic processed each record. Rather, the entire set of record offsets returned from the last `poll` will be committed. Listing 5.6 shows an asynchronous commit that takes a callback as an argument. CommitAsync is the path to manually commit without blocking your next iteration. One of the options that you can use in this method call is the ability to send in a callback.

Listing 5.6 Asynchronous Commit

```
while (keepConsuming) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        log.info("[Consumer Record] offset = {}, key = {}, value = {}",
          record.offset(), record.key(), record.value());    }
    consumer.commitAsync(callback);     ❶
}
```

❶ The `commitAsync` method is a non-blocking call that allows the logic to continue.

If you think back to chapter 4, this is similar to how we used asynchronous sends with a callback as well for acknowledgments. To implement your own callback you need to use the interface: `OffsetCommitCallback`. You can define an onComplete method definition to handle exceptions or successes as you wish. Listing 5.7 shows how to create an asynchronous commit with a callback by implementing the `OffsetCommitCallback` interface. This instance allows us to have log messages to determine our success or failure even though our code is not waiting for a response before moving on to the next instruction.

Listing 5.7 Asynchronous Commit with Callback

```
public static void commitOffset(long offset,
                                int partition,
                                String topic,
                                KafkaConsumer<String, String> consumer) {
   OffsetAndMetadata offsetMeta = new OffsetAndMetadata(offset + 1, "");

   Map<TopicPartition, OffsetAndMetadata> offsetMap = new HashMap<>();
   offsetMap.put(new TopicPartition(topic, partition), offsetMeta);

   consumer.commitAsync(offsetMap, (map, e) -> {     ❶
     if (e != null) {    ❷
       for (TopicPartition key : map.keySet()) {
         log.info("Commit failed: topic {}, partition {}, offset {}", key.topic(),
             key.partition(), map.get(key).offset());
       }
     } else {
       for (TopicPartition key : map.keySet()) {
         log.info("OK: topic {}, partition {}, offset {}", key.topic(), key.partition(),
             map.get(key).offset());
       }
     }
   });
 }
```

❶ A lambda that creates OffsetCommitCallback instance.

❷ Implementing `onComplete` to handle commit failure or success in an asynchronous manner.

Why would you want to choose synchronous or asynchronous commit patterns? For one, you need to keep in mind that your latency will be higher if you are waiting for a blocking call. This time factor might be worth it if your requirements include needs for data consistency. What does consistency mean in this instance? It really is a question of how you need to process your data. Do you need to wait to process your messages in order before committing the offsets that have been processed even in the case of failures? If a failure occurred before a commit, does your logic handle reprocessing messages that your application might have seen before but did not mark as committed to Kafka? These decisions help determine the amount of control you need to exercise when informing Kafka which messages your logic considers as processed.

5.6 Reading From a Compacted Topic

Consumers should be aware if they are reading from a compacted topic. Chapter 7 will go further into how these topics work to, in effect, update records that have the same key value. In short, the partition log is compacted by Kafka on a background process and records with the same key can be removed except for the last one. If you do not need a history of messages, but rather just the last value, you might wonder how this concept works with an immutable log that only adds records to the end. At this point, the biggest gotcha for consumers that might cause an error is that when reading records, consumers might still get multiple entries for a single key! How is this possible? Since compaction runs on the log files that are on-disk, compaction might not see every message that exists in memory during cleanup. Clients will need to be able to handle this case where there is not only one value per key. Please have logic in place to handle duplicate keys and, if needed, ignore all but the last value.

To pique your interest in compacted topics, note that Kafka uses its own compacted internal topic that relates directly to your consumer offsets themselves. The topic that holds the consumer clients' offsets is titled: __consumer_offsets. Compaction makes sense here since for a specific combination of a consumer group, partition, and topic, only the latest value is needed since it will have the latest offset consumed.

5.7 Reading for a Specific Offset

While there is no lookup of a message by a key option in Kafka, it is possible to seek to a specific offset. Thinking about our log of messages being an ever-increasing array with each one message having an index, we have a couple of options including starting from the beginning, going to the end, starting at a given offset, or finding offsets based on times.

5.7.1 Start at the beginning

One issue that you might run into is that you want to read from the beginning of a topic even if you have already done so. Reasons could include logic errors and a desire to replay the entire log or a failure in our data pipeline flow after starting with Kafka. The important configuration to note for this behavior is to set `auto.offset.reset` to earliest [61]. Another technique that can be used is to run the same logic but use a different group id. In effect, this means that the commit offset topics that Kafka uses internally would not be able to find an offset value and will be able to start at the first one it finds since it would not have any data on the new consumer group. Listing 5.8 is an example of setting the property `auto.offset.reset` to 'earliest'. Setting a group id to a random UUID would also help you achieve starting with no consumer group offset history.

Listing 5.8 Earliest Offset

```
Properties props = new Properties();
props.put("group.id", UUID.randomUUID().toString());      ❶
props.put("auto.offset.reset", "earliest");     ❷
```

❶ Group id that Kafka would not have a stored offset recorded for

❷ Earliest offset we still have retained in our logs will be used

Also, the Java client consumer has the following method available: `seekToBeginning`. This is yet another way to achieve the same result as the code in Listing 5.8.

5.7.2 Going to the end

Sometimes you just want to start your logic from when the consumers start-up and forget about past messages. Maybe the data is already too late to have business value in your topic. Listing 5.9 shows the properties you would set to get this behavior of starting with the latest offset. Using a UUID isn't necessary except for testing when you want to make sure that you don't find a consumer offset from before and will instead default to the latest offset Kafka has for your subscriptions.

Listing 5.9 Latest Offset

```
Properties props = new Properties();
props.put("group.id", UUID.randomUUID().toString());      ❶
props.put("auto.offset.reset", "latest");     ❷
```

❶ Create a group id that Kafka would not have a stored offset for

❷ Latest record offset will be used

The Java client consumer also has the following method available: `seekToEnd`. This is another way to achieve the same result as the configuration changes in Listing 5.9.

5.7.3 Seek to an Offset

Kafka also gives you the ability to find any offset and go directly to it. The seek action sets a specific offset that will be used by the next poll on the consumer. The topic and partition are needed as the coordinates for giving your consumer the information it needs to set that offset. Listing 5.10 shows an example of seeking to the offset number 5000 for a single topic and partition combination. A TopicPartition object is used to provide the topic name and partition of interest which is partition one in this example. The `seek` method takes the topic information to set up the provided offset as the starting place of the next poll of the consumer.

Listing 5.10 Seek to a Specific Offset

```
long offset = 5000L;
TopicPartition partitionOne = new TopicPartition(topicName, 1);  ❶
consumer.seek(topicPartition, offset);      ❷
```

❶ A TopicPartition object provides the topic name and partition of interest.

❷ The `seek` method takes the topic information and offset.

Let's try to think of a use-case in which this might be used. Let's say that an analytics job has processed 4999 records before it failed on the next 10,000 records. Instead of starting at offset 0, the processing could seek to this specific offset, 5000 in this case, and continue onward without having to reprocess data unnecessarily.

5.7.4 Offsets For Times

One of the trickier offset search methods is `offsetsForTimes`. This method allows you to send a map of topics and partitions as well as a timestamp for each in order to get a map back of the offset and timestamp for those topics and partitions given. This can be used in situations where a logical offset is not known, but a timestamp is known. For example, if you have an exception that was logged related to an event, you might be able to use a consumer to determine the data that was processed around your specific timestamp.

As Listing 5.11 shows, we have the ability to retrieve the offset and timestamps per a topic/partition when we map each to a timestamp. After we get our map of metadata returned from the `offsetsForTimes` call, we then can seek directly to the offset we are interested in by seeking to the offset returned for each respective key.

Listing 5.11 Seek Offset by Timestamps

```
...
Map<TopicPartition, OffsetAndTimestamp> offsetMap =
consumer.offsetsForTimes(timeStampMapper);    ❶
...
// We need to use the map we get
consumer.seek(partitionOne, offsetMap.get(partitionOne).offset());    ❷
```

❶ Timestamps used to find the first offset greater or equal to that time

❷ Seeking to the first offset provided in the offsetMap

One thing to be aware of is that the offset returned is the first message with a timestamp that makes your criteria. However, due to the producer resending messages on failures or variations in when timestamps are added (by consumers, perhaps), this timestamp is not an official order. Offsets are ordered in the log as they have always been by increasing offset and timestamps do not have this absolute ordering.

5.8 Reading Concerns

One thing that we have talked about is that the consumer is in the driver's seat about when data comes to your application. However, let's picture a common scenario. We had a minor outage but bought back all of our consumers with new code changes. Due to the amount of messages we missed, we suddenly see our first poll bring back a huge amount of data. Is there a way that we could have controlled this being dumped into our consumer application memory? `fetch.max.bytes` is one property that will help us avoid too many surprises. The value is meant to set the max amount of bytes per request. However, the consumer might make parallel fetch calls at the same time from different partitions and impact the true total on your client.

5.9 Retrieving data for our factory requirements

Let's try to use this information we gathered about how consumers work to see if we can start working on our own solutions designed in chapter 3 for use with Kafka in our e-bike factory: but from the consumer client perspective in this chapter. As noted in Chapter 3, we wanted to make sure that we did not lose any audit messages when operators completed commands against the sensors. One requirement was that there was no need to correlate (or group together) any events across the individual events. This means that there are no concerns on the order or need to read from specific partitions, any consumer reading any partition should be good. Also, another one of the requirements was to not lose any messages. One safe way to make sure that our logic is executed for each audit event is to manually commit the offset per record after it is consumed. To control the commit as part of the code: `enable.auto.commit` will be set to false.

Listing 5.12 shows an example of leveraging a synchronous commit after each record is processed for the Audit feature. Details of the next offset to consume in relation to the topic and partition of the offset that was just consumed is sent as a part of each loop through the records. Note that it might seem odd to add 1 to the current offset, but this offset sent to your broker is supposed to be the offset of the next message that will be consumed. The method `commitSync` is called and passed the offSet map containing the offset of the record that was just processed.

Listing 5.12 Audit Consumer Logic

```
...
props.put("enable.auto.commit", "false");     ❶

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)) {

      consumer.subscribe(List.of("kinaction_audit"));

      while (keepConsuming) {
        var records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
          log.info("offset = {}, key = {}, value = {}",
                   record.offset(), record.key(), record.value());

          OffsetAndMetadata offsetMeta = new OffsetAndMetadata(record.offset() + 1, "");    ❷

          Map<TopicPartition, OffsetAndMetadata> offsetMap = new HashMap<>();
          offsetMap.put(new TopicPartition("audit", record.partition()), offsetMeta);     ❸

          consumer.commitSync(offsetMap);     ❹
        }
      }
    }
...
```

❶ Auto commit set to false.

❷ Adding 1 to the current offset determines the next offset that will be read.

❸ The offset map allows for a topic and partition key to be related to a specific offset.

❹ `commitSync` called to commit the offsets processed.

Another goal of the design for our e-bike factory was to capture our stage alert status and track the stage alert trend over time. Even though we know our records have a key that is the stage id, there is no need to consume a group at a time or worry about the order. Listing 5.13 shows the `key.deserializer` property being set so the consumer knows how to deal with the binary data that was stored in Kafka when we produced the message. In this example, `AlertKeySerde` is used for the key to deserialize. Since message loss isn't a huge concern, allowing auto commit of messages is good enough in this situation.

Listing 5.13 Alert Trending Consumer

```
...
props.put("enable.auto.commit", "true");      ❶
props.put("key.deserializer",
  "org.kafkainaction.serde.AlertKeySerde");     ❷
props.put("value.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<Alert, String> consumer = new KafkaConsumer<Alert, String>(props);
consumer.subscribe(Arrays.asList("kinaction_alerttrend"));

while (true) {
    ConsumerRecords<Alert, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<Alert, String> record : records) {
        log.info("offset = {}, key = {}, value = {}",
          record.offset(), record.key().getStageId(), record.value());
    }
}
...
```

❶ Auto commit is used since lost messages are not an issue

❷ AlertKeySerde key deserializer

The last requirement was to have any alerts quickly processed to let operators know about any critical issues. Since the producer used a custom `Partitioner`, we will assign a consumer directly to that same partition to alert on critical issues. Since a delay in case of other alerts is not desirable, the commit will be for each offset in an asynchronous manner.

Listing 5.14 shows the consumer client logic focused on critical alerts assigning themselves to the specific topic and partition that would have been used when producing alerts when the custom partitioner class `AlertLevelPartitioner` was used. In this case, it was partition number 0. The consumer assigns itself the partition rather than subscribing to the topic. For each record that comes back from the consumer poll, an asynchronous commit will be used with a callback. A commit of the next offset to consume will be sent to the broker and should not block the consumer from processing the next record. The asynchronous commit method is sent the topic and partition mapping to offset information as well as a callback to run. This ability to handle any issues with a callback at a later time helps make sure that other records can be processed quickly.

Listing 5.14 Alert Consumer

```
props.put("enable.auto.commit", "false");

KafkaConsumer<Alert, String> consumer = new KafkaConsumer<Alert, String>(props);
TopicPartition partitionZero = new TopicPartition(kinaction_alert", 0);     ❶
consumer.assign(Arrays.asList(partitionZero));     ❷

while (true) {
    ConsumerRecords<Alert, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<Alert, String> record : records) {
        log.info("offset = {}, key = {}, value = {}",
        record.offset(), record.key().getStageId(), record.value());
        commitOffset(record.offset(), record.partition(), topicName, consumer);     ❸
    }
}

...
public static void commitOffset(long offset,int part, String topic,
  KafkaConsumer<Alert, String> consumer) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(offset + 1, "");

    Map<TopicPartition, OffsetAndMetadata> offsetMap =
      new HashMap<TopicPartition, OffsetAndMetadata>();
    offsetMap.put(new TopicPartition(topic, part), offsetMeta);

    OffsetCommitCallback callback = new OffsetCommitCallback() {
    ...
    };
    consumer.commitAsync(offsetMap, callback);     ❹
}
```

❶ The TopicPartition used for critical messages.

❷ Consume assigns itself the partition rather than subscribing to the topic.

❸ Each record is committed in an asynchronous manner.

❹ The asynchronous commit leverages the offsetMap and callback arguments.

Of course, your exact decisions could be slightly different, but the above options seem to satisfy our core design requirements from chapter 3.

NOTE **Brokers use consumers too!**

If you think about how many replicas of a partition you have, you will notice that the data of a partition's leader replica likely has to be copied to other brokers that host follower replicas. For this replication, Kafka brokers use fetch requests like the consumer client we have used so far in this chapter! It is interesting to see how understanding the usage of parts of Kafka explains some of the inner workings of Kafka itself. We will get into more details about brokers copying data in Chapter 6.

Overall, the consumer can be a very complex piece of our interactions with Kafka. Some options can be done with property configuration alone, but if not, you can leverage your knowledge of topics, partitions, and offsets to navigate your way to the data you need.

5.10 Summary

- Consumer clients provide developers a way to get data out of Kafka. As with producer clients, consumer clients have a large number of configuration options available for clients to set rather than custom coding being needed.
- Consumer groups allow more than one client to work as a group to process records. By grouping together, clients can process data in parallel. Kafka coordinates changes on the groups behalf such as clients joining or dropping from the group dynamically.
- Offsets represent the position of a record in regards to a commit log that exists on a broker. By using offsets, consumers can control where they want to start reading data. This offset can even be a previous offset that it has already seen (ability to replay records again).
- Consumers can read data in a synchronous or an asynchronous manner. If asynchronous methods are used, the consumer can leverage code in callbacks to run logic once data is received.

6 Brokers

This chapters covers

- Exploring the role of brokers, and their duties.
- Evaluating options for certain broker configuration values.
- Explaining replicas and how they work to stay up-to-date.
- Assessing what JMX is and what it can provide.

So far in our discussions, we have been dealing with Kafka from the view of an application developer interacting from external applications and processes. However, Kafka is a distributed system that needs attention to run in its own right. Let's look at the parts that make the Kafka brokers work!

6.1 Introducing the Broker

While we have focused on the client-side of Kafka so far, our focus will now shift to another powerful component of the ecosystem: brokers. Being a broker in Kafka means being a part of a cluster of machines - brokers work together with other brokers to form the core of the system.

As we start to discover Kafka, those who are familiar with Big Data concepts or worked with Hadoop before, might hear familiar terminologies such as rack awareness (knowing which physical server rack a machine is hosted on) and partitions. For example, Kafka has a rack awareness feature that will make replicas for a partition exist physically on separate racks [62]. This is important since if all of the servers that make up our cluster are on the same rack, and the entire rack is offline due to a power or network issue, it would be the same as if we had lost our entire data center. Using these familiar data terms should make us feel at home as we draw new parallels between what we've worked with before and Kafka.

When seting up our own Kafka cluster, it is important to know that we have another cluster to be aware of: ZooKeeper.

6.2 Role of ZooKeeper

At this point, ZooKeeper is a key part of how the brokers work and is a requirement for Kafka to run. Since it needs to be running and exist before the brokers do, we will start our dicussion here.

> **IMPORTANT** **ZooKeeper Removal**
>
> **As noted in chapter 2, to simplify the requirements of running Kafka, the Apache Kafka community proposed the replacement of ZooKeeper with its own managed quorum [63]. Since this work was not yet complete at the time of publication, with an early access release version arriving in 2.8.0, ZooKeeper is still discussed in this work.**

Since ZooKeeper needs to have a quorum (a minimum number) in order to elect leaders and reach a decision, this cluster is indeed important for our brokers [64]. ZooKeeper itself holds information on the brokers that are in the cluster [65]. This list should be a current membership list and can change over time. How do brokers join this list? When a broker starts, it will have a unique id that will be listed on ZooKeeper [66]. This id can either be created with the configuration `broker.id` or `broker.id.generation.enable=true`. Nodes can leave the group as well. ZooKeeper also helps Kafka make decisions - like electing a new controller. The controller is a single broker in a cluster that takes on additional responsibilities and will be covered in more depth later in this chapter. Any broker could become the controller, but only one, and it's important that all of the brokers know who the controller is. ZooKeeper helps the brokers by coordinating this assignment and notification.

With all of this interaction with the brokers, it is very important that we have ZooKeeper running when starting our brokers. The health of the ZooKeeper cluster will impact the health of our Kafka brokers. For instance, if our ZooKeeper instances are blown away, topic metadata and configuration would be lost. Also, the assignments of replicas to broker ids would be lost, and would be difficult to recover.

Usually, we won't need to expose the details of our Zookeeper cluster (IP addresses and ports) to our producer and consumer applications. Certain frameworks versions you use may also still provide a means of connecting our client application with our Zookeeper cluster due to legacy needs. One example of this was version 3.1.X of Spring Cloud Stream (docs.spring.io/autorepo/docs/spring-cloud-stream-binder-kafka-docs/3.1.x/reference/html/) which still allowed us to set the `zkNodes` property. The value defaulted to `localhost` and should usually be left alone in most cases, so as to avoid a ZooKeeper dependency. This property is marked as deprecated but you never know if you will encounter older code for maintenance and

you want to keep your eyes out for it. Why is this not needed currently and in the future? Besides the fact that Kafka will not always require ZooKeeper, it is also important for us to avoid unnecessary external dependencies in our applications. In addition, it gives us fewer ports to expose if we are working with firewalls for Kafka and our client to communicate directly.

Using the Kafka tool `zookeeper-shell.sh` which is located in the `bin` folder of our Kafka installation, we can connect to a ZooKeeper host in our cluster and look at how data is stored. Looking at the path `/brokers/topics` we will see a list of the topics that we have created. At this point, we should have the `kinaction_helloworld` topic in the list. We might also see the topic `__consumer_offsets` which is created once we use a consumer client with our cluster. That topic is one that we did not need to create, it is a private topic used internally by Kafka. This topic stores the committed offsets for each topic and partition by consumer group id. Knowing about these internal topics helps to avoid surprises when we see topics in our cluster that we didn't create.

Listing 6.1 Listing our topics

```
bin/zookeeper-shell.sh localhost:2181   ❶
ls /brokers/topics   ❷

# OR
bin/kafka-topics.sh --list --bootstrap-server localhost:9092   ❸
```

❶ Connect to our local ZooKeeper instance.

❷ List all the topics with `ls` command.

❸ Using the kafka-topics command connect to ZooKeeper as well to provide the same information

The output should include the topic that we created in chapter 2: `[kinaction_helloworld]`

Note that we can also use a different Kafka tool, `kafka-topics.sh` to see the list of topics as well and get the same results! Both commands in Listing 6.1 connect to ZooKeeper for their data but do so with a different command interface!

It is important when reading anything about ZooKeeper and Kafka to note that KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum is currently in progress at the time of publication [67]. Even when ZooKeeper is no longer helping to power Kafka, we might need to work with clusters that have not migrated yet, and we will likely see Zookeeper in documentation and reference material for quite a while. Overall, being aware of the tasks that Kafka used to rely on ZooKeeper to perform, and the shift to handling those inside a Kafka cluster with internal metadata nodes will provide insight into the moving pieces of the entire system.

6.3 The Job of a Kafka broker

Being a Kafka broker means being able to coordinate with the other brokers as well as talking to ZooKeeper. In testing, or working with proof-of-concept clusters, we might have only one broker node. However, in production, we will almost always have multiple brokers. Figure 6.1 shows how brokers exist in a cluster and how they will be home to Kafka's data logs. Also, clients will be writing and reading from brokers to get information into and out of Kafka and demand broker attention.

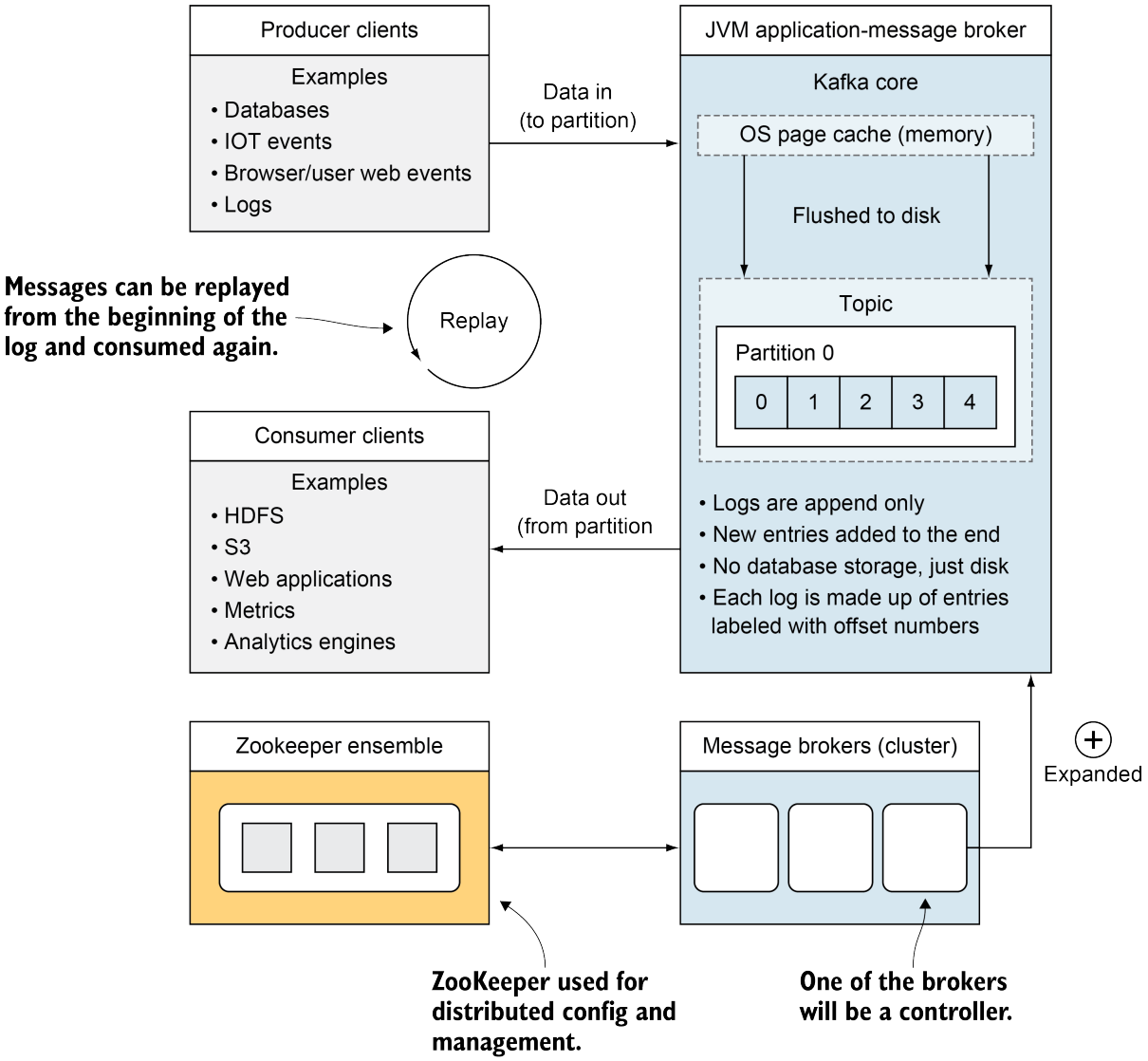

Figure 6.1 Brokers

Producer requests are sent, to the broker, by the producer clients. When a broker starts to handle a request for a lead partition replica that it owns, it performs timeout logic as well as logic pertaining to `RequiredAcks` [68].

The Kafka protcol also defines fetch requests that are also handled by the brokers and are sent by

consumer clients and handle logic such as `MaxWaitTime`. Fetch requests also come from partition replicas that are followers of the leader replica partition for a topic [69]. To remain in-sync, the followers fetch data in order to become up-to-date. To draw a parallel between the consumer clients used in Chapter 5, fetch requests are used inside the Kafka cluster between replicas just like they are by consumer clients outside of the cluster. These inter-broker communications, like follower replicas having to communicate with the leader replica on different brokers, are a part of the job of a broker.

6.4 Settings at the Broker Level

Configuration is an important part of working with Kafka clients, topics, and brokers. Recall the setup steps to create our first brokers from Appendix A, we modified a file titled `server.properties`. This file is a common way to pass a specific configuration to a broker instance and was passed as a command-line argument to the broker startup shell script. As with the producer and consumer configuration, look for the IMPORTANCE label of High in the documentation listed at kafka.apache.org/documentation/#brokerconfigs. The `broker.id` configuration property should always be set to an integer value that will be unique in the cluster. This file also deals with configuration related to listeners, log locations, log retention, ZooKeeper, and group coordinator settings [70].

Server configuration properties are also a good place to provide sensible defaults for our topics. However, these default configuration values can be overridden for specific topics based on the needs of our application and we prefer to make sure they are set (and not defaulted by users when possible). One of the clearest and most important examples of a default would be the property: `default.replication.factor`. This configuration property controls the number of replicas that will be used when a topic is created without the option itself set [71]. So let's say that we want 3 copies of the data as a default. However, let's say that we have other topics whose data is less important in a recovery situation. To make this more concrete, let's think of a website that handles traffic for an online bakery. The order information from customers is considered by the owners to be important enough to have multiple copies. They probably wouldn't want to mess up someone's wedding cake details! However, the section of their website that handles chat history might not be worth attempting to recover in a failure. Kafka allows us to do both reliable and not-as-reliable structures for our topics. What tradeoffs do we have regarding replicas and why wouldn't we always have more than one copy? The more copies of messages we have the more disk space we use over the cluster. Business value should usually help guide that decision.

Let's do an example of what happens when we have only one copy of our data and the broker it is on goes down.

Make sure that our local test Kafka is cluster running with three nodes and create a topic with a `replication.factor` of 1.

Listing 6.2 Listing our topics

```
bin/kafka-topics.sh --create \
  --bootstrap-server localhost:9092 \
  --replication-factor 1 \
  --partitions 1 \
  --topic kinaction_one_replica   ❶

bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 \
  --topic kinaction_one_replica    ❷

Topic: one-replica    PartitionCount: 1    ReplicationFactor: 1    Configs:
    Topic: kinaction_one_replica    Partition: 0    Leader: 2    Replicas: 2    Isr: 2
```

❶ Creating a topic with only 1 partition and 1 replica.

❷ Describing the kinaction_one_replica topic to see all data located on the broker with id 2.

When we run the commands in Listing 6.2 to create and describe the topic `kinaction_one_replica`, we'll see that there is only one value in the fields: Partition, Leader, Replicas, and In-Sync Replicas (ISR) and that the broker will be the same id value. This means that the entire topic depends on that one broker being up and working. If we terminated the broker with id 2 in this example and then tried to consume a message from that topic, we would get a message like the following: `1 partitions have leader brokers without a matching listener`. Since there were no replica copies for the partition of the topic, there is no easy way to keep producing or consuming to that topic without recovery of that broker. While this was just one specific example, it illustrates the importance that sensible defaults for broker configuration can be overridden by users of the cluster by creating their topics manually as in Listing 6.2.

Another important configuration property to define is where our application logs state and errors during normal operation.

6.4.1 Kafka's Other Logs: Application Logs

As with most applications, Kafka provides logs for letting us know what is going on inside of the application. In the discussion that follows, the term application logs will refer to the logs that we usually think of when working with any other application: whether debugging or auditing. These application logs are not related to the record logs that form the backbone of Kafka's feature set. The locations where these application logs are stored are also entirely different than those for records. When we start a broker, the application log directory is in the Kafka base installation directory under the folder `logs/`. We can change this location by editing the `config/log4j.properties` file and the value for `kafka.logs.dir`.

TIP	Log Location For Messages
	Make sure to check the server.properties configuration file before deploying to production. This file is passed in when starting the server, and some of the default configuration values are not meant for production. For example, the `log.dirs` property tells Kafka where to store the partition data, and the default is `/tmp/kafka-logs`. This location is meant to be temporary and is not a great choice for production data. So please check all your log directories!

6.4.2 Server Log

The server log, `server.log` is where we would look if there is a startup error or an exception that terminates the broker. It seems to be the most natural place to look first for any issues. Look or use the `grep` command for the heading `KafkaConfig values`. This is helpful since many errors and unexpected behaviors can be traced back to configuration issues on startup. To make you you are overwhelmed when you look at the directory which holds this file, note that you will likely see other files like controller.log (if the broker had ever been in that role) and older files dated with the same name.

`logrotate` is a tool that can be used for log rotation and compression but there are many other tools available as well to manage older server logs. Also, it is often a good idea to have these logs on a different disk altogether than the one that stores our message logs. The less contention for writing to disk the better for overall performance.

Something else to mention in regards to these logs is that they are located on each broker. They are not aggregated by default into one location. Various platforms might do this on our behalf or we can gather them with a tool like Splunk (www.splunk.com/). It is especially important to remember to gather these logs when using something like a cloud environment in which the broker instance might not exist when we are trying to analyze logs at a later point-in-time.

6.5 Controllers Duties

As we discussed in Chapter 2, each partition will have a single leader replica. A leader replica will reside on a single broker at any given time. A broker can host the leader replica of multiple partitions, and any broker in a cluster can host leader replicas, but only one broker in the cluster will act as the controller. The role of the controller is to manage the state of partitions and replicas for the entire cluster. The controller also will perform other administrative actions like partition reassignment, creating topics, deleting topics, and creating partitions [72].

The controller leverages ZooKeeper to detect restarts and failures in the cluster as a whole. If a broker is restarted, the controller will send the leader information as well as the In-Sync Replica (ISR) information for the broker re-joining the cluster. For a broker failure, the controller will

select a new leader and update the ISR. These values will be persisted to ZooKeeper. In order to maintain broker coordination, it also sends the new leader and ISR changes to all other brokers in the cluster [73]. While a broker is serving as the controller, it can still perform its traditional broker role as well.

When the controller broker fails, the cluster can recover due to having the data in ZooKeeper, however there is a higher operational cost to restarting the controller, as this kicks off the controller election process. So, when we consider a rolling upgrade of a cluster, shutting down and restarting one broker at a time, it is best to do the controller last [74]. Otherwise, we might end up restarting the controller multiple times.

To figure out which broker is the current controller, we can use the zookeeper-shell script to look up the id of the broker as shown in Listing 6.3. The path `/controller` exists in ZooKeeper and we are running one command to look at that current value. Running that command for my cluster showed my broker with id 0 as the controller.

Listing 6.3 Listing the current controller

```
bin/zookeeper-shell.sh localhost:2181   ❶
get /controller   ❷
```

❶ Connect to your ZooKeeper instance.

❷ Use `get` against the controller path.

Figure 6.2 shows all of the output from ZooKeeper and how the brokerid value is returned: `"brokerid":0`. The current cluster broker id 0 not only acts as a broker, but it also is serving as the 1 controller in the cluster with additional duties. If a migration or upgrade of this cluster was to occur, we would upgrade this broker last due to this role.

```
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
get /controller
{"version":1,"brokerid":0,"timestamp":"1540874053577"}
cZxid = 0x2f
ctime = Mon Oct 29 23:34:13 CDT 2018
mZxid = 0x2f
mtime = Mon Oct 29 23:34:13 CDT 2018
pZxid = 0x2f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x166c33ffa650000
dataLength = 54
numChildren = 0
```

Figure 6.2 Example Controller Output

There will also be a controller log file with the name: `controller.log` that serves as an application log on broker 0 in this case. This will be important when we are looking at broker actions and failures. The `state-change.log` is also useful as it records the decisions that the controller logic completed.

6.6 Partition Replica Leaders and their role

As a quick refresher, topics are made up of partitions. Partitions can have replicas for fault tolerance. Also, partitions are written on the disks of the Kafka brokers. One of the replicas of the partition will have the job of being the leader. The leader is in charge of handling the writes for that partition from external producer clients. Since this leader is the only one with the newly written data, it also has the job of being the source of data for the replica followers. The In-Sync Replicas (ISR) list is maintained by the leader. Thus, it knows which replicas are caught up and have seen all the current messages. The replicas will act as consumers of the leader partition and will fetch the messages. Similar to how external consumers have offset metadata about which messages have been read, metadata is also present with replication offset checkpointing to track which records have been copied to the followers [75].

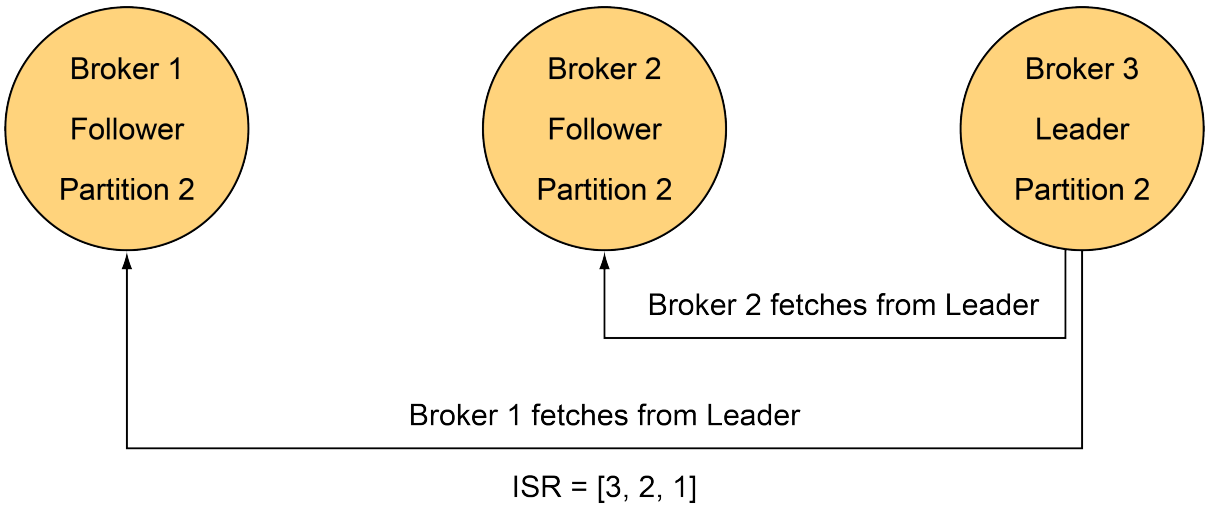

Figure 6.3 Leader

Figure 6.3 shows a 3 node cluster and an example of how leadership is defined. Broker 3 holds the leader replica for partition 2. Broker 2 and Broker 1 are followers of this leader. As the leader, Broker 3 will handle all of the reads and writes from external producers and consumers. The leader will also handle the fetch requests it receives from the other two brokers as they pull new messages into their copies. The ISR list would include the leader as the first position (3) and the remaining followers that are staying up-to-date on their copies of the messages from the leader: ie, ISR list [3,2,1].

In some cases, the broker that fails could have hosted the leader replica for a partition. The Controller and ZooKeeper would be leveraged in this case to reassign the leader for that partition. In Figure 6.4, the previous example in Figure 6.3 has experienced a failure. Since this broker 3 is not available, a new leader is elected. This new leader in this example is Broker 2. Once a follower, it was elected as a leader replica to keep Kafka serving and receiving data for that partition. The ISR list is now [2,1] with the first position reflecting the new leader replica being hosted on broker 2.

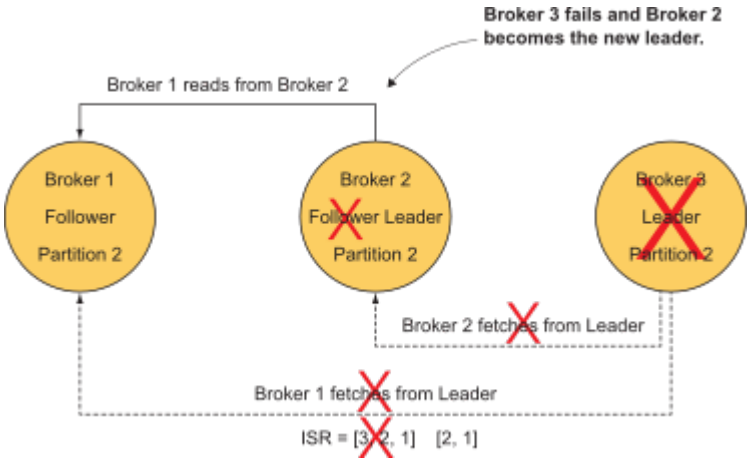

Figure 6.4 New Leader Elected

In Chapter 5 we discussed a Kafka Improvement Proposal KIP-392 which allows consumer clients to fetch from the closest replica [76]. Reading from a preferred follower (rather than the leader replica) is something that might make sense if our brokers span physical data centers. If our network connections are slow due to the leader being remote or costs are higher, this is a valid option to explore. However, when discussing leaders and followers in this book, unless stated otherwise, we will focus on default leader read and write paths.

6.7 In-Sync Replicas (ISRs) Defined

In-Sync Replicas (ISRs) are a key piece to really understanding Kafka. For a new topic, a specific number of replicas will be created, and added to the initial ISR list. This number can either be from a parameter or as a default from the broker configuration.

One of the details to note with Kafka is that replicas do not heal themselves by default. If you lose a broker on which one of your copies of a partition existed, Kafka does not currently create a new copy. I mention this since some users are used to filesystems like HDFS that will maintain their replication number (self-heal) if a block is seen as corrupted or failed. So an important item to look at when monitoring the health of our system might be how many of our ISRs are indeed matching our desired number.

Why is watching this number so important? Let's say that we have a topic that is only 1 partition. That partition is replicated 3 times. In the best-case scenario, we would have 2 copies of the data that is in our lead partition replica. This of course means that the follower replicas are caught up with the leader. But what if we lose another ISR? It is good to keep a pulse on how many copies you have before it hits 0!

It is also important to note that if a replica starts to get too far behind in copying messages from the leader, it might be removed from the list of in-sync replicas (ISRs). The leader will notice if a follower is taking too long and will drop it from its list of followers [77]. The leader will continue to operate with a new ISR list. The result of this slowness to the ISR list is the same as in Figure 6.4 in which a broker failed. The broker is removed, the ISR list is smaller, and we have a corresponding weakening of fault tolerance.

If we need all of our data and can't miss a single message, then we need to avoid this scenario. If we find that our ISRs are falling below the desired number, we will need to troubleshoot and correct the situation.

6.8 Unclean Leader Election

An important tradeoff to consider is the importance of uptime vs the need to keep our data from being lost. What if we have no in-sync-replicas but lost our lead replica due to a failure? When `unclean.leader.election.enable` is true, the controller will select a leader for a partition even if it is not up-to-date, and the system will keep running. The problem with this is that data could be lost since none of the replicas had all of the data at the time of the leader's failure. While data loss might not be okay for payment transactions, it might be okay for a web analytics engine to miss a hover event on a couple of pages. The `unclean.leader.election.enable` setting is cluster-wide, but can be overridden at the topic level [78].

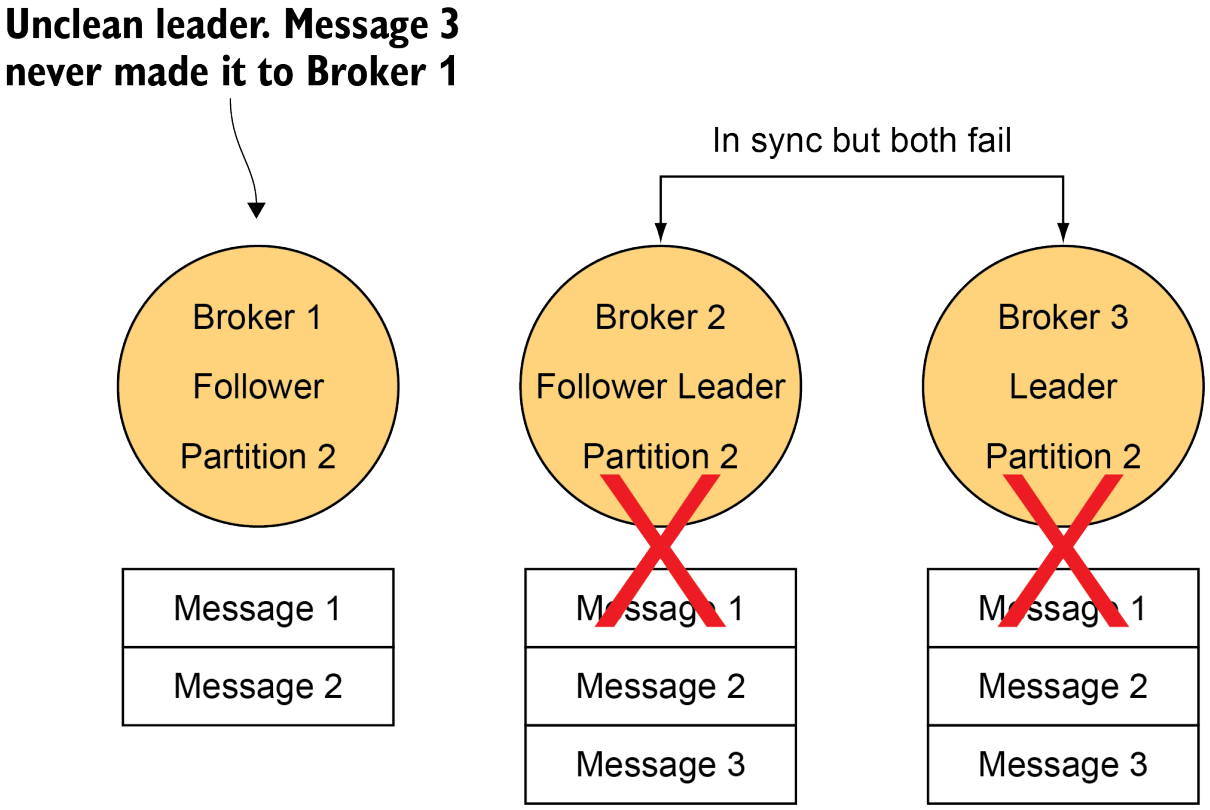

Figure 6.5 Unclean Leader Election

Figure 6.5 shows data loss in the case of a partition that has a total of 3 replicas. In this case, both Broker 3 and 2 failed and are not online. Since unclean leader election was allowed, Broker 1 is made the new leader even though it was not in-sync with the other brokers. Broker 1 never saw `Message 3` and would not be ever able to present that data to clients. At the cost of missing data, this option allows us to keep serving clients if our business case warrants this tradeoff.

6.9 Metrics from Kafka

Kafka, written for the Java virtual machine (JVM), uses Java Management Extensions (JMX) to enable us to peek into the Kafka process. There are many tools we could use to capture and view this data from our applications. We will look at the steps to use Grafana® (grafana.com/) [79] and Prometheus® (prometheus.io/) to set up a simple monitoring stack that will provide a visual representation of the activity in our cluster. We'll use Prometheus to extract and store the Kafka metrics data. Then we'll send that data to Grafana to produce helpful graphical views.

To fully understand why we are setting up all of the following tools, let's quickly review the components and the work each one is doing in Figure 6.6.

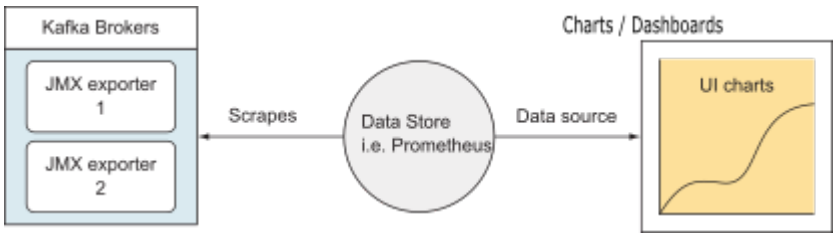

Figure 6.6 Graph Flow

- JMX is leveraged to look inside the Kafka applications.
- The Kafka exporter is taking the JMX notifications and exporting them into the Prometheus format.
- Prometheus scrapes the exporter data and stores the metrics data.
- Grafana is taking the information from Prometheus and displaying that information in a visual dashboard.

There are many Docker images and docker-compose files that have all of these tools bundled up or you could install each tool to a local machine in order to explore the steps above in greater detail.

For the Kafka exporter, let's work with a tool available at github.com/danielqsj/kafka_exporter. I prefer the simplicity of this tool since we can just run it and give it one (or a list) of Kafka servers to watch and it might work well for your use-cases as well. Notice that we will get many client and broker specific metrics since there are quite a few options that we might want to monitor. However, this is not a complete list of the metrics available to us.

Figure 6.7 shows a query against a local data store, like a local instance of Prometheus, that is gathering metrics from our kafka_exporter tool. As we discussed about partitions, Kafka replicas do not heal themselves automatically, so one of the things we should monitor is under-replicated partitions. If this number is above 0, we might want to look at what is going on in the cluster and why there is a replica issue. We might display the data from this query in a chart or dashboard, or potentially send an alert.

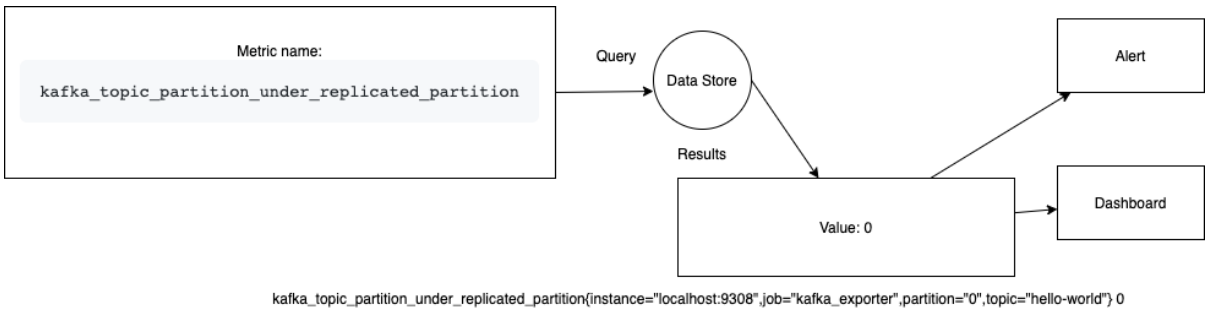

Figure 6.7 Metric Query Example

As noted, the Kafka exporter we are using does not expose every JMX metric. To get more JMX metrics, we can set the `JMX_PORT` variable when starting our Kafka processes [80]. Other tools are available that use a Java agent to produce the metrics to an endpoint or port which Prometheus can scrape. Java agents use the Java Instrumentation API to gather data about our applications.

Listing 6.4 Starting a broker with a JMX Port

```
JMX_PORT=9990 bin/kafka-server-start.sh config/server0.properties ❶
```

❶ Adding the JMX_PORT variable when starting the cluster

Listing 6.4 shows setting the variable `JMX_PORT` when starting a broker. If we already have a broker running and do not have this port exposed, we will need to restart the broker to affect this change. We may also want to automate the setting of this variable to ensure it is enabled on all future broker restarts.

6.9.1 Cluster Maintenance

As we consider moving to production, we will want to configure more than one server. There is nothing wrong with starting small, we could start with 3 brokers and 3 ZooKeeper nodes for each production workload. This number comes from needing an uneven number for ZooKeeper and also more than 1 server for failure cases. Starting with small clusters will help minimize cost and maintenance, while still allowing for failure or maintenance of one server per cluster while maintaining functionality.

Another decision to think about is whether you want one giant cluster for your entire enterprise or one per product team. Some organizations, like LinkedIn where Kafka was created, have 60 brokers (if not more) as part of one production cluster (and multiple other clusters as well) [81].

Other teams prefer to keep smaller clusters that power their smaller product suites. There are many ways to handle a production cluster and both methods listed above can work.

Another item to note is that various pieces of the ecosystem like Kafka clients, Connect clients, Schema Registry, and the REST Proxy do not usually run on the same servers as the brokers

themselves. While we might be running all of these exercises on a laptop and this shows that it can be done, we definitely don't want all of these processes running on a single server when we are trying to handle production workloads safely and efficiently.

To draw a parallel to similarities with tools from the Hadoop ecosystem, Kafka can scale well horizontally with more servers. Let's look at what it takes to add a server to a cluster.

6.9.2 Adding a Broker

Beginning with a small cluster is a great way to start, as we can always add brokers to grow our footprint. To add a Kafka broker to our cluster, we just start-up a new Kafka broker with a unique id. That is pretty much it! But, there is something to be aware of in this situation. The new broker will not be assigned any partitions. Any topic partitions that were created before this new broker was added would still exist on the brokers that existed at the time of their creation. Topic partitions are not moved automatically [82]. If we are okay with the new broker only handling new topics, then we don't need to do anything else. However, if we do want to change the existing layout, we can use the tool: `kafka-reassign-partitions`. This script is located in the `bin` directory of our Kafka installation. There are other tools that have been developed to help with this situation. One such tool is the Cluster Manager for Apache Kafka (CMAK) (github.com/yahoo/CMAK) that also has other options to help us manage our cluster.

6.9.3 Upgrading your Cluster

As with all software, updates and upgrades are a part of life. Not all systems can be bought down all at once and upgraded due to production workloads or business impact. One technique that can be used to avoid downtime for our Kafka applications is the rolling restart. This means just upgrading and restarting one broker at a time. Figure 6.8 shows each broker being upgraded one at a time before moving onto the next broker.

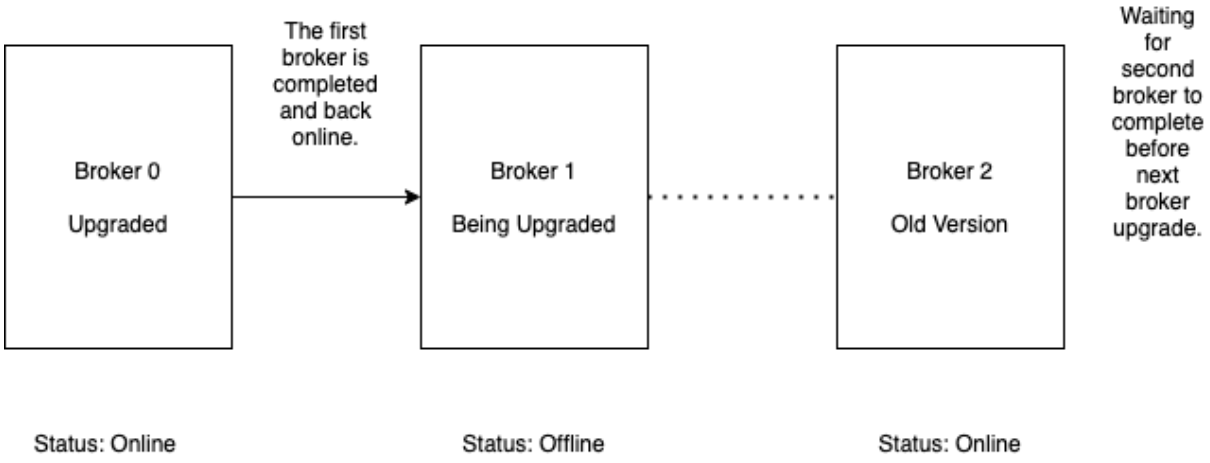

Figure 6.8 Rolling Restart

One very important broker configuration property is `controlled.shutdown.enable`. Setting this to `true` will enable the transfer of partition leadership before a broker shuts down [83]. Another setting to be aware of is `min.insync.replicas`. Since Kafka does not automatically create a new partition replica on a different broker, we could be short one partition from our in-sync-replica total during broker downtime [84]. If we start with 3 replicas and remove a broker that hosts one of those replicas, a minimum replica number of 3 will result in clients not being able to continue until the broker rejoins and syncs up. We need to be aware of these numbers and the impact they have on our applications.

As a reminder, another good practice is to identify which broker is serving as the controller in our cluster and let it be the last server upgraded. Not having the controller bounce between brokers will make the upgrade process more efficient.

Why else besides broker upgrades might we want to perform a rolling restart? Some broker configuration changes require a restart of Kafka. Applying OS security patches, or other server maintenance tasks might require restarting the broker's host server. And don't forget about the Java virtual machine! Kafka and ZooKeeper both depend on Java to run and Java itself might have security patches, or version upgrades that need to be applied. These will also require broker restarts and should be part of your overall cluster maintenance plan.

6.9.4 Upgrading your clients

While Kafka does its best to decouple the clients from the broker, it is always a good idea to know the versions of clients with respect to brokers. Clients can usually be upgraded after all of the Kafka brokers in a cluster have already been upgraded.

There is also the option to use newer clients without worrying about the broker version. This bidirectional client compatibility feature was a new feature in Kafka 0.10.2. Brokers version 0.10.0 or later support this feature [85]. As with any upgrade though, take a peek at the version notes to make sure you will have compatibility for newer versions. However, just because you can does not mean that you should. For example, performance could be impacted by brokers and clients that have to convert between versions. If possible, it is usually worth keeping clients and broker versions in sync.

6.9.5 Backups

Kafka does not have a backup strategy like one would use for a database. We don't take a snapshot or disk backup per se. Since Kafka logs exist on disk, why not just copy the entire partition directories? While nothing is stopping us from doing this, one concern is making a copy of the data directories. To do an accurate copy without missing data, we would probably have to stop the broker before performing the copy. If we wanted a complete copy of a topic with more than one partition, we could be talking about multiple brokers. Rather than performing manual copies and coordinating across brokers, one of preferred option is for a cluster to be backed by a second cluster. Between the two clusters, events are then replicated between topics. One of the earliest tools that you might have seen in production settings is called MirrorMaker. The basic architecture of MirrorMaker logically sits between two separate clusters.

This tool can basically be thought of as a consumer client and producer client hooked together that can act as a standalone tool. The data from the source cluster is consumed with a consumer client and then written to the backup cluster with a producer client. Since the clusters are separate, data is taken from the original cluster. Then data lands on a same-named topic on the second cluster. This independence includes offsets that could be different. One detail to note is that the event's key (if using a keyed message) will be the same so partitioning (and thus ordering) would be correct [86].

Listing 6.5 MirrorMaker

```
bin/kafka-mirror-maker.sh \   ❶
  --consumer.config source-consumer.properties \   ❷
  --producer.config target-producer.properties \   ❸
  --whitelist kinaction_helloworld   ❹
```

❶ Shell script to start MirrorMaker.

❷ Passing configuration to the consumer.

❸ Passing configuration to the producer.

❹ The whitelist names the topic for replication.

When we run the MirrorMaker tool from a terminal, as shown in Listing 6.5, we will pass it configuration for the source and destination clusters [87]. Note that we do not have two local clusters, so this command is an example only. The `--whitelist` option is used to let the tool know which topics should be replicated. This value can also be a regular expression, in which case all matching topics will be replicated. The `source-consumer.properties` file will hold the ZooKeeper server as well as the bootstrap servers for the brokers for the cluster that we want to be the source of our data. The `target-producer.properties` file will hold similar information for the destination or copy cluster. We can also set parameters like the key and value serializers or any other producer properties we would set if we were writing the client ourselves.

While MirrorMaker is still a popular option, a newer version of this tool called MirrorMaker 2.0 was released with Kafka version 2.4.0 [88]. In the `bin` directory of the Kafka install directory, we will find the shell script named `kafka-mirror-maker` which was used in Listing 6.5 as well as the new MirrorMaker 2.0 script named `connect-mirror-maker`. Listing 6.6 shows the usage of the new MirrorMaker 2.0 tool.

Listing 6.6 MirrorMaker 2.0

```
bin/connect-mirror-maker.sh mirror-maker2.properties    ❶
```

❶ This version of MirrorMaker takes a configuration file to define the replication requirements.

Where the first version of MirrorMaker used consumer and producer clients, version 2.0 leverages the Kafka Connect framework instead. The framework is reflected in the script name itself `connect-mirror-maker`. Another difference from the previous version of the tool, is that a configuration file holds the information needed for replication rather than command line arguments. Listing 6.7 shows an example configuration file (not meant to run without updates neeeded) that could be passed into the MirrorMaker 2.0 script based off the file `connect-mirror-maker.properties` in the Kafka source code [89].

Listing 6.7 MirrorMaker 2.0 Configuration File

```
clusters = Cluster1, Cluster2     ❶

Cluster1.bootstrap.servers = Cluster1_localhost:9092, Cluster1_localhost:9093     ❷
Cluster2.bootstrap.servers = Cluster2_host0:9092, Cluster2_host1:9092

Cluster1->Cluster2.enabled = true   ❸

Cluster1->Cluster2.topics = .*    ❹

# Internal Topic Settings     ❺
checkpoints.topic.replication.factor=3
heartbeats.topic.replication.factor=3
offset-syncs.topic.replication.factor=3
offset.storage.replication.factor=3
status.storage.replication.factor=3
config.storage.replication.factor=3
```

❶ Cluster aliases for referencing each cluster.

❷ Each cluster will have a list of brokers to connect to on startup.

❸ Cluster 1 will replicate to Cluster 2.

❹ Cluster 1 will replicate all topics to Cluster 2 due to the match all regular expression.

❺ Internal topics will also be replicated.

In this example, the provided configuration in Listing 6.7 is setting up a primary cluster, which

we give an alias of Cluster1, to replicate to Cluster2. Note that we do not have two local clusters, so this command and configuration are an example only. Each cluster provides a list of bootstrap servers for initial connections to be made. While replication can also be enabled from Cluster2 to Cluster1, this configuration is only enabling one-way replication. All topics from Cluster1 are also being replicated due to the regular expression `.*`. Besides the change to use Kafka Connect, what are the advantages of moving to MirrorMaker 2.0? The feature set has expanded to detect new topics and partitions, automatically sync topic configuration, and even MirrorMaker-compatible legacy mode are a few of the benefits [90].

There are also some other open-source, as well as enterprise offerings, for mirroring data between clusters. Confluent Replicator has a proprietary license but also some pretty nice features that are similar to the MirrorMaker 2.0 current feature set.

6.10 A Note on Stateful Systems

One of the ongoing trends for many technology companies is the drive to move applications and infrastructure to the cloud. Kafka is an application that definitely works with stateful data stores. In this book material, we will be working on our own nodes and not with any cloud deployments. Instead of working with details related to Kubernetes, Apache Mesos, Docker, and AWS, we will try to keep focused on Kafka and just acknowledge that many companies do work with Kafka in these environments. There are some great resources including Confluent's site on using the Kubernetes Confluent Operator API (www.confluent.io/confluent-operator/) as well as Docker images available to do what you need. Another interesting option is Strimzi (github.com/strimzi/strimzi-kafka-operator) if you are looking at running your cluster on Kubernetes or OpenShift. At the time of this writing, it is a Cloud Native Computing Foundation (www.cncf.io/) sandbox project. If you are familiar with the above tech tools, it might be a quick way for you to kick the tires on a Proof of concept (PoC) setup if you find some interesting projects out in Docker Hub.

However, there is not a one-size-fits-all mandate for our infrastructure. So let's talk about some reasons that we might have Kafka in a cloud or Kubernetes environment: using Kubernetes as a focus point to narrow our discussion. If we have a majority (if not all) of our applications in a specific setup, it might make sense to keep that setup for our Kafka environment. If Kubernetes runs most of the applications at our company, it might not be a good idea to do something different for our Kafka setup. Another benefit of Kubernetes that stands out would be the ability to create new clusters quickly and with different storage and service communication options which Gwen Shapira explores further in the paper Recommendations for Deploying Apache Kafka on Kubernetes [91]. For some companies, giving each product its own cluster might be easier to manage than having one huge cluster for the entire enterprise. The ability to spin up a cluster quickly rather than adding physical servers can provide the quick turnaround products need to leverage Kafka in their business.

A common question is why does the challenge of state come up when talking about containerized environments? Containers are usually meant to have short lifespans. The idea of scaling your application by starting up or removing compute instances as needed works well when state does not have to be migrated first to avoid being lost with the image. However, use-cases were developed over time with the state being managed by external data sources that could be mounted into the container.

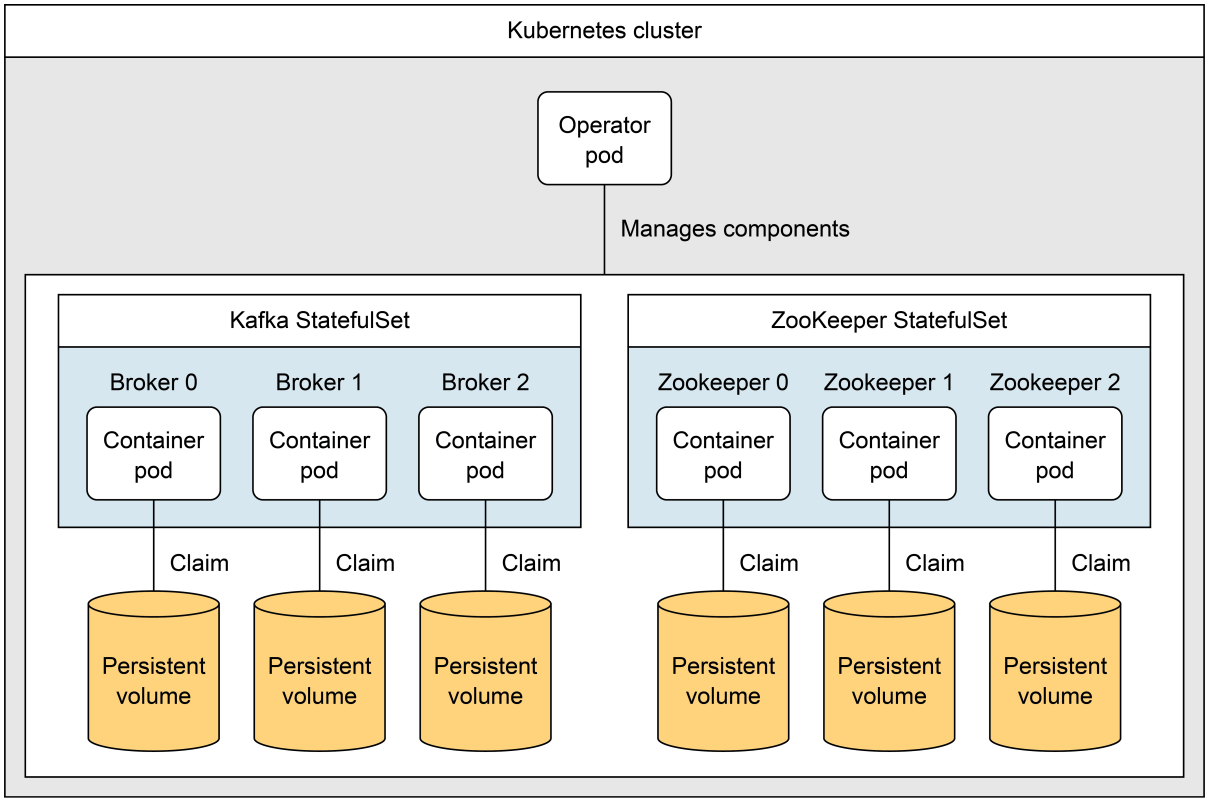

Figure 6.9 Kubernetes on Kafka

Figure 6.9 shows a very general outline of how Kafka brokers could be set up in Kubernetes with an operator used - similar to how the Confluent and Strimzi operators might work. These terms are Kubernetes specific and I do not provide much explanation since I do not want to shift the focus away from learning about Kafka itself but rather give a general overview. This operator is its own pod that lives inside of the Kubernetes cluster. Each broker will be in its own pod as well as a part of a logical group called a StatefulSet. The purpose of the StatefulSet is to manage the Kafka pods and help guarantee ordering and a persistent identity for each pod. So if the pod that hosts a broker (the JVM process) with id 0 fails, a new pod will be created with that identity (not a random id) and will be able to attach to the same persistent storage volume as before. Since these volumes hold the messages of the Kafka partitions, the data is maintained. This statefulness helps overcome the sometimes short lives of containers. Each ZooKeeper node would also be in its own pod and part of its own StatefulSet.

For those who are new to Kubernetes or are anxious about the transition to such a platform, one

migration strategy that can be helpful is to run Kafka clients and applications on a Kubernetes cluster before the Kafka brokers. Besides being stateless, running our clients in this manner can help us get a feel for Kubernetes at the start of our journey. However, we should not neglect the need to understand Kubernetes well in order to run Kafka well on top of this platform.

One developer team of 4 that I worked with recently focused half of the team on Kubernetes and half on running Kafka. Of course, this ratio might not be what every team encounters. Developer time required to focus on Kubernetes depends on our team and experiences overall as we move ahead in our deployment journey.

6.11 Exercise

Since it can be hard to apply some of our new learning in a hands-on manner and since this chapter is heavier on commands than code, it might be helpful to have a quick exercise to explore a different way to discover the metric under-replicated partitions. Besides using something like a Grafana dashboard to see this data, what command-line options can we use to discover this information?

- Let's say that we want to confirm the health of one of our topics named `kinaction_replica_test` that we created with each partition having 3 replicas. We would want to make sure we have 3 brokers listed in the ISR list in case there is ever a broker failure. What command should we run to look at that topic and see the current status?

Listing 6.8 shows an example of describing that topic. Notice that the `ReplicationFactor` is 3 and the Replicas list shows 3 broker ids as well. However, the ISR list only shows 2 values when it should show 3!

Listing 6.8 Describing topic replica-test for ISR count

```
$ bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 \
  --topic kinaction_replica_test    ❶
Topic:kinaction_replica_test    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: kinaction_replica_test    Partition: 0    Leader: 0    Replicas: 1,0,2    Isr: 0,2
 ❷
```

❶ Note the topic parameter and the describe flag being used.

❷ Topic specific information about leader, partition, and replicas.

While we noticed the issue by looking at the details of the command output, we could have also used the --under-replicated-partitions flag to see any issues quickly! Listing 6.9 shows using the flag `--under-replicated-partitions`. This flag quickly filters out the hard to see ISR data and will only output under replicated partitions to the terminal [92]!

Listing 6.9 Under-replicated-partitions flag

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --under-replicated-partitions   ❶
Topic: kinaction_replica_test    Partition: 0    Leader: 0    Replicas: 1,0,2    Isr: 0,2   ❷
```

❶ Note the under replicated partition flag being used

❷ Notice that the ISR result only lists 2 brokers!

Listing 6.9 shows that when using the `--describe` flag, we do not have to limit the check for under replicated partitions to a specific topic. We can run this command to display issues across topics and to quickly find issues on our cluster. We will explore more of the tools included with Kafka out-of-the-box when we talk about administration tools in Chapter 9. When using any of the commands in this chapter, it is always a good idea to run the command without any parameters and read the command options that are available for troubleshooting.

As we examined more about Kafka in this last chapter, we've come to realize we are running a complex system. However, there are various command-line tools as well as JMX metrics to help us monitor the health of our cluster. In our next chapter, we will continue to use commands to complete specific tasks like rebalancing our topic partitions as well as maintaining this dynamic system throughout its lifetime.

6.12 Summary

- Brokers are the centerpiece of Kafka and provide the logic with which external clients interface with our applications. More than one broker makes up a cluster to help provide not only scale but also reliability.
- ZooKeeper (or the Kafka Raft Metadata mode) is leveraged in order to provide agreement in a distributed cluster. One example is to elect a new controller between multiple available brokers.
- Configuration can be set at the broker level to help manage your cluster. This can help set reasonable defaults that your clients can override for specific options.
- Replicas allow for a number of copies of data to span across a cluster. This helps in the event a broker fails and can not be reached. In-Sync Replicas (ISRs) are those that are current and up-to-date with the leader's data and can take over leadership for a partition without data loss.
- JMX can be used to help produce graphs to visually monitor a cluster or alert on potential issues.
- Mirror Maker (now at version 2.0) is an option to provide a backup strategy across different clusters for those looking to make sure their data is highly available.

7 Topics and partitions

This chapters covers

- Diving into topics including creation parameters and configuration options
- Exploring how partitions exist as log files written on disk
- Examining how segments impact the data inside partitions
- Check out a way to test with `EmbeddedKafkaCluster`
- Evaluating topic compaction and its impact on how data can be retained

In this chapter, we will look further into how we might store our data across topics as well as how to create and maintain topics. This will include how partitions fit into our design considerations and how we can view our data on the brokers. All of this information will help us as we look at how to make a topic show data like a database table that updates, rather than an append only log!

7.1 Topics

To quickly refresh our memory, it is important to know that a topic is more of a logical concept rather than a physical structure. It does not usually exist on only one broker. Most applications consuming Kafka data will view that data as being in a single topic: no other details are needed for them to subscribe. However, behind the topic name are one or more partitions which actually hold the data. The logs that are written to the broker filesystems that make up a topic are the result of Kafka writing the data in the cluster. Figure 7.1 shows partitions that make up one topic named `helloworld`. A single partition's copy is not split between brokers and has a physical footprint on each disk. Figure 7.1 also shows how those partitions are made up of messages that are sent to the topic.

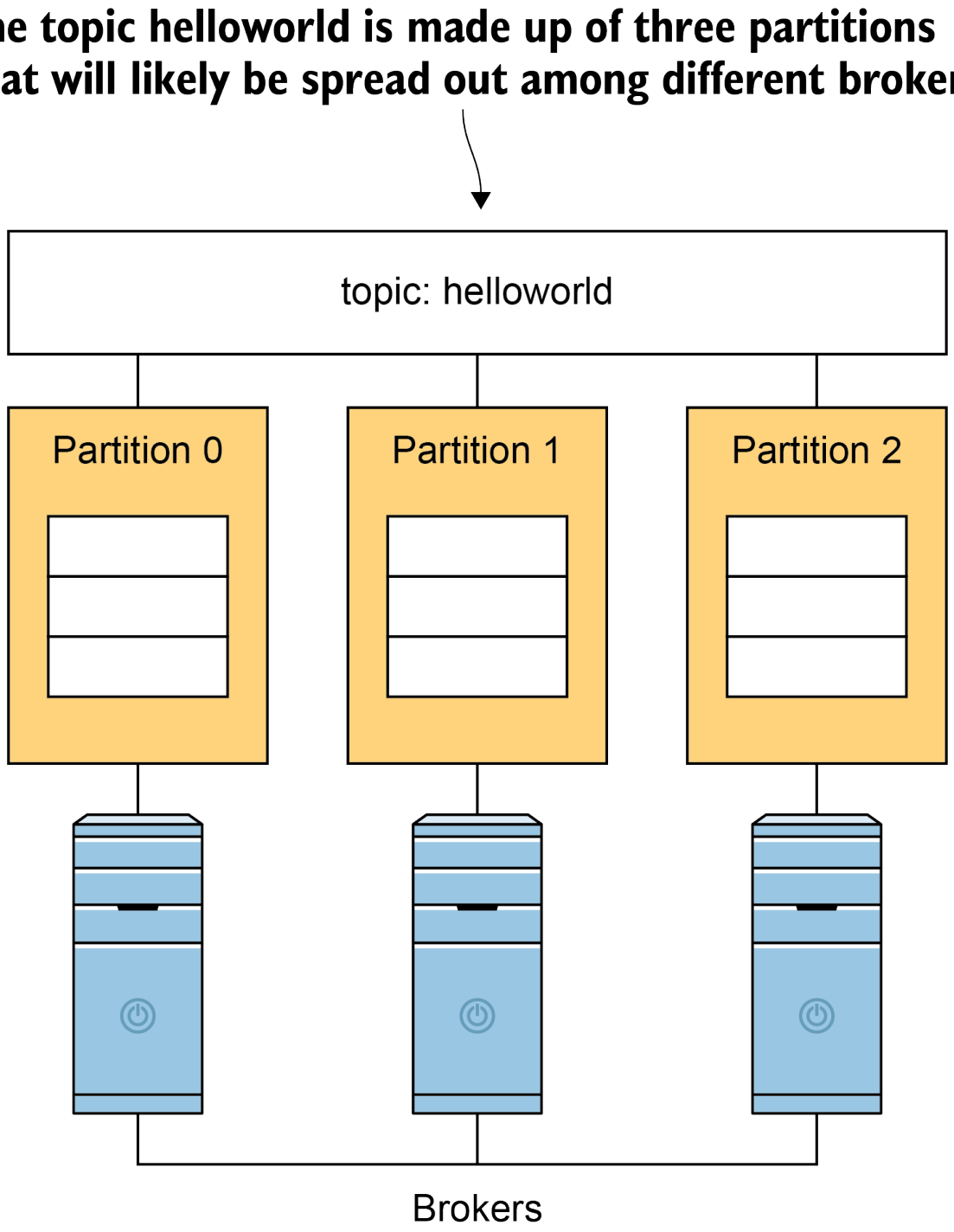

Figure 7.1 Example Topic With Partitions

If writing to a topic is so simple in getting started examples, why do we need to understand the role and pieces that make up a topic? At the highest level, how our consumers get to the data will be impacted. Let's say that our company is selling reservations for a training class using a web-based application that sends events of user actions into our Kafka cluster. Our overall application process could generate a number of events. For example, there would be an event for the initial search on location, one for the specific training being selected by the customer, and a third for payment being received. Should the producing applications send all of this data to a

single topic or several topics? Should we decide that each message is a specific type of event and should remain separated in different topics? There are tradeoffs with each approach and some things to consider that will help us determine the best approach to take in each situation.

We see topic design as a two-step process. The first being for the events I have, do they belong in 1 topic or more than one. Second, for each topic, what is the number of partitions I should use?. The biggest takeaway is that partitions are a per-topic design question, not a cluster-wide limitation or mandate. While we can set a default number to be used in topic creation, in most cases we should consider the usage of the topic and the data it will hold. We should have a reason to pick a specific number of partitions and Jun Rao has a fantastic article on the Confluent blog about this every subject titled: "How to choose the number of topics/partitions in a Kafka cluster?" [93].

Let's say that we want to have a partition for each server as a generic rule. However, note that just because we have a partition on a server, does not mean producers will write evenly between them. To do so, we would have to ensure that each partition leader is spread out in that manner and stays that way. We need to get familiar with your data! Let's take a look at a checklist of items to think about in general and in this scenario:

- Data correctness

First off, data correctness is at the top of most data concerns in real-world designs. In regards to topics, this would involve making sure that events that must be ordered end up in the same partition (and thus the same topic). While events can likely be placed in an order based on an event timestamp by your consumers, it will be more trouble (and error-prone) to handle cross-topic event coordination than it is worth. If we are using keyed messages and need them in order, we will very much care about partitions and any future changes to those partitions. With our three example events above, it **might** be helpful to place the events with a message key including the customer id for the actual 'booked' and 'paid' events in three seperate topics. These events are customer-specific and it would be helpful to make sure that payment of a booking occurs for that specific customer. The search events themselves, however, may not be of interest or need to be ordered for a specific customer if our analytics team is looking for the most popular searched cities rather than customer information.

- Volume of messages of interest per consumer

Next we should consider the volume of messages of interest per consumer. For the training system above, let's look at the number of events when we consider the topic placement. The search events themselves would far outnumber the other events. Let's say that a training location near a large city gets 50,000 searches a day but only has room for 100 students. Traffic on most days produces 50,000 search events and less than 100 actual 'booked' training events. Would the

Billing team have an application that would want to subscribe to a generic event topic in which it used or cared about less than 1% of the total messages? Most of the consumer's time would be, in effect, filtering out the mass of events to process only a select few.

- How much data you will be trying to process

Another point to consider is how much data we will be processing. Will the number of messages require multiple consumers to be running in order to process within the time constraints required by our applications? If so, we have to be aware of how the number of consumers in a group is limited by the partitions in our topic. It is easier at this point to create more partitions than you think you might require. Having more capacity for consumers to grow, will allow us to increase in volume without having to deal with re-partitioning data. However, it is important to remember that partitions are not an unlimited free resource per Rao's article above. More partitions require more open file handles for example. Also, having more brokers to migrate, in case of a broker failure, could be a headache in the future. It's best to find a happy medium and go with that as we design our system. Figure 7.2 shows how our design might be best suited with two topics for the three event types we used in our scenario. As always, more requirements or details could change our implementation in the future.

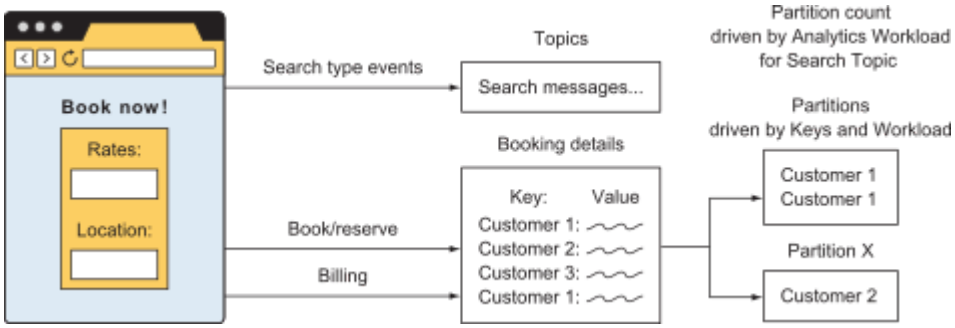

Figure 7.2 Example Training Event Topic Design

Another thing to consider when deciding on the number of partitions for a topic, is that reducing that number is not currently supported [94]. There may be ways to do it, but it is definitely not advised. Let's think about why this would not be desirable. While consumers subscribe to a topic, they really are attached to a partition. The removal of a partition would lose context and out-of-order messages could occur when/if a consumer starts reading from a new reassigned partition. It is somewhat of a parallel to how a commit history on a public Git repository can seem inconsistent for clients after a rebase. If a developer rebases, the branches have to be merged together later. This results in two sets of commits and history has changed! History is a big deal for source control systems! While not exactly the same data impacts, the client view has a harder time following the upstream changes. This is where we need to make sure our keyed messages and consuming clients can follow any changes we make at the broker level. We can impact consumers with your actions!

Now that we've discussed topic design, let's dig a little deeper into the options that we can set

when creating topics. We touched on these briefly when we created topics to produce messages to in Chapter 3.

7.1.1 Topic Creation Options

Kafka topics have a couple of core options that must be set in order to create a topic. While we have been creating topics since chapter 2 with our `helloworld` topic, we need to make sure we dig into the basic parameters that were glossed over. These include the following: `bootstrap-server`, `topic`, `partitions`, and `replication-factor` [95]. For these options listed, it would be best to treat these decisions with thought and care. Defaults might be a good catch-all to make sure topics are created with certain levels of failover (like a minimum of 3 replicas) but our preference is to be intentional.

Another important decision to make at creation time is if you ever want to delete a topic. Since this operation is a pretty big deal that we want to make sure can not happen without a logical confirmation, Kafka requires us to configure the option `delete.topic.enable`. If this is set to 'true' we will be able to successfully delete the topic and it will be removed [96].

Honestly, since client code and version changes may have different required options or commands in the future, it is nice to know that Kafka scripts, in general, have good usage documentation: I would recommend first just running the command `kafka-topics.sh` to see what various actions you can attempt. Listing 7.1 shows our incomplete command to get help.

Listing 7.1 Listing our topic options

```
bin/kafka-topics.sh      ❶
```

❶ Running the generic kafka topic related command

Looking through that output, `--create` stands out. Adding that parameter to our still incomplete command helps us get further information related to the create action itself such as `Missing required argument "[topic]"`. Listing 7.2 shows our incomplete command being built a little further.

Listing 7.2 Listing our topic options

```
bin/kafka-topics.sh --create      ❶
```

❶ Adding the --create option to the Kafka topics command to look at the command-specific error as well as the help documentation again

Why spend time even talking about these steps? Some users are familiar with manual (man) pages as part of their Unix work. Even though Kafka does not present data about how to use the tooling in that manner, it is available before you have to search on Google!

As we think of a name to represent our data, there are a couple of rules that might help us avoid random problems later. Avoid trying to use spaces. Another gotcha that we touched on while looking at how Kafka manages consumer offsets is the informal standard of internal topic names starting with two underscores like `__schemas`. To avoid any confusion, we should not begin our topics with two underscores. We are even prompted with a warning when creating new topics about metric names being a cause for consistency in our topic naming conventions.

Once we have a name that does not conflict with any existing topics, we can create it. For our examples, we'll create 'kinaction_topic' with a replication factor of two and with two partitions. Listing 7.3 shows the syntax to be used in the command prompt.

Listing 7.3 Creating another topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --create \    ❶
  --topic kinaction_topic \    ❷
  --replication-factor 2 \    ❸
  --partitions 2              ❹
```

❶ Adding the create option to our command

❷ Naming our topic

❸ Making sure we have 2 copies of our data

❹ Declaring our topic will consist of 2 partitions

After we create our topic, we can describe that same topic in order to make sure our settings look correct.

```
~/kafka $ bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic kinaction_topic
Topic: kinaction_topic  PartitionCount: 2       ReplicationFactor: 2    Configs: segment.bytes=1073741824
        Topic: kinaction_topic  Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: kinaction_topic  Partition: 1    Leader: 0       Replicas: 0,2   Isr: 0,2
```

Figure 7.3 Describing Topic With 2 Partitions

Notice in figure 7.3 how our partition and replication-factor should match the command we just ran.

Another option that is good to take care of at the broker level is to set `auto.create.topics.enable` to false [97]. Doing this makes sure that we are creating topics on purpose and not by a producer sending a message to a topic name that was mistyped and never actually existed (before a message was attempted). While not very tightly coupled, producers and consumers do usually need to know the correct topic name of where their data should live. This automatic topic creation can cause confusion. Note that while testing and learning Kafka, auto-created topics can be helpful.

For a concrete example, if we run the command `kafka-console-producer.sh`

`--bootstrap-server localhost:9092 --topic notexisting` without that topic existing, Kafka will create that topic for us. When we run `kafka-topics.sh --bootstrap-server localhost:9092 --list` we would now have that topic (`notexisting`) in our cluster. It will use the broker-level defaults for the required fields that we would have specified if we had created the topic manually.

If we create a new topic but forget to set the correct partition size, we can use the `--alter` option to change the partition count of an existing topic [98]. Making this change right away before we add data is the easiest option by far. If we already have data in the partitions, the story becomes more complex, especially for keyed messages. Even if we are not concerned about keyed messages ending up on a specific partition, it is still important to understand that any additional partitions will be empty until more messages are produced. Kafka does not automatically move existing data for us.

If our consumer does depend on message keys to partition data, our options might be a little more nuanced, as changing partition count will change partition assignment for most keys. One option is to create a new topic, republish keyed messages to the new topic and have our consumers read from the new, rather than the older topic. Listing 7.4 shows the alter command that we would use to update an existing topic to 3 (from 2) partitions.

Listing 7.4 Adding more partitions to an existing topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --alter \    ❶
  --topic kinaction_topic --partitions 3      ❷
```

❶ Alter is needed since we are updating an existing topic.

❷ Our partition count will be up from 2 to 3 total partitions.

This command used to provide a notice that reminded us how keyed messages could be impacted by this change! More recent versions do not seem to show that message as part of the output anymore. However, it is always a good to be reminded of the impact we may have on our clients.

```
[~/kafka $ bin/kafka-topics.sh --bootstrap-server localhost:9092 \
[>   --alter --topic kinaction_topic --partitions 3
[~/kafka $ bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic kinaction_topic
Topic: kinaction_topic  PartitionCount: 3       ReplicationFactor: 2    Configs: segment.bytes=1073741824
[       Topic: kinaction_topic  Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
[       Topic: kinaction_topic  Partition: 1    Leader: 0       Replicas: 0,2   Isr: 0,2
        Topic: kinaction_topic  Partition: 2    Leader: 0       Replicas: 0,2   Isr: 0,2
```

Figure 7.4 Adding Another Partition With Alter

Once the alter is complete, running a describe command against our topic should show the PartitionCount field value as 3 and Partitions 0, 1, and 2 listed, as in Figure 7.4.

7.1.2 Replication Factors

For practical purposes, we should plan on having less than or equal to the number of replicas than the total number of brokers. In fact, attempting to create a topic with the number of replicas being greater than the total number of brokers will result in an error. It is interesting to imagine the failure case if not. Let's say we only have two brokers and we want three replicas of a partition. One of those replicas would exist on one broker and two on the other broker. In this case, if we lost the broker that was hosting two of the replicas, we would be down to only one copy of the data. Losing multiple replicas of your data at once is not the ideal way to provide recovery in the face of failure.

7.2 Partitions

Moving on from dealing with Kafka commands at a mostly topic level, let's start to look deeper at partitions. From a consumer standpoint, each partition is an immutable log of messages. It should only grow and append messages to our data store. While this data does not grow indefinitely in practice, thinking of the data being added to, rather than modified in place, is a good mental model to maintain. Consumer clients can not directly delete messages. This is what makes it possible to replay messages from a topic, which is a feature that can help us in many scenarios.

7.2.1 Partition Location

One thing that might be helpful is to look at how the data is stored on our brokers. To start, let's peek at the configuration value for `log.dirs`. Under that directory, we should be able to see a subfolder that is named with a topic name and a partition number. If we pick one of those folders and look inside, we will see a couple of different files with the extensions:

- `.index`
- `.log`
- `.timeindex`

Figure 7.5 shows how a single partition (in this case 1) in our test topic looks by doing a directory listing.

```
~/kafka $ ls /tmp/kafka-logs-0/kinaction_topic-1/
00000000000000000000.index      00000000000000000000.log        00000000000000000000.timeindex  leader-epoch-checkpoint
```

Figure 7.5 Partition Directory Listing

Sharp-eyed readers might see the files named `leader-epoch-checkpoint` and maybe even those with a .snapshot extension (not shown). In general, `leader-epoch-checkpoint` is related to leader failures and replicas. The snapshot files are files that are internal Kafka snapshots that we will not spend time looking at in-depth [99]. The files with the `log` extension are where our data payload will be stored. Other important information in this file is the offset of the message as well as the `CreateTime` field. Why the need for any other files then? Since Kafka is built for

speed, it uses the `index` file to store a mapping between the logical message offset and a physical position inside the `index` file. Having the index for Kafka to find a message quickly and then directly read from a position in the log file makes it easy to see how Kafka can still serve consumers looking for specific offsets quickly. The `timeindex` file works similarly, however, using a timestamp rather than an offset value.

One thing that we have not talked about before is that partitions are made up of segments. In essence, this means that on a physical disk, a partition is not one single file, but rather split into segments [100]. Segments are a key part of helping messages be removed based on retention policies. Figure 7.6 shows how multiple segments might make up a partition.

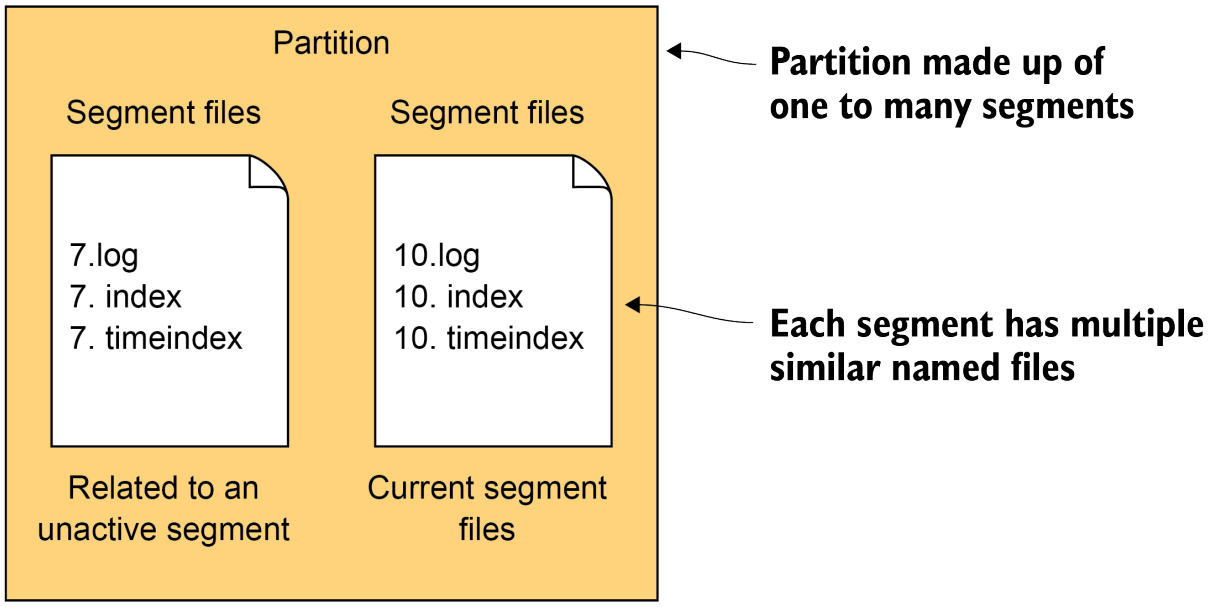

Figure 7.6 Segments Make up a Partition

An active segment will be the file to which new messages are currently being written. In our illustration, 10.log is where messages are being written in the partition directory. Older segments will be managed by Kafka in various ways that the active segment will not be: this includes being managed for retention based on the size of the messages or time configuration. These older segments (like 7.log) can be eligible for topic compaction which we will touch on later in this chapter.

To recap what we now know about segments, we know why we might have multiple files with the same name, but with `.index`, `.timeindex`, and `.log` extensions in a partition directory. For example, if we have four segments we should have a set of 4 files each with one of the previous three extensions for a total of twelve files. If we only see one of each file extension, we only have 1 segment.

7.2.2 Viewing Segments

Let's try to take a peek at a log file to see the messages we have produced to the topic so far. If we open it in a text editor, we will not see those messages in a human-readable format. Kafka includes a script, in the /bin directory, called `kafka-dump-log.sh` which we can leverage to look at those log segments. Listing 7.5 shows us passing the `print-data-log` parameter to look at a segment log file for partition 1 of the topic named 'kinaction_test' [101].

Listing 7.5 Looking at a dump of a log segment

```
bin/kafka-dump-log.sh --print-data-log \     ❶
  --files /tmp/kafka-logs-0/kinaction_test-1/00000000000000000000.log
 ❷
```

❶ Using `print-data-log` we are looking to print out the data that can not be viewed easily with a normal text editor

❷ You pass a file or comma-separated list of files to read

By using the `--files` option, which is required, we chose to look at one segment file: 00000000000000000000.log. Assuming the command is successful, we should see a list of messages with offsets as well as other related metadata like compression codecs used. Definitely, an interesting way to see how Kafka is placing messages on the broker and the data it retains around those messages.

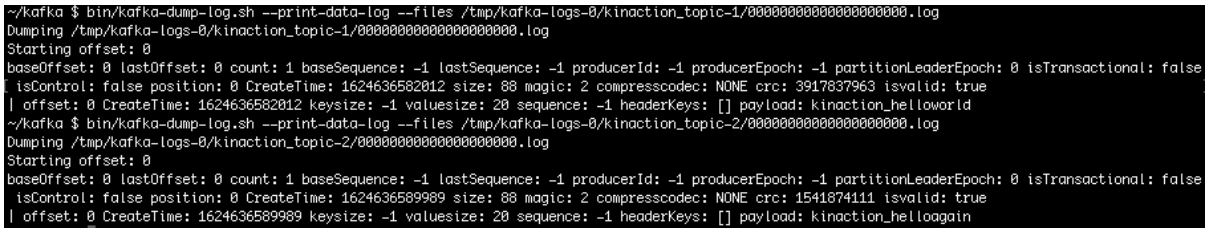

Figure 7.7 Viewing Log Segment

Looking at Figure 7.7, we can see the payloads in text that is a little easier to read than when we tried to `cat` the log file directly. For example, we can see messages across two segment files with the payloads 'kinaction_helloworld' and 'kinaction_helloagain'. Hopefully, you will have more valuable data!

As for the large number name of the log file, it is not random. The segment name should be the same as the first offset in that file. In our examples, since we are just starting with new topics, we will more than likely see the segments as all zeros.

One of the impacts of being able to see this data is that we now have to be concerned with who can see it. Do we have ways to limit access to the message brokers to only certain developers so that not just anyone can read the log files directly, i.e with tools such as `cat`? Do we require that your data-at-rest be encrypted? Since data security and access controls are common concerns

with most data that holds value, we will look at ways you can secure Kafka and topics in Chapter 10.

Of course, facts about the segment log and index files are internal Kafka broker details that we would not normally rely on in our applications. However, knowing how to look at these logs might be helpful when reasoning about our data or troubleshooting a problem.

7.3 More Topic and Partition Maintenance

It helps to imagine Kafka as a living and complex system (it is distributed after-all) that might need some care and feeding from time to time. In this section, we are going to tackle removing topics and preferred replica elections.

7.3.1 Removing a Topic

While we usually focus on not removing data from production settings, as we go along and start our explorations of topics we might make some mistakes. It's good to know that we can indeed remove a topic. All the messages of a topic will be removed and this is not something we would do unless we're ready to get rid of that data for good. Listing 7.6 shows how to use the `kafka-topic` command we have used before, but this time to delete a topic named `kinaction_topic` [102].

Listing 7.6 Deleting a topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --delete --topic kinaction_topic    ❶
```

❶ The delete option is passed to our Kafka topics command in order to mark the topic for deletion.

After running this command we will likely see a note that the topic is marked for deletion.

7.3.2 Reelect Leaders

As a reminder from our partition discussion earlier, each partition has a leader that handles all client-related data needs. The other replicas are only followers that do not talk to clients. When topics are first created, the cluster tries to spread these leaders evenly among the brokers to split some of the leaders' workload evenly [103]. The 'preferred' leader will not change for the topic as it will always be considered the replica that is first in the ISR list even across broker failures. The script `kafka-leader-election.sh` can be used in this scenario to help bring leadership back to replicas as they existed on the creation of a topic [104]. Listing 7.7 shows a simple command syntax that will run against one topic and partition.

Listing 7.7 Electing a Preferred Leader

```
bin/kafka-leader-election.sh --bootstrap-server localhost:9092    \
  --election-type preferred --partition 1 \   ❶
  --topic kinaction_test   ❷
```

❶ Only partition 1 will be impacted

❷ The topic `kinaction_test` will be used for the partition given

With a large cluster and many partitions to re-elect, we could also provide a JSON file to pass the `kafka-leader-election.sh` command called `path-to-json-file`. This file will contain a map of each topic to the partition that should undergo election. Listing 7.8 shows an example of a JSON file that will help us limit the scope of changes we wish to make.

Listing 7.8 Example Preferred Leader JSON

```
 {"partitions":
       [{"topic": "kinaction_test", "partition": 3},     ❶
        {"topic": "kinaction_test_1", "partition": 3 }
       ]
}
```

❶ Using this file with the path-to-json-file parameter can help us limit which topics and partitions we want to impact

To see if this command succeeded we can describe the topic and look for the partition leader broker id to match the first broker id in the list labeled `Replicas`.

7.3.3 EmbeddedKafkaCluster

With all of the options we have changed with partitions, it might be nice to test them as well. What if we could spin up a Kafka cluster without having a real production-ready cluster handy? Kafka Streams provides an integration utility class called `EmbeddedKafkaCluster` [105] that serves as middle ground between mock objects and a full-blown cluster. This class provides an in-memory Kafka cluster. While built with Kafka Streams in mind, we can leverage it in our testing of Kafka clients. An example below is setup like the tests found in Kafka Streams in Action by William P. Bejeck Jr. [106]. That title and his following book, Event Streaming with Kafka Streams and ksqlDB, show more in-depth testing examples that we recommend checking out including his suggestion of using Testcontainers. Listing 7.9 shows testing with `EmbeddedKafkaCluster` and JUnit 4.

Listing 7.9 Testing with EmbeddedKafkaCluster

```
@ClassRule
public static final EmbeddedKafkaCluster embeddedKafkaCluster
    = new EmbeddedKafkaCluster(BROKER_NUMBER);   ❶

private Properties producerConfig;
private Properties consumerConfig;

@Before
public void setUpBeforeClass() throws Exception {
    embeddedKafkaCluster.createTopic(TOPIC, PARTITION_NUMBER, REPLICATION_NUMBER);
    producerConfig = TestUtils.producerConfig(embeddedKafkaCluster.bootstrapServers(),
            AlertKeySerde.class,
            StringSerializer.class);   ❷

    consumerConfig = TestUtils.consumerConfig(embeddedKafkaCluster.bootstrapServers(),
            AlertKeySerde.class,
            StringDeserializer.class);   ❸
}

@Test
public void testAlertPartitioner() throws InterruptedException {
    AlertProducer alertProducer =  new AlertProducer();
    try {
        alertProducer.sendMessage(producerConfig);   ❹
    } catch (Exception ex) {
        fail("Made producer call with EmbeddedKafkaCluster should not throw exception"
        <linearrow /> + ex.getMessage());
    }

    AlertConsumer alertConsumer = new AlertConsumer();
    ConsumerRecords<Alert, String> records = alertConsumer.getAlertMessages(consumerConfig);
    TopicPartition partition = new TopicPartition(TOPIC, 0);
    List<ConsumerRecord<Alert, String>> results = records.records(partition);
    assertEquals(0, results.get(0).partition());   ❺
}
```

❶ A JUnit specific annotation is used to help create the cluster with a specific number of brokers

❷ The producer configuration needs to be set to point to the embedded cluster brokers

❸ The consumer configuration also needs to point to the embedded cluster brokers

❹ The client is called without any changes - clueless of the underlying cluster being embedded or not

❺ A Test asserts that the embedded cluster handled the message from production to consumption

When testing with `EmbeddedKafkaCluster`, one of the most important parts of the setup is to make sure that the embedded cluster is started before the actual testing begins. Since this cluster is temporary, another key point is to make sure that the producer and consumer clients know how to point to this in-memory cluster. To discover those endpoints, the method `bootstrapServers()` can be used to provide the configuration needed to the clients. Injecting that configuration into the client instances is again up to the configuration strategy but can be as simple as setting the values with a method call. Besides these configuration, the clients should be

able to test away without the need to provide mock Kafka features! The test in Listing 7.9 verifies that the `AlertLevelPartitioner` logic was correct. Using that custom partitioner logic with a critical message should have landed the alert on partition 0. By retrieving the messages for `TopicPartition(TOPIC, 0)` and looking at the included messages, the message partition location was confirmed in this test. Overall, this level of testing is usually considered integration testing and moves you beyond just a single component under test. At this point, we have tested our client logic together with a Kafka cluster - the integration of more than one module.

TIP	**POM Dependency Updates**
	Make sure that you reference the `pom.xml` changes in the source code for chapter 7. There are various JARs that were not needed in previous chapters. Also, some JARs are only included with specific classifiers noting that they are only needed for 'test' scenarios.

7.3.4 Kafka Testcontainers

If you find that you are having to create and tear-down your infrastructure, one option that could be used (especially for integration testing) is to use Testcontainers (www.testcontainers.org/modules/kafka/). This Java library leverages Docker and one of a variety of JVM testing frameworks like JUnit. Testcontainers depends on Docker images running to provide your running cluster. If your workflow is Docker-based or a development technique your team uses well, Testcontainers is worth looking into for having a Kafka cluster setup for testing.

7.4 Topic Compaction

Now that we have a solid foundation on topics being made up of partitions and partitions being made up of segments, it is time to talk about the details of log compaction.

To make it clear, topic compaction is different from the retention of existing segment log files when compared to a regular topic. With compaction, the goal is not to expire messages but rather to make sure that the latest value for a key exists and not maintain any previous state. As just referenced, compaction depends on a key being part of the messages and not being null [107]. Otherwise, the whole concept fails if there is no 'key' to update a value for in practice.

`cleanup.policy=compact` is the topic level configuration option we use to create a compacted topic [108]. This is different from the default configuration value that was set to 'delete' before our override. Listing 7.10 shows adding the config option needed for this new compacted topic.

Listing 7.10 Creating a compacted topic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create \    ❶
  --topic kinaction_compact --replication-factor 3 --partitions 3 \
  --config cleanup.policy=compact                                   ❷
```

❶ We create the topic like any other topic.

❷ Setting the `cleanup.policy` config parameter makes this topic compacted.

One of the easiest comparisons for how a compacted topic presents data can be seen in how a database table updates an existing field rather than appending more data. Let's say that we want to keep a current membership status for an online membership. A user can only be in one state at a time, either a 'Basic' or 'Gold' membership. At first, the user enrolled for the 'Basic' plan but over time ungraded to the 'Gold' plan for more features. While this is still an event that Kafka stores, in our case we only want the most recent membership level for a specific customer (our key). Figure 7.8 shows an example using 3 customers.

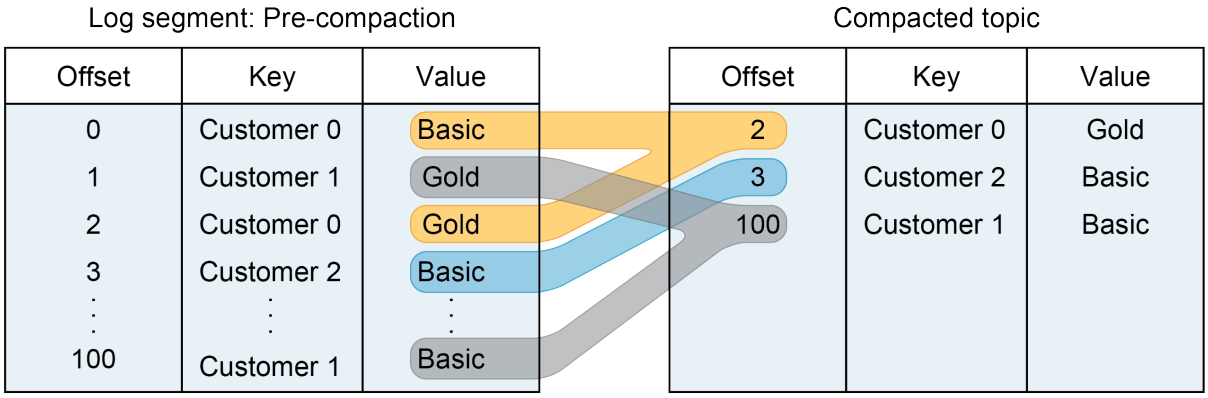

Figure 7.8 Compaction in General

After compaction is done on the example topic, the latest Customer 0 update is all that exists in the topic. A message with offset 2 replaced the old value of 'Basic' (message offset 0) for Customer 0 with 'Gold'. Customer 1 has a current value of 'Basic' since the latest key specific offset of 100 updated the previous offset 1 'Gold' state. Since Customer 2 only had one event, that event carried over to the compacted topic without any changes. Another interesting thing to consider is that compaction can appear to cause missing offsets. In effect, this is the end result of messages with the same key replacing an earlier message.

Another real-world example of why one would want to use a compacted topic is Kafka's internal topic __consumer_offsets. Kafka does not need a history of offsets that a consumer group has consumed, it just needs the latest offset. By storing the offsets in a compacted topic, the log in effect gets an updated view of the current state of its world.

7.4.1 Compaction Cleaning

When a topic is marked for compaction, a single log can be seen in a couple of different states: clean or dirty. Clean is the term for the messages that have been through compaction before. Duplicate values for each key should have been reduced to just one value. The dirty messages are those that have not yet been through compaction. Multiple values might still exist for this message for a specific key until all messages are cleaned. Figure 7.9 shows how a pointer is used to know which messages have been processed with compaction and which messages have yet to be visited [109].

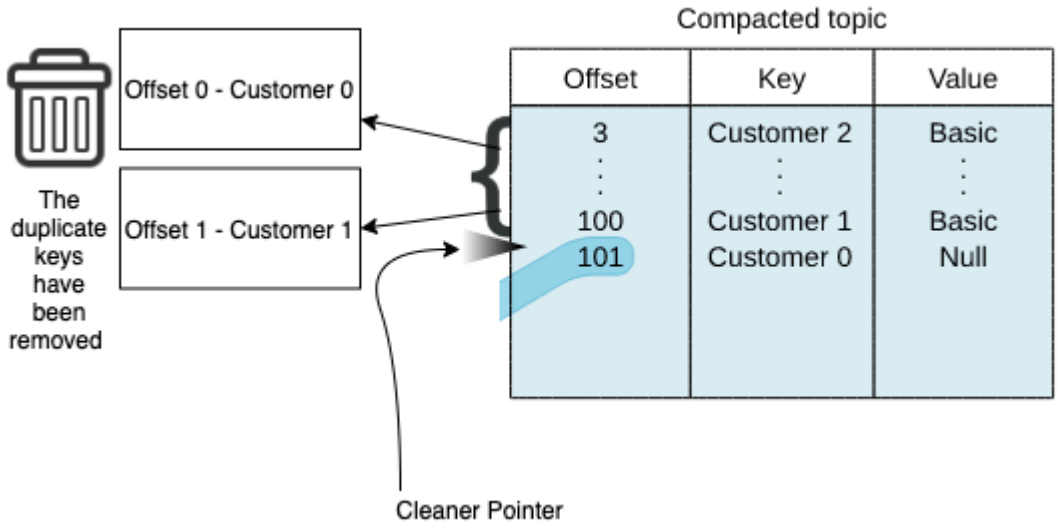

Figure 7.9 Compaction Cleaning

Looking closely at the offsets in this example, we can see that there are gaps in the cleaned segment offset numbers. Since duplicate key messages are left with the latest only, we might have certain offset numbers removed from the segment file. In the dirty sections, we will likely see the ever-increasing offset numbers we are used to, without random jumping numbers. As a reminder, this clean and dirty processing only occurs on log segments that are not the active segment. Messages that are newly arriving will not be cleaned until the segment file that they belong to is not the active log anymore.

7.4.2 Compaction Record Removal

Even though we have the notion of an immutable stream of events, there is a time when updating a value for a customer might include a removal of that value. Following our subscriber level scenario above, what if the customer decides to remove their account? By sending an event with the customer key and a null message value, Kafka will keep this last message for a certain time period. The message is considered a 'tombstone'. If you have used other systems like HBase the notion is very similar. Besides being a part of an immutable log, why would Kafka still serve this message to consumer clients? One reason is that our consumers might want to know about this final state [110]. In another scenario, the billing subscription team might want to know that a customer will not be billed further due to their current state (of removing their subscription). Figure 7.10 shows that the null value does not remove a message immediately but is served as any other message.

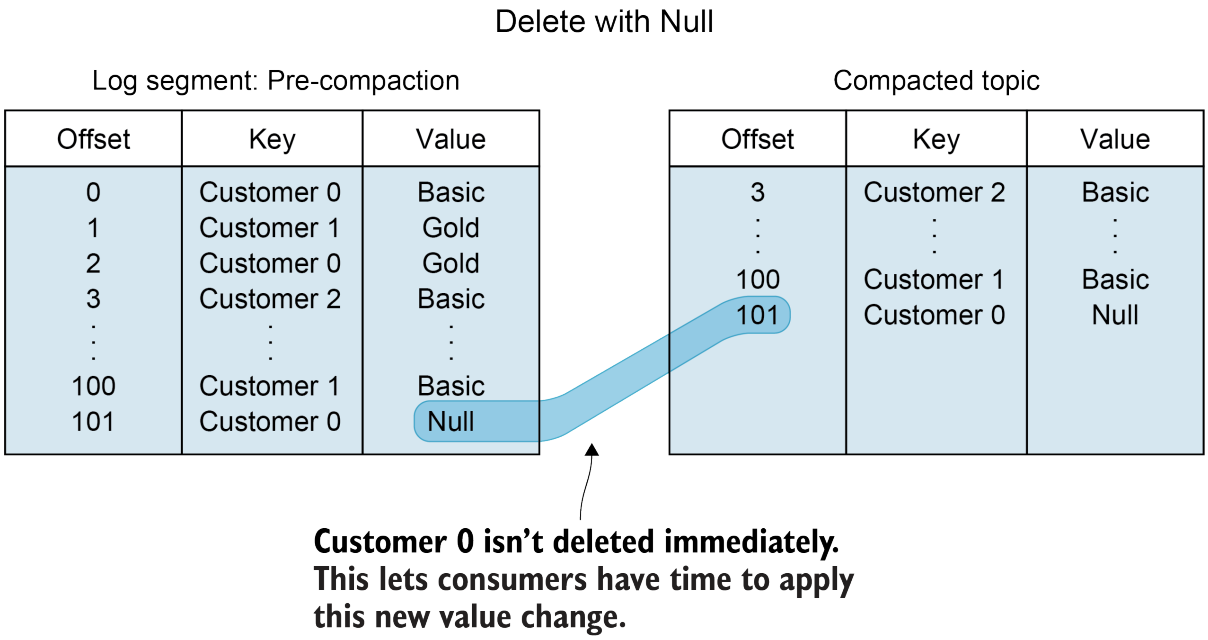

Figure 7.10 Compaction For a Deleted Value

As an one example, regulations or laws might be a valid reason that we might want to make sure that some of our topics are setup for compaction depending on your customer's location. While these are external rules that an application may or may not have to deal with: Kafka can help us fulfill those data requirements with its core feature set. Our previous discussion of segments can help us further understand when the data is really removed from the disk. Compaction only runs on inactive segments, not with active segments. In other words, those segments being currently written to are not subject to the update fashion of compactions. This also means that client consumers can potentially see more than one message from a specific key! One contributing factor to how often data is enabled for specific processing is when that log segment is rolled. If a

topic does not get a lot of data, we might want to force the rotation based on a period of time. `log.roll.hours=24` is a configuration option we have used to have segments rotated about once a day [111].

Throughout this chapter, we have looked at the various details of topics, partitions, and segments. While broker specific, they can indeed impact our clients. Since we have experience now with how Kafka stores some of its own data, we are going to spend some time in our next chapter discussing how we can store our data. This will include longer-term storage options for data.

7.5 Summary

- Topics are a logical concept rather than a physical structure. To understand the behavior of the topic, a consumer of that topic needs to know about the number of partitions, and the replication factors in play as well.
- The replication factor determines how many copies (of each partition) you want to have in case of failures.
- Partitions makeup topics and are the basic unit for parallel processing of data inside a topic. Log-file segments are written in partition directories and are managed by the broker.
- Reelecting leaders and deleting topics are maintenance that need to be completed with thought about how clients could be impacted by these changes.
- Integration testing can be used to test certain parition logic.
- Topic compaction is a way to provide a view of the latest value of a specific record. Remember, topic compaction is different from the expiration process of data.

8 Kafka storage

This chapters covers

- Inspecting requirements to determine how long to retain data.
- Investigating tools and options to help with data movement into and out of Kafka.
- Considering the data architectures Kafka enables such as Lambda and Kappa.
- Exploring storage choices for cloud and containers instances.

So far we have thought of our data as moving into and out of Kafka for brief periods of time. Another decision to analyze is where our data should live longer term. When you think of databases like MySQL or MongoDB, you may not always think about if or how that data is expiring. Rather, you know that the data is likely going to exist for the majority of your application's entire lifetime. In comparison, Kafka storage logically sits somewhere between the long-term storage solutions of a database and the transient storage of a message broker; especially if we think of message brokers holding onto messages until they are consumed by a client as it is often in other message brokers.

Let's look at a couple of options for storing and moving data in our environment.

8.1 How Long to Store Data

Currently, the default retention for data in Kafka topics is seven days but this can be easily configured by time or data size. However, can Kafka hold data itself for a period of years? One real-world example is how the New York Times uses Kafka [112]. The content in their cluster is in a single partition that was less than 100GB at the time of writing. If you recall from our discussion in Chapter 7 about partitions you know that all of this data will exist on a single broker drive (as will any replica copies exist on their own drives) as partitions are not split between brokers. Since storage is considered to be relatively cheap and the capacity of modern hard drives is way beyond hundreds of gigabytes, most companies would not have any size issues with keeping that data around. Is this a valid use of Kafka or an abuse of its intended purpose and design? As long as you have the space for your planned growth on a disk for the future, you might have found a good usage pattern for your specific workload!

How do we configure retention for brokers? The main considerations are the size of the logs and the length of time the data exists. Table 8.1 shows some of the broker configuration options that are helpful for retention [113]:

Table 8.1 Broker Retention Configuration

Key	Purpose
`log.retention.bytes`	The largest size in bytes threshold for deleting
`log.retention.ms`	The length in milliseconds a log will be maintained before being deleted
`log.retention.minutes`	Length before deletion in minutes. log.retention.ms is used if both are set
`log.retention.hours`	Length before deletion in hours. log.retention.ms and log.retention.minutes would be used before this value if either of those were set

How do we disable log retention limits and allow them to stay 'forever'? By setting both `log.retention.bytes` and `log.retention.ms` to -1 we can effectively turn off data deletion [114].

Another thing to consider is how we **may** get similar retention for the latest values by using keyed events with a compacted topic. While data can still be removed during compaction cleaning, the most recent keyed messages will always be in the log. This is a good way to retain data in use cases where we do not need every event (or history) of how a key changed state to the current value.

What if we want our data to stick around for a while, but simply do not have the disk space to hold our data on brokers? Another option for long-term storage is to move the data outside of Kafka and not retain it internally to the Kafka brokers themselves. Before data would be

removed by retention from Kafka, we could store the data into a database, HDFS, or upload our event messages into something like an S3 bucket in Amazon Web Services (AWS). All of these paths are valid options and provide more cost effective means of holding onto our data after our consumers have processed it.

8.2 Data Movement

Almost all companies seem to have a need for transforming the data that they receive. Sometimes it is specific to an area within the company or due to third-party integrations. A popular term that many people use in this data transformation space is extract, transform, load (ETL). Graphical tooling or code can be used to take data in its original format, transform the data, and then place it into a different table or data store. Kafka can play a key role in these data pipelines.

8.2.1 Keeping the original event

One thing that I would like to note is my preference for event formats inside of Kafka. While open for debate and your use-case requirements, my preference is to store messages in the original format in a topic. Why keep the original message and not format it right away before placing it into a topic? For one, having the original message makes it easier to go back and start over if you messed up your transform logic. Instead of having to try to figure out how to fix our mistake on the altered data, you can always just go back to the original data and start again. I know that usually most of us will have that experience when trying to format a date. Or the first time you run a regular expression! Sometimes you need a couple of shots at making the data the format you want. Another pro of getting the entire message is the fact that data you don't use today might be used in the future. Let's say the year is 1995 and you are getting a field from a vendor called 'email'. Your business will never need that field right? Once you see the need to launch your first email marketing campaign, you'll be thanking your past self that you kept that original, "useless" data. While the email field might be a trivial example for some, it is interesting to think about the usages in the data analysis space. What if your models start to see trends on data that you once thought wouldn't matter. By having all of the data fields, you might be able to go back to that data and find insights you never expected.

8.2.2 Moving away from a batch mindset

Does the general topic of ETL or data pipelines bring terms to mind such as batch, end of day, monthly, or even yearly? One of the shifts from data transformation processes of the past is the idea that you can continuously stream your data into various systems without delay. With Kafka, for example, you can keep the pipeline running in near real-time and you can leverage its stream processing platform to treat your data as an infinite series of events. I mention this as a reminder that Kafka can help enable a shift in the way you think of your data altogether. You do not have to wait for a nightly job to run and update a database. You also do not have to wait for a nightly window with less traffic to do intensive ETL tasks, you can do them as they stream into your system and have pipelines that are constantly working for your applications in real-time. Let's take a look at some of the tools that have been developed in this space that might help you leverage your pipelines in the future.

8.3 Tools

Data movement is a key to many systems, Kafka included. While you can stay inside the open-source Kafka and Confluent offerings like Connect, which was discussed in chapter 3, there are other tools that might fit your infrastructure or are already available in your tool suite. Depending on your specific data source or sinks, these options below might help you achieve your goals. Note that why some tools include sample commands, more setup would be required before we can run these commands on our local machines.

8.3.1 Apache Flume

If you were first introduced to Kafka through work in the Big Data space, it is a strong possibility that you might have used Flume in relation to your cluster. If you have ever heard the term "Flafka", you have definitely used this Kafka and Flume integration. Flume can provide an easier path for getting data into a cluster and relies more on configuration than custom code. If you wanted to ingest data into your Hadoop cluster and already have support from a vendor like Cloudera on these various pieces, Flume is a solid option to get data into your Kafka cluster.

Figure 8.1 shows an example of how a Flume agent is running on a node as its own process. It watches the files local to that server and then uses the configuration for the agent that you provided to send data to a sink destination.

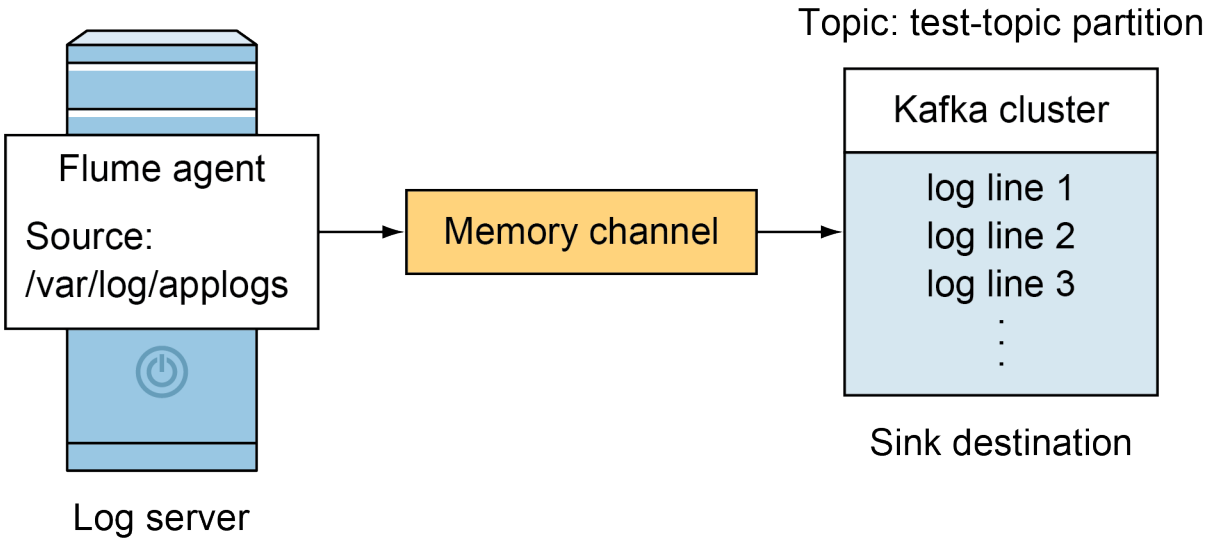

Figure 8.1 Flume Agent

Let's take a look again at integrating log files (our source of data) using a Flume agent into a Kafka topic (our data sink). Listing 8.1 shows a sample configuration file that would be used to set up a local Flume agent to watch a directory for changes [115]. The changes would be placed in a Kafka topic titled `kinaction_test_topic`. To imagine this example with a comparison, it is like using a `cat` command on a file in a directory to read a file and sending the result to a specific Kafka topic.

Listing 8.1 Flume configuration for watching a directory

```
ag.sources = logdir                                    ❶
ag.sinks = kafkasink
ag.channels = c1

#Configure the source directory to watch
ag.sources.logdir.type = spooldir                      ❷
ag.sources.logdir.spoolDir = /var/log/applogs
ag.sources.logdir.fileHeader = true

# Describe the Kafka sink
ag.sinks.kafkasink.channel = c1                        ❸
ag.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
ag.sinks.kafkasink.kafka.topic = kinaction_test_topic
ag.sinks.kafkasink.kafka.bootstrap.servers = localhost:9092,localhost:9093,localhost:9094
ag.sinks.kafkasink.kafka.flumeBatchSize = 10
ag.sinks.kafkasink.kafka.producer.acks = 1
ag.sinks.kafkasink.kafka.producer.linger.ms = 5
ag.sinks.kafkasink.kafka.producer.compression.type = snappy

# Memory channel configuration
ag.channels.c1.type = memory                           ❹
ag.channels.c1.capacity = 1000
ag.channels.c1.transactionCapacity = 100

# Bind both the sink and source to the same channel
ag.sources.logdir.channels = c1                        ❺
ag.sinks.kafkasink.channel = c1
```

❶ We can define your custom names for the source, sink, and channel.

❷ We are using the `spooldir` specific source type configuration that is needed to let Flume know which directory to watch for log entries.

❸ This section helps define our topic and Kafka cluster information where we want our data to end up.

❹ A channel is used as a transport method to get data from the source to the sink.

❺ Like a pipe in Unix, the source is attached to the sink by the channel we defined in our configuration.

Listing 8.1 shows how we could configure a Flume agent running on a server. At this point in your Kafka journey, you should notice the sink configuration looking a lot like the properties we have used before in our Java client producer code. It is important to note those properties, in the example above, with the prefix `kafka.producer`. The Flume agent will use those prefixed values when creating its internal Kafka producer client so we can use those other options if needed. On the flip side, if we are using Kafka as our source, we can use the prefix `kafka.consumer` to use any valid consumer property as well.

It is also interesting to note that Flume can use Kafka as not only a source or as a sink, but also as a channel. Since Kafka is seen as a more reliable channel for events, Flume can leverage Kafka to deliver messages between various sources and sinks. In our example for Listing 8.1, we are using a memory channel that could lose messages if the agent failed and only contained that information in RAM. Using a Kafka channel might be a better idea for higher availability. If you are reviewing Flume configuration and see Kafka mentioned, be sure to notice where and how it is actually used.

Listing 8.2 Flume Kafka Channel configuration

```
ag.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel                ❶
ag.channels.channel1.kafka.bootstrap.servers =
 <linearrow /> localhost:9092,localhost:9093,localhost:9094   ❷
ag.channels.channel1.kafka.topic = channel1-kinaction_test                              ❸
ag.channels.channel1.kafka.consumer.group.id = flume-cgroup                             ❹
```

❶ The Kafka channel type used in Flume is the built-in class KafkaChannel

❷ As with any other Kafka cluster configuration, we provide our servers to connect to

❸ The topic that will hold the data between source and sink

❹ We can provide a consumer group to avoid collisions with your other consumers

Listing 8.2 shows the Flume agent configuration we could use to provide a reliable channel between various sources and sinks that Flume supports.

8.3.2 Debezium

Debezium (debezium.io) describes itself as a distributed platform that helps turn databases into event streams. In other words, updates to our database can be treated as events! If you have a database background (or not!) you may have heard of the term change data capture (CDC). As the name implies, the data changes can be tracked and used to react to that change. At the time of writing this chapter, MySQL, MongoDB, and PostgresSQL servers were supported with more systems slated to be added. Debezium uses connectors and Kafka Connect to record the events and our application would consume from Kafka as a normal client. Figure 8.2 shows an example of Debezium when it is registered as a connector in regards to Kafka Connect.

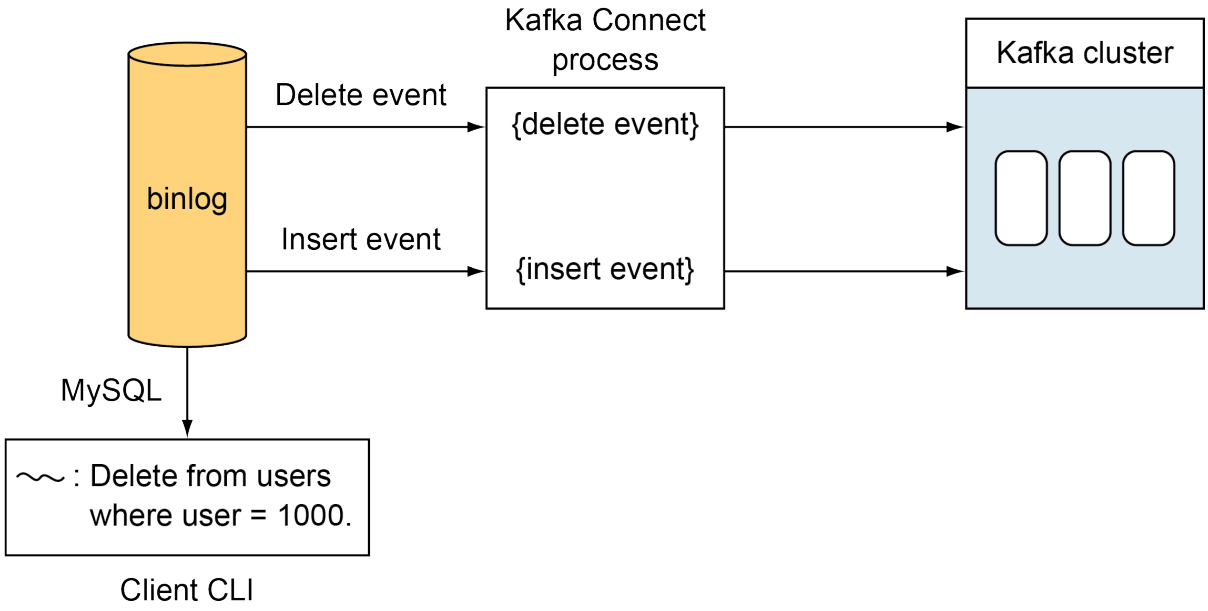

Figure 8.2 Kafka Connect and Debezium used with a MySQL Database

In our scenario, a developer was using a command-line interface (CLI) and deleted a user against the MySQL database instance that was being monitored for changes. Debezium would capture that event being written to the database's internal log and that event would go through the connector service and feed into Kafka! If a second event such as a new user being inserted into the database, a new event would be captured.

An additional note, while not Kafka-specific, there are other examples of using techniques like CDC to provide timely events or changes to your data that might help you draw a parallel to what Debzium is aiming for overall. If you have ever used a platform like Meteor (www.meteor.com/), you can see a couple of ways data changes can be captured and acted on in your application. MongoDB has the concept of an operations log (oplog). This specific log holds a continuous record of actions that have modified the data stored in your database instances. By watching this log for changes, your application or platform can propagate these events to other parts of your logic. This allows web UI pages to be updated in near real-time without the need for a reload of the web page. In comparison, what if we did not have this log available for a

specific database? There is always the option of polling against a database. While not always the most efficient, it is a valid path to identify changes and differences between polls.

8.3.3 Secor

Secor (github.com/pinterest/secor) is an interesting project that has been around since 2014 from Pinterest that aims to help persist Kafka log data to a variety of storage options including S3 and Google Cloud Storage [116]. The options for output are also various including sequence files, ORC, Parquet files as well as other formats. As always, one major benefit of these projects having source code in a public repository is the fact that you can see how other teams have implemented requirements that might be similar to ours. Figure 8.3 shows how Secor would act as a consumer of a Kafka cluster, very much like any other application. Having a consumer being added to a cluster for data backup is not a big deal, it leverages the way Kafka has always handled multiple readers of the events!

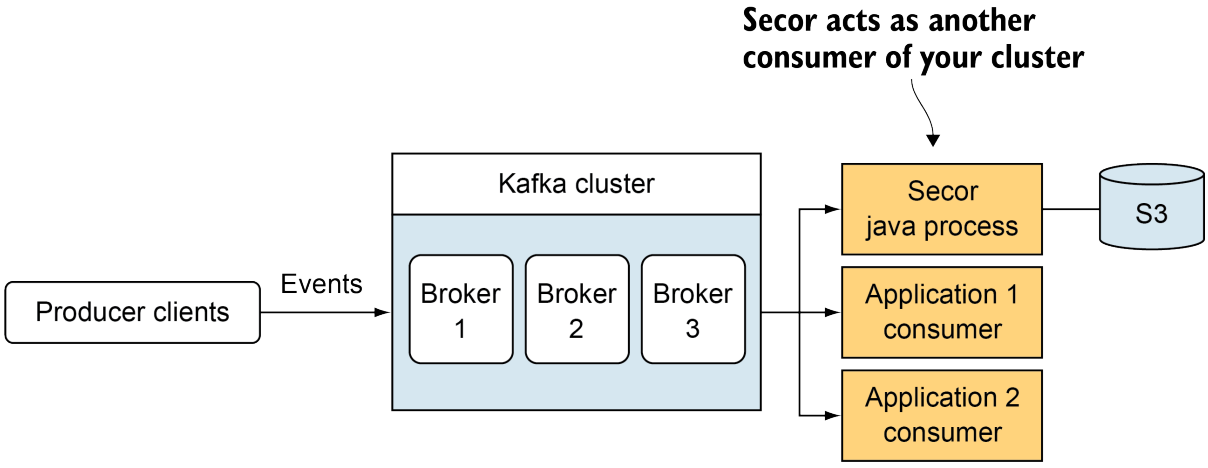

Figure 8.3 Secor acting as a consumer and placing data into S3

Secor runs as a Java process and can be fed our specific configuration. In effect, it acts as another consumer of our existing topic(s) to gather data to end up in a specific destination like an S3 bucket. Secor does not get in the way of our other consumers and allows us to have a copy of our events so that they are not lost once Kafka retention removes data from its logs.

Invoking Secor should be familiar to those who are used to working with jars in a Java environment. Arguments can be passed to the Secor application with standard '-D' parameters. The most important file to update is the configuration options which you can place in a properties file. This file is where we can fill in the details about our specific S3 bucket, for example.

8.3.4 Example Use-Case for Storage of Data

Let's think about an example of how moving data out of Kafka could be used at a later time. First, to clarify, we will break down our usage of the same data between two different areas. One area is working with the data as it comes into Kafka, in an operational manner. Operational data is the events that are produced by our day to day operations. We can think of the event to order an item from a website as an example. A 'purchase' event triggers our application into motion and does so in a low latency way. The value of this data to our real-time applications might warrant keeping the data for a couple of days until the order is complete and mailed. After this timeframe, the event may become more important for our analytical systems. Analytical data, while based on that same operational data, is usually used more to make business decisions. In traditional systems, this is where systems like a data warehouse, an Online Analytical Processing system (OLAP), and Hadoop shine. Those events can be data-mined using different combinations of fields in our events in different scenarios to find insights into sales data. If we notice that sales of cleaning supplies always spike before a holiday, we might be able to use that data to generate better sale options for our business in the future.

8.4 Bringing data back into Kafka

One of the most important things to note is that just because our data has left Kafka once, it does not mean that it can't be put back in again! Figure 8.4 shows an example of data that lived out its normal lifespan in Kafka and was archived in S3. When a new application logic change required that older data be reprocessed, we did not have to create a client to read from both S3 and Kafka. Rather, using a tool like Kafka Connect, we were able to load that data from S3 back into Kafka. The interface stayed the same from the point of view of our applications! While it might not seem obvious at first glance why we would want to do such a thing, let's consider a situation in which we find value in moving our data back into Kafka after we processed it and the retention period has passed.

Imagine a team working on trying to find patterns in data that they have collected throughout years of handling events. In our example, there are terabytes of data. To serve operational real-time data collection, this data has been moved from Kafka into HDFS after real-time consumers have dealt with the messages. So does our application logic now have to pull from HDFS directly? Why not just pull it back into Kafka and our application can process the data as it had before? Loading data into Kafka again is a valid way of reprocessing data that may have aged out of our system. Figure 8.4 shows another example of how we can move data back into Kafka.

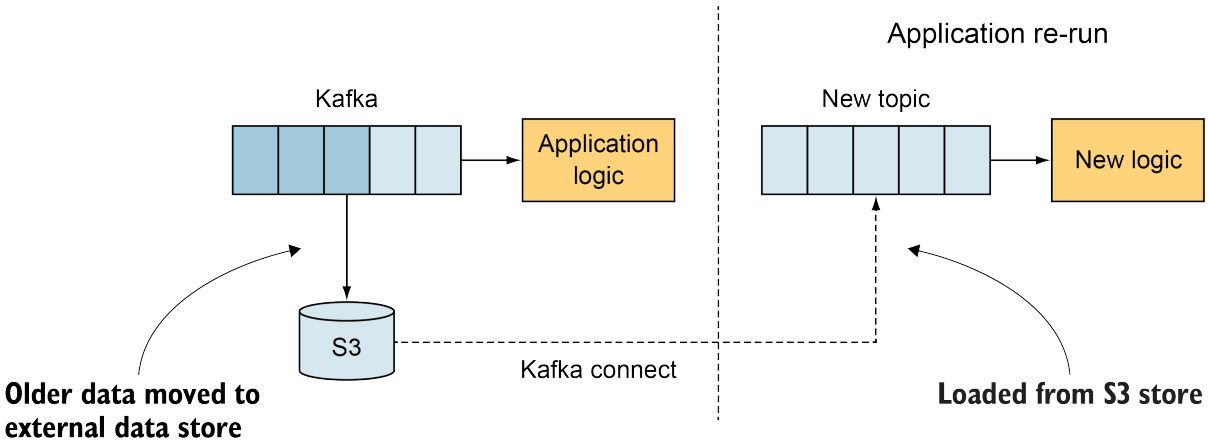

Figure 8.4 Data Back into Kafka

After some time, events are not available to the applications due to data retention configuration within Kafka. However, we have a copy of all previous events in an S3 bucket. Let's say that we have a new version of our previous application and would prefer to go through all of the previous data events as our previous application. However, since they are not in Kafka, do we pull them from S3 now? Do we want our application logic to pull from various sources or just to have one interface (that being Kafka)? For us, we can create a new topic in our existing Kafka cluster and load the data from S3 with Kafka Connect placing the data into a new Kafka topic. Our application can then run against Kafka, processing events without having to change any processing logic.

The thought process is really to keep Kafka as the interface of your application and not have to create multiple ways to pull data into processing. Why create and maintain custom code to pull from HDFS, S3, etc when you can use an existing tool like Connect or Secor to move the data to or from Kafka. Once you have your data in that one interface, you can process it the same. Keep in mind this technique only applies to data that has been removed from Kafka. If you still have the total timeline of data that you need in Kafka, you can always seek to the earlier offsets.

8.4.1 Tiered Storage

A newer option from Confluent Platform version 6.0.0 on, is called Tiered Storage. In this model, local storage is still the broker itself and remote storage is introduced for data that is older (and stored in a remote location) and controlled by time configuration (`confluent.tier.local.hotset.ms`) [117]. This remote location can be an AWS S3 bucket for example. KIP-405 (cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage#KIP405:KafkaTieredStorage-Kafkalocalstorageandoperationalcomplexity) introduces this option to Kafka itself.

8.5 Architectures with Kafka

While there are various architectural patterns to build your our products such as Model-View-Controller (MVC), Peer-to-Peer (P2P), or Service-oriented architecture (SOA) to name a few, Kafka and viewing your data as events can change the way you think about your entire architectural design. Let's take a peek at a couple of architectures that could be powered by Kafka (and to be fair other streaming platforms). This will help us get a different perspective on how we might design systems for our customers. The term 'Big Data' is used in reference to some of these discussions. It is important to note that the amount of data and the need to process that data in a timely manner were drivers that led to some of these system designs. However, these architectures are not limited to 'Fast Data' or 'Big Data' applications only. By hitting the limits of specific traditional database technologies, new views on data evolved.

8.5.1 Lambda Architecture

If you have ever researched or worked with data applications that have included needs for both batch processing and operational workloads, you might have seen references to the Lambda Architecture. The implementation of this architecture can start with Kafka as well but is a little more complex. The real-time view of the data is combined with a historical view to serve end-users. The complexity of merging these two views of the data should not be ignored. From experience, it was a challenge to rebuild the serving table. Also, you are likely going to have different interfaces for your data to maintain as you work with the results from both systems. The book "Big Data" written by Nathan Marz with James Warren discusses the Lambda Architecture more fully and goes into details about the Batch, Serving, and Speed layers [118]. Figure 8.5 shows an example of how taking customer orders can be thought of in a batch and real-time way. The customer totals from the previous days can be combined with orders happening during the day to combine into a combined data view to end-users.

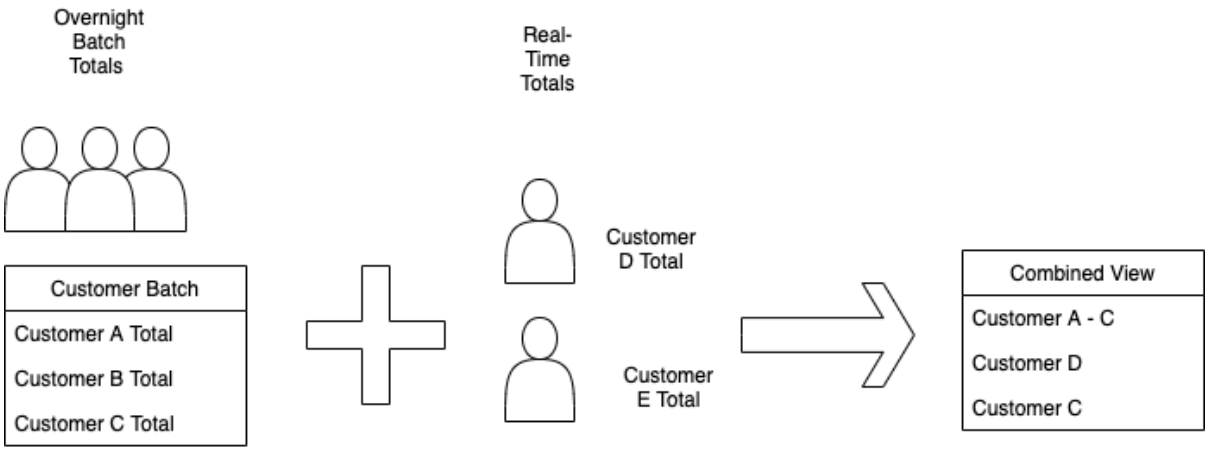

Figure 8.5 Lambda Architecture

Taking the concepts from Figure 8.5 and to get a feel for this architecture, let's look at each layer at a high level that is discussed in the "Big Data" book by Marz.

- Batch: This layer is similar to the way batch processing with Map-Reduce occurs in a system like Hadoop. As new data is added to your data stores, the batch layer can continue to pre-compute the view of the data that has already arrived in the system.
- Speed: This layer is similar in concept to the batch layer accept it produces views from recent data.
- Serving: This layer updates the views it sends to consumers after each update to the batch views.

For the end-user, the Lambda architecture unites data from the serving layer and the speed layer to answer requests with a complete view of all recent and past data. This real-time streaming layer is the most obvious place for Kafka to play a role, but it can also be used to feed the Batch layer.

8.5.2 Kappa Architecture

Another architectural pattern that can leverage the power of Kafka is the Kappa Architecture. This architecture was proposed by the co-creator of Kafka, Jay Kreps [119]. Think about wanting to maintain a system that impacts your users without disruption. One way to do this is to switch out your updated views like in Lambda. Another way to do this is by running the current system in parallel to the new one and cutting over once the new version is ready to serve traffic. Part of this cutover is of course making sure that the data that is being served by the older version will be reflected correctly in the newer version.

You only regenerate the user-facing data when you need to, there is no need to merge old and new data which is an on-going process for some Lambda implementations. It does not have to be a continuous job, but rather invoked when you need an application logic change. Also, no need to change your interface to your data, Kafka can be leveraged by both your new and old application code at the same time. Figure 8.6 shows how how customer events are used to create a view without using a batch layer.

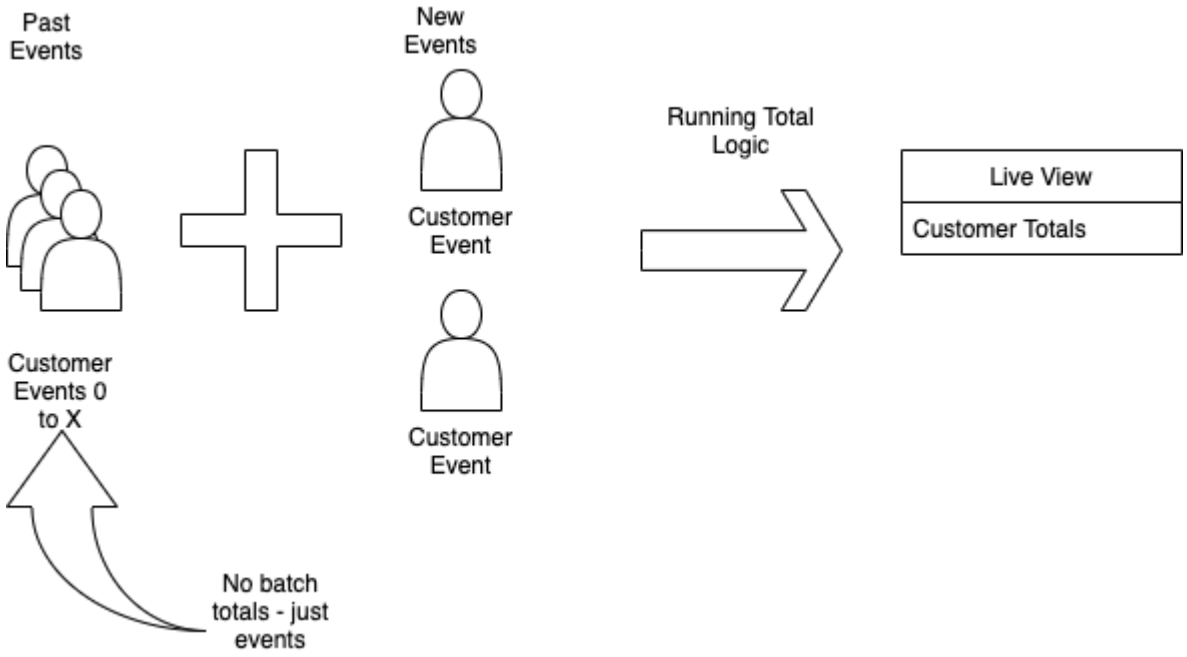

Figure 8.6 Kappa Architecture

Figure 8.6 shows customer events from the past and present being used directly to create a view. Imagine the events being sourced from Kafka and then using Kafka Streams or ksqlDB to read all the events in near real-time and creating a view for the end-users. If a change is ever needed to how the customer events are processed, a second application could be created with different logic (like a new ksqlDB query) and use the same data source (Kafka) as before. There is no need to have a batch layer (and manage it) as there is only streaming logic used for making your end-user views.

8.6 Multiple Cluster setups

Most of our topics and discussions so far have been from the viewpoint of our data in the context of one cluster. Kafka scales well and it is not unheard of to reach hundreds of brokers for a single cluster. However, a one size cluster will not fit all infrastructures. One of the concerns we run into when talking about cluster storage is where you serve your data in relation to your end-user clients. In this section, we are going to talk about scaling by adding clusters rather than by adding brokers alone.

8.6.1 Scaling by adding Clusters

Usually, the first things to scale would be the resources inside your existing cluster. The number of brokers is usually the first option and makes a straight-forward path to growth. However, Netflix's MultiCluster strategy is a captivating take on how to scale Kafka clusters [120]. Instead of using the broker number as the way to scale the cluster, they found they could scale by adding clusters themselves! They found that it might make sense for event producers to talk to one Kafka cluster and have those events moved to other Kafka clusters for consumption. In this setup, consumers would be reading messages from a Kafka cluster that is entirely different from the ingestion cluster.

As per their scenario, the amount of data going into the cluster was not a problem. However, once they had multiple consumers pulling that data out at the same time, they saw a specific bottleneck for their network bandwidth. Even with 1 or 10 Gigabit Ethernet (GbE) data center networks, the rate of data and the number of bytes out can add up quickly. Figure 8.7 shows an example of how having many consumers divided across clusters might help reduce the network impact of only one cluster being used. Note that this means each cluster would be on its own distinct network and not all in the same network or the problem would likely be worse.

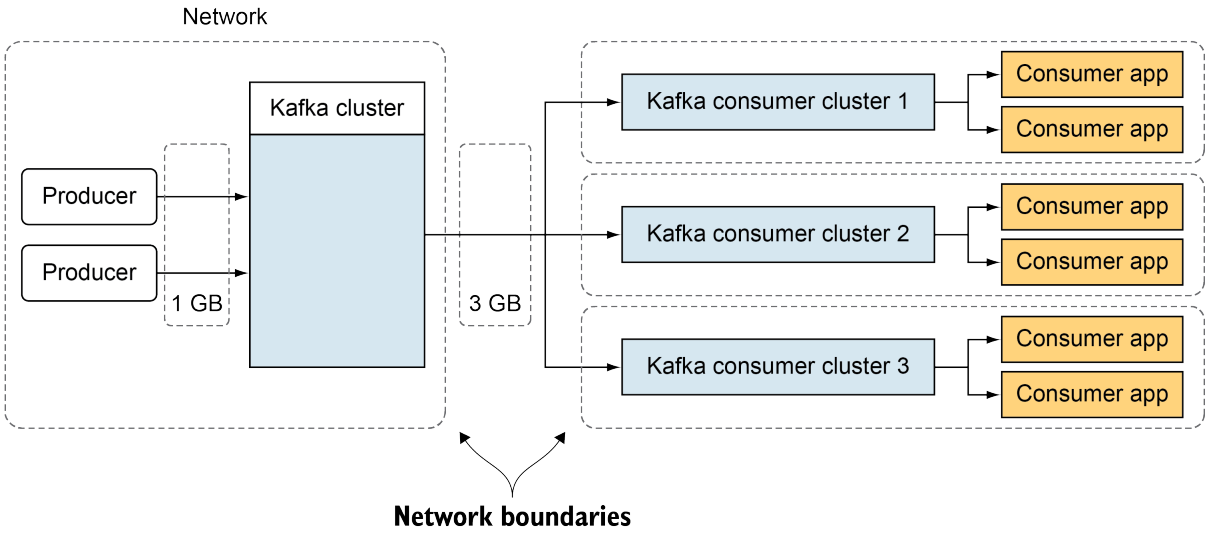

Figure 8.7 Multiple Cluster Architecture

Imagine a scenario (we'll assume generous network data amounts for ease of understanding) where we have producers that are sending about a gigabyte (GB) of data per second to one cluster. If we have six different consumers, we would be sending 6 copies of that 1 GB of data for around 7 GB per second (again let's not worry about exact numbers). We do not have much more room on a 10 GbE network. Figure 8.7 shows that if we have one cluster ingesting all of our events, we can narrow the amount of copies we send out in this specific network. The other clusters will help divide the total load.

This design brings to mind the idea of Command Query Responsibility Segregation (CQRS). For

more details on CQRS check out Martin Fowler's site at martinfowler.com/bliki/CQRS.html [121]. Specifically, the idea of separating the load of reading data from that of writing data. Each action can scale in an independent manner without limiting other actions. While CQRS is a pattern that can add complexity to our systems, it is interesting to note how this specific example helps manage the performance of a large cluster by separating the load of producers sending data into Kafka from the much larger load of consumers reading the data.

Another benefit of this setup is the ability to do upgrades or fail-over in a non-rolling update fashion. Instead of rolling broker updates, the new cluster is bought up, sent traffic from producers and/or the old cluster, and then the older cluster is removed after the new one is ready to go.

8.6.2 Data over Two Regions

Another important item to consider is the distance we have between our application's consumers and the data in our cluster. Data that is local to our applications will take much less time to load than would data pulled from the other side of the country if we only had a single cluster as our source. Let's look at one of the most common cluster setups that we have worked with: the Active-Active region setup [122].

In an Active-Active model each data-center has its own cluster. End-users of the clusters usually connect to the data center cluster that is closest to them unless there is an outage or failure scenario. The clusters have data replicated between them for both clusters to use no matter where the data originated using MirrorMaker or something similar like Confluent Replicator [123]. A service-level agreement (SLA) might also drive us to pursue this option. If we can not meet our client's needs in an agreed amount of time, we might have to change the location of the data to make that possible. Storage location does have a practical impact on our design.

8.7 Cloud and Container-Based Storage Options

While we talked about not sharing drives between our operating system and our Kafka log directories in Chapter 6, we did not address the types of instances to use in environments that provide more short-lived storage. With AWS being a popular cloud deployment option, let's focus on a specific example even though these points would be similar for persistent disk solutions on other cloud providers.

8.7.1 Amazon Elastic Block Store

In regards to AWS, some users might be tempted to use local instance storage only when planning out their cluster. Since Kafka was built to handle broker failures, the cluster more than likely would be able to recover on newly created broker instances as long as the cluster was otherwise healthy. This solution is straightforward for the most part, you start a new broker node and the cluster will work on gathering data for its partitions from the other brokers. This usually cheaper storage option might be worth the impact of the network traffic increase and any time delays for retrieving data vs other options.

However, Elastic Block Store (EBS) could be a safer option for our use-case. In case you are not familiar with the EBS service, it provides persistent block storage that helps data remain even in the case of a compute instance failure. Besides the fact that you can reassign the EBS volume to a new instance, the broker would only have to recover from messages that were missed during downtime vs the entire data partitions having to be replicated to that node. Amazon also offers instances that are optimized for writing Kafka logs: ie that are built for sequential I/O that is used in writing log files. Confluent shared a study on deplopyments with AWS considerations in which they looked at the storage type tradeoffs [124]. For example, the benchmarked performance of the st1 EBS was better than instance storage for their testing at that time.

Another concern to address is having our broker storage spread across multiple availability zones (AZs). While this does not address a region failure, a failed AZ will not cause our cluster to be unavailable.

As always, remember that Kafka itself keeps evolving and reacting to needs that users run into as daily challenges. KIP-392 shows an item that was under discussion at the time of writing that seeks to help address the issues of a Kafka cluster spanning data centers titled: 'Allow consumers to fetch from closest replica' [125]. Be sure to check out recent KIPs from time to time to see how Kafka is evolving in exciting ways!

8.7.2 Kubernetes Clusters

Dealing with a containerized environment, we might run into similar challenges as we would in the cloud. If we hit a poorly configured memory limit on our broker, we might find ourselves on an entirely new node without our data unless we have the data persisted correctly! Unless we are in a sandbox environment in which we can lose data, persistent volume claims will be needed by our brokers to make sure that our data will survive any restarts, failures, or moves. While the broker instance container might change, we should be able to claim the previous persistent volume (similar to how we can use EBS in AWS after a failure).

Kafka applications will likely use the StatefulSet API in order to maintain the identity of each broker across failures or pod moves. This static identity also helps us claim the same persistent

volumes that were used before our pod was down.

There are already Helm charts (github.com/confluentinc/cp-helm-charts) to help us get started with a test setup as we explore Kubernetes [126]. Confluent does have plans to release a Confluent Operator to help as well. The scope of Kubernetes is relatively large to cover in our discussion, but the key concerns are present regardless of our environment. Our brokers have an identity in the cluster and are tied to the data that each is related to. To keep the cluster healthy, those brokers need the ability to identify their broker managed logs across failures, restarts, or upgrades.

8.8 Summary

- Data retention should be driven by business needs. Decisions to weigh include the cost of storage and the growth rate of that data over time.
- Size and time are the basic parameters for defining how long data is retained on disk.
- Long-term storage of data outside of Kafka, like in S3, is an option for data that might need to be retained for long periods. Data can be reintroduced by producing the data into a cluster at a later time if needed.
- The ability of Kafka to handle data quickly and also replay data can enable architectures such as the Lambda and Kappa architectures.
- Cloud and containers workloads often involve short-lived broker instances. Data that needs to be persisted requires a plan for making sure newly created or recovered instances can utilize that data across instances.

9 Management: tools and logging

This chapters covers

- Exploring helpful scripting administration client options
- Examining REST APIs, tools, and utilities that can be leveraged for management
- Mananging the logs of Kafka and ZooKeeper
- Monitoring the cluster with JMX and related metrics
- Understanding how advertised listeners work with clients
- Tracing example using interceptors with headers

We have spent some time discussing brokers in depth in Chapter 6 and client concerns throughout the earlier chapters. We've seen some best practices that can be applied in most situations, but there will always be environments where special handling is required. The best way to keep a cluster moving along is to understand the data that is flowing through it, and to monitor that activity at runtime. While operating Kafka may not be as involved as writing and running Java applications per se, it still requires monitoring log files and being aware of what is happening with our workloads.

9.1 Administration clients

So far, we have performed most of our cluster management activities with the command line tools that come with Kafka. And in general, we will need to be comfortable with a shell environment to set up and install Kafka. However, there are some helpful options we can use to branch out from these provided scripts.

9.1.1 Administration in code with AdminClient

One useful tool to look at is the `AdminClient`. Although the Kafka shell scripts are great to have at hand for quick access or one-off tasks, there are situations, such as automation, where the Java AdminClient really shines. The AdminClient is in the same `kafka-clients.jar` that we used for the Producer and Consumer clients. It can be pulled into a Maven project (see the pom.xml from Chapter 2) or it can be found in the `share/` or `libs/` directory of the Kafka installation.

Let's look at how we could execute a command we have completed before, to create a new topic, but this time with the AdminClient. Listing 9.1 shows how we ran this from the command line in Chapter 2.

Listing 9.1 Creating the kinaction_selfserviceTopic Topic by command line

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \     ❶
  --create --topic kinaction_selfserviceTopic --partitions 2 \    ❷
  --replication-factor 2    ❷
```

❶ Using the `kafka-topic.sh` script to create a new topic.

❷ Including our custom integers for the number of partitions and replicas for our topic.

Though this command-line example works fine, we don't want to be called every time someone needs a new topic in our dev environment. Instead, we'll create a self-service portal that other devlopers can use to create new topics on our development cluster. The form for our application will take in the following information: a name for the topic, and the numbers for partitions and replicas. Figure 9.1 shows an example of how this application might be set up for end-users. Once the user submits the web form, the admin-client Java code runs to create a new topic.

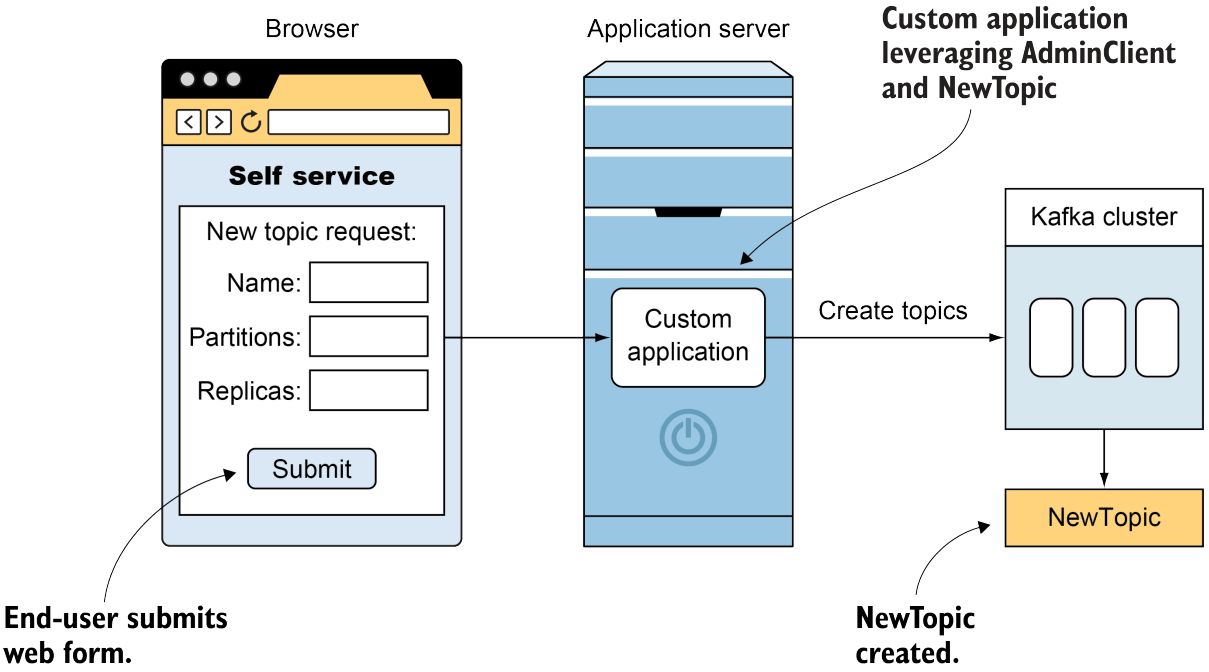

Figure 9.1 Self Service Kafka Web Application

In this example, we could add logic to make sure that naming conventions for topics fit a certain pattern if we had such a business requirement. This could be a way to maintain more control over our cluster rather than users working from the command-line tools. To start, we need to create a `NewTopic` class. The constructor for this class takes three arguments:

- topic name
- the number of partitions
- the number of replicas

Once we have this information, we can leverage the AdminClient object to complete the work. The AdminClient takes a Properties object that contains the same properties we've used with other clients, like `bootstrap.servers` and `client.id`. Note that the class `AdminClientConfig` holds constants for configuration values such as `BOOTSTRAP_SERVERS_CONFIG` as a helper for those names. Then we'll call the method `createTopics` on the client. Notice that the result, `topicResult`, is a Future. Listing 9.2 shows how to use the `AdminClient` class to create a new topic called 'selfserviceTopic'.

Listing 9.2 Using the AdminClient Create a Topic

```
NewTopic requestedTopic = new NewTopic("kinaction_selfserviceTopic", 2,2);  ❶
AdminClient client = AdminClient.create(props);                             ❷
CreateTopicsResult topicResult =
client.createTopics(Collections.singleton(requestedTopic));                 ❸

// topicResult.values().get("kinaction_selfserviceTopic").get();            ❹
```

❶ Create a `NewTopic` object with the topic name, two partitions, and two replicas.

❷ Create an AdminClient, which is the client interface to the cluster.

❸ Invoke `createTopics` method on the client which returns a `Future` object.

❹ The commented line shows how to get a specific `Future` for the topic 'kinaction_selfserviceTopic' that we just created

At this time there is no synchronous API, but we can make a synchronous call by using the get() function. In our case, that would mean starting with the `topicResult` variable and evaluating the Future that was returned for that specific topic name.

Because this API is still evolving, the following list of tasks that can be accomplished with the AdminClient highlights only some of the common functions that are available at the time of writing.

List of Client Admin Tasks [127]

- alter config
- create/delete/list ACLS
- create partitions
- create/delete/list topics
- describe/list consumer groups
- describe the cluster

The AdminClient is a great tool for building a user-facing application for those who wouldn't normally need or want to use the Kafka shell scripts, while also providing a way to control and monitor what is being done on the cluster.

9.1.2 Kafkacat

Kafkacat (github.com/edenhill/kafkacat) is a very handy tool to have on your workstation, especially when connecting remotely to your clusters. At this time, it focuses on being a producer and consumer client that can also give you metadata about your cluster. If you ever wanted to quickly work with a topic and didn't have the entire Kafka tools downloaded to your current machine, this executable will help you avoid the need to have those shell or bat scripts!

Listing 9.3 shows how to quickly get data into a topic using `kafkacat`, compared with the `kafka-console-producer` script that we used in Chapter 2.

Listing 9.3 Kafkacat Producer usage

```
kafkacat -P -b localhost:9092 -t kinaction_selfserviceTopic      ❶

// vs the shell script we used before

bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_selfserviceTopic                             ❷
```

❶ By sending in a broker from our cluster as well as our topic name, we can quickly write messages to that topic with the -P flag

❷ A reminder of the same functionality with the console producer command

Notice that for the `kafkacat` command the `-P` argument is passed to set the command into producer mode, which is going to help us send messages to the cluster. The `-b` flag is used to pass in our broker-list and the `-t` is used to pass the name of our target topic. Because we may also want to test the consumption of these messages, let's look at how we can use `kafkacat` as a consumer (listing 9.4). As before, we see the comparison between running the `kafkacat` command vs the `kafka-console-consumer` command. Notice that while the `-C` command is used to flag that the consumer mode is being used, the broker information is sent in with the same parameter as the producer mode.

Listing 9.4 Kafkacat Consumer usage

```
kafkacat -C -b localhost:9092 -t kinaction_selfserviceTopic      ❶

// vs the shell script we used before

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_selfserviceTopic                             ❷
```

❶ By sending in a broker from our cluster as well as a topic name, we can read messages from that topic with the -C flag

❷ A reminder of the same functionality with the console consumer command

Having a quick way to test our topics and gather metadata on our cluster make this small utility a nice-to-have in our toolbox. But by this point, you might be wondering if there are any other tools that we can use that are not command-line driven. And the good news, is yes, there are! For those that like REST, there is Confluent's REST Proxy.

9.1.3 Confluent REST Proxy API

Sometimes the users of our cluster might prefer to use APIs that are RESTful, because it is a common way to work between applications, either due to preference or ease-of-use. Also, some companies with very strict firewall rules about ports might express caution with opening more ports like those we have used so far for the broker connections, for example, 9092. One good option is to use the Confluent REST Proxy API (figure 9.2). The functionality of this proxy is similar to the `kafkacat` utility that we just discussed.

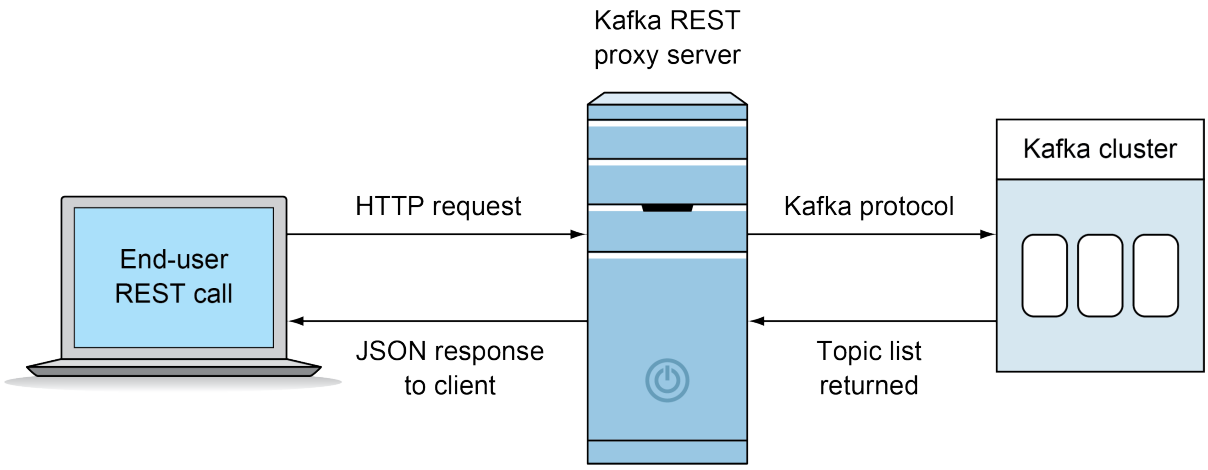

Figure 9.2 Confluent REST Proxy

At the time of this writing, the administration functions are limited more to querying the state of your cluster. The Confluent documents list administration options as future supported features [128]. This proxy is a separate application that would likely be hosted on its own server for production usage.

To test out how to leverage the REST proxy, let's start it up. For this to work, we will need to already have ZooKeeper and Kafka instances running before we start the proxy, as shown in listing 9.5.

Listing 9.5 REST Proxy startup

```
bin/kafka-rest-start.sh etc/kafka-rest/kafka-rest.properties    ❶
```

❶ Running this command from the installed Kafka folder will start-up the REST endpoint for our testing.

Since we're already familiar with listing topics, let's look at how that can be done with the REST proxy, using a command like `curl` to hit an HTTP endpoint (listing 9.6) [129].

Listing 9.6 Curl call to REST Proxy for Topic list

```
curl -X GET -H "Accept: application/vnd.kafka.v2+json" \                    ❶
  localhost:8082/topics                                                     ❷


// Output:
["__confluent.support.metrics","_confluent-metrics","_schemas","kinaction_alert"]
```

❶ Leveraging the Accept header helps specify the most specific format and version

❷ The endpoint /topics is our target to retrieve a list of topics we've created as well as the internal topics used by Kafka itself.

Because this is a 'GET' request, we can copy the following URL into a browser and see the

result: localhost:8082/topics. But using a tool like `curl` allows us to control the header we send with the request. By using 'Accept', we are telling our Kafka cluster what format and version we are using. The `v2` is the API version we are using with the format JSON that pertains to our metadata requests.

Because this is an evolving API, please keep up with the Confluent REST Proxy API Reference (docs.confluent.io/current/kafka-rest/api.html#crest-long-api-reference) as newer versions will come out and possibly have more features.

9.2 Running Kafka as a systemd Service

One decision we need to make, concerning running Kafka, is how to perform starts and restarts of our brokers. Those who are used to managing servers as a Linux-based service with a tool like Puppet (puppet.com/), may be familiar with installing service unit files and would likely leverage that knowledge to create running instances. For those not familiar with `systemd`, it initializes and maintains components throughout the system [130]. One common way to define Zookeeper and Kafka are as unit files that are used by `systemd`.

Listing 9.7 shows an example service unit file that would start a ZooKeeper service when the server starts. It will also restart after an abnormal exit. In practice, this means something like a `kill -9` command against the PID of this process would trigger a restart of the process. If you installed the Confluent tar during your setup (or refer to Appendix A if needed) there is an example service file located in the path `lib/systemd/system/confluent-zookeeper.service`. The Using Confluent Platform systemd Service Unit Files site at (docs.confluent.io/platform/current/installation/scripted-install.html#overview) provides details on using these files further.

Listing 9.7 Zookeeper Unit File

```
[Unit]
After=network.target

[Service]
Type=simple
ExecStart=/usr/kafka/bin/zookeeper-server-start.sh
 <linearrow /> /usr/kafka/config/zookeeper.properties   ❶
ExecStop=/usr/kafka/bin/zookeeper-server-stop.sh   ❷
Restart=on-abnormal   ❸

[Install]
WantedBy=multi-user.target
```

❶ ExecStart captures the start command that will be run and should be similar to what we manually ran so far to start ZooKeeper

❷ ExecStop captures how to property shut down the ZooKeeper instance

❸ The Restart condition would run ExecStart if an error condition caused the process to fail

The unit file should look very familiar to how we have been starting ZooKeeper so far in our examples. Listing 9.8 shows an example unit file for a Kafka broker. This file adds a dependency on ZooKeeper being started before it would be started itself. There is also an example file for the Kafka service in the Confluent tar at the path `lib/systemd/system/confluent-kafka.service`.

Listing 9.8 Kafka Unit File

```
[Unit]
Requires=zookeeper.service
After=zookeeper.service                                                    ❶

[Service]
Type=simple
ExecStart=/usr/kafka/bin/kafka-server-start.sh
 <linearrow /> /usr/kafka/bin/config/server.properties > /usr/kafka/broker.log 2>&1   ❷
ExecStop=/home/kafka/kafka/bin/kafka-server-stop.sh                        ❸
Restart=on-abnormal                                                        ❹

[Install]
WantedBy=multi-user.target
```

❶ This section would make sure that our ZooKeeper service is running. This dependency makes sense if both ZooKeeper and Kafka are on the same server.

❷ ExecStart captures the start command that will be run and should be similar to what we manually ran so far to start Kafka

❸ ExecStop captures how to property shutdown the Kafka instance

❹ The Restart condition would run ExecStart if an error condition caused the process to fail

The trick to this unit file is the fact that it requires the zookeeper.service we previously defined to be available and running before the Kafka service is started. This is a simplified example because our services are likely not going to be running on the same server.

Listing 9.9 shows that because our unit files are defined, we can now manage the services with `systemctl` commands.

Listing 9.9 Kafka Startup with Systemctl

```
sudo systemctl start zookeeper   ❶
sudo systemctl start kafka       ❷
```

❶ Using systemctl command to start the ZooKeeper service

❷ Using systemctl command to start the Kafka service

If you are using the example files that come with downloading the Confluent bundle, once you unzip the folder, check inside the root folder `../lib/systemd/system` to see examples of other service files that you can leverage for other services, like Connect, Schema Registry, and the Rest API to name a few. If you need help for different OS versions, a site like www.digitalocean.com/community/tutorials/how-to-install-apache-kafka-on-ubuntu-18-04 might provide help as well.

9.3 Logging

Besides Kafka's event logs that hold our event data, one of the other topics that we need to remember is the application logs, which Kafka produces as part of being a running program. The logs addressed in this section are not the events and messages from Kafka servers, but the output of the operation of Kafka itself. And we can not forget about Zookeeper either!

9.3.1 Kafka application logs

While we might be used to one log file for an entire application, Kafka has multiple log files that we might be interested in or need for troubleshooting. Due to multiple files, we might have to look at modifying different log4j appenders to maintain the necessary views of our operations.

NOTE **Which Kafka Appender**

The kafkaAppender name is not the same thing as the KafkaAppender itself (KafkaAppender). To use that as our appender we would need to update the following line as well as include dependencies for the clients and appender jars of the same versions:

```
log4j.appender.kafkaAppender=org.apache.kafka.log4jappender.KafkaLog4jAppender


<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-log4j-appender</artifactId>
    <version>2.7.1</version>
</dependency>
```

instead of the value `org.apache.log4j.ConsoleAppender` class. This is an interesting take on putting our logs directly into Kafka as some solutions parse the log files themselves and send them to Kafka.

By default, the server logs will continue being added to the directory as new logs are produced. No logs are removed - and this might be the preferred behavior if these files are needed for auditing or troubleshooting. However, if we do want to control the number and size, the easiest way is to update the file `log4j.properties` before we start the broker server. Listing 9.10 sets two important properties for the kafkaAppender: MaxFileSize and MaxBackupIndex [131].

Listing 9.10 Kafka server log retention

```
log4j.appender.kafkaAppender.MaxFileSize=500KB      ❶
log4j.appender.kafkaAppender.MaxBackupIndex=10      ❷
```

❶ This settings helps define the size of the file in order to determine when to create a new log file.

❷ This value helps set the number of older files to keep. This might be helpful if you want more than just the current log if needed for troubleshooting.

Note that modifying the `kafkaAppender` changes only how the server.log file is treated. If we want to apply different file sizes and backup file numbers for various Kafka-related files, we can use the appender to log file name table to know which appenders to update. Note that the appender name in the left column will be the logging key which will affect how the log files on the right will be stored on the brokers [132].

Table 9.1 Appender to Log Pattern

Appender Name	Log File Name
kafkaAppender	server.log
stateChangeAppender	state-change.log
requestAppender	kafka-request.log
cleanerAppender	log-cleaner.log
controllerAppender	controller.log
authorizerAppender	kafka-authorizer.log

Changes to `log4j.properties` require the broker to be restarted, so it is best to determine our logging requirements before starting our brokers for the first time if possible. We could also change the value with JMX but the value would not be persistent across restarts of the broker.

As we look at managing our logs, it is important to consider where the logs are being sent. Feeding error logs from Kafka into the same Kafka instance is something to avoid. While this advice might seem simple, it is amazing to see how many configuration values in other tools, like Flume, might be pointing to brokers in the same cluster. Evaluate where the errors are shipped to other tools and determine if that transport relies on the same cluster we are trying to monitor.

While we have focused on Kafka logs, we need to address our ZooKeeper logs as well. Since ZooKeeper is running and logging data just like our brokers, we will need to be mindful of logging output for those servers as well.

9.3.2 Zookeeper logs

Depending on how ZooKeeper was installed, we may also need to modify its logging configuration. The default configuration for ZooKeeper does not remove log files. However, our Kafka install may have added that for us. Either way, it is a good idea to make sure that the retention of the following application logs is what we would need for troubleshooting [133].

- `autopurge.snapRetainCount` This is the number of recent snapshots and the related transaction logs in the dataDir and dataLogDir locations. Once the number is exceeded, those log files are deleted. Depending on our needs, we might want to keep more or less. For example, if the logs are only used for troubleshooting, we would need lower retention than if they are needed for audit scenarios.
- `autopurge.purgeInterval` For cleanup to occur, we need to set this value to an integer greater than the default of 0. This will be the number of hours for how often the purge will take place.
- `snapCount` ZooKeeper logs its transactions to a transaction log. Setting this value will determine the amount of transactions that are logged to one file. If there are issues with file size, we might need to set this number less than the default (10,000).
- `preAllocSize` Actual mileage might vary with this setting, but we can modify the block size to allocate a smaller size than the default of 64 MB.

There are also other solutions to log rotation and cleanup that we might consider beyond log4j. `logrotate`, for example, is a helpful tool that enables options such as log rotation and compression of logs files.

Logfile maintenance is an important administration duty. However, there are other tasks that we need to consider as we start to roll out a new Kafka cluster, such as making sure that clients can connect to our brokers.

9.4 Firewall

Depending on network configuration, we might need to serve clients that exist inside the network or those out of the network where the Kafka brokers are set up.

Kafka brokers can listen on multiple ports. For example, the default for a plain text port is 9092. An SSL port can also be set up on that same host at 9093. Both of these ports might need to be open depending on how clients are connecting to our brokers.

In addition, Zookeeper ports include 2181 for client connections. Port 2888 is used by follower Zookeeper nodes to connect to the leader Zookeeper node and port 3888 is also used between Zookeeper nodes to communicate [134].

Also, if connecting remotely for JMX or other Kafka services (such as the REST Proxy), remember to account for any exposure of that port to other environments or users. In general, if we are using any command-line tools that require a port on the end of the hostname, for

ZooKeeper or Kafka servers, we need to make sure that these ports can be reached, especially if a firewall is in place.

9.4.1 Advertised listeners

One error when connecting that often appears like a firewall issue is when using the listeners and advertised.listeners properties. Clients will have to use the correct hostname to connect if given - so it will need to be reachable however the rules are setup. For example, let's look at an example `listeners` vs `advertised.listeners` where those values might not be the same.

Let's imagine we are connecting to a broker and can get a connection when the client starts up, but not when it attempts to consume messages? How could we have this behavior that appears inconsistent? Remember that whenever a client starts up, it connects to any broker to get metadata about which broker to connect to. The first initial connection from the client uses the information that is located in the Kafka `listeners` configuration. What it gives back to the client to connect to next is the data in Kafka `advertised.listeners`. This makes it so the client might be connecting to a different host to do its work. Figure 9.3 shows how the client uses one hostname for the first connection attempt then uses a different hostname on its second connection. This second hostname was given to the client from its first initial call as the new location to connect to.

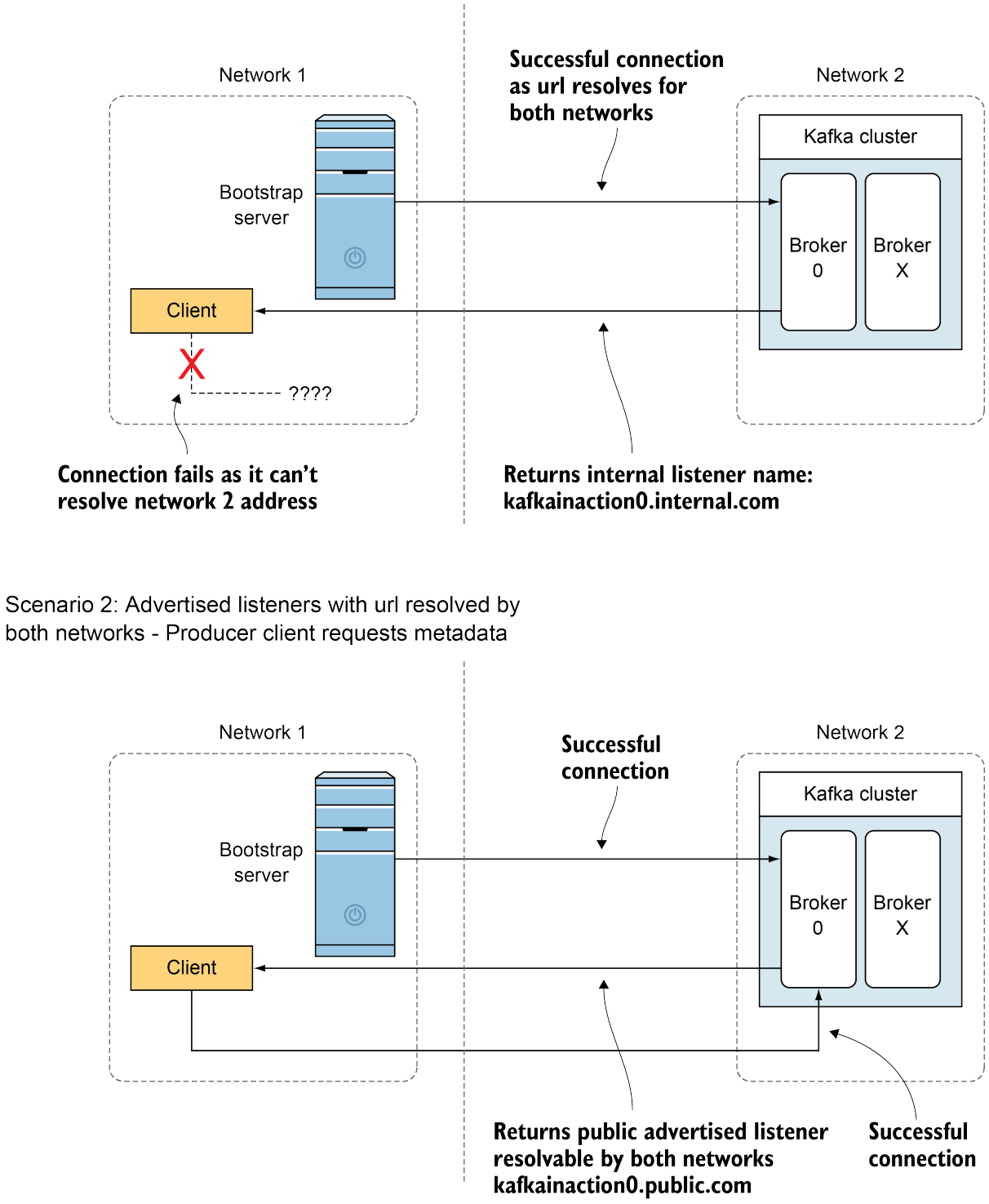

Figure 9.3 Advertised listeners difference from listener

`inter.broker.listener.name` is also an important setting to look at that will determine how the brokers connect across the cluster to each other. If the brokers cannot reach each other,

replicas will fail and the cluster will not be in a good state, to say the least! For an excellent explanation of advertised listeners check out the article by Robin Moffatt titled "Kafka Listeners – Explained" if you wish to dig into more details [135].

9.5 Metrics

In Chapter 6, we looked at an example of setting up a way to see some JMX metrics from our application. The ability to see those metrics is the first step. The harder part is picking out which are important to our cluster. Let's take a peek at some that are likely to highlight areas of concern.

9.5.1 JMX Console

It is possible to use a GUI to explore the exposed metrics and get an idea of what is available. VisualVM (visualvm.github.io/), is one example. Other tools are available that can help with this as well, such as the popular JConsole. Looking around at the available JMX metrics can help us discover points of interest for which we might choose to add alerts. When installing VisualVM, be sure to go through the additional step of installing the MBeans browser. We can do this by going to the menu bar Tools → Plugins.

As noted in Chapter 6, we must have a `JMX_PORT` defined for each broker we want to connect to. This can be done with an environment variable in the terminal like so: `export JMX_PORT=1099`. It is interesting to note that if you have ZooKeeper running on the same server as your broker, its configuration might rely on this same property. Make sure that you scope it correctly to be separate for each broker as well as each ZooKeeper node.

`KAFKA_JMX_OPTS` is also another option to look at for connecting remotely to Kafka brokers. Make sure to note the correct port and hostname. The following example uses port 1099 and the localhost as the hostname. Note that the other parameters allow us to connect without SSL and not have to authenticate. Listing 9.11 shows an example setting `KAFKA_JMX_OPTS` with various arguments [136].

Listing 9.11 Kafka JMX Options

```
KAFKA_JMX_OPTS="-Djava.rmi.server.hostname=127.0.0.1    ❶
  -Dcom.sun.management.jmxremote.local.only=false       ❷
  -Dcom.sun.management.jmxremote.rmi.port=1099          ❸
  -Dcom.sun.management.jmxremote.authenticate=false     ❹
  -Dcom.sun.management.jmxremote.ssl=false"             ❹
```

❶ Setting the hostname for the RMI server to be localhost

❷ Allowing remote connections

❸ The port we are exposing for JMX

❹ Turning off authenticate and SSL checks

Let's take a look at a key broker metric and how to locate the value we need with the help of Figure 9.4, which shows using a small MBeans representation to see the value of UnderReplicatedPartitions. Using the name such as `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`: we drill down what looks like a folder structure starting with kafka.server.

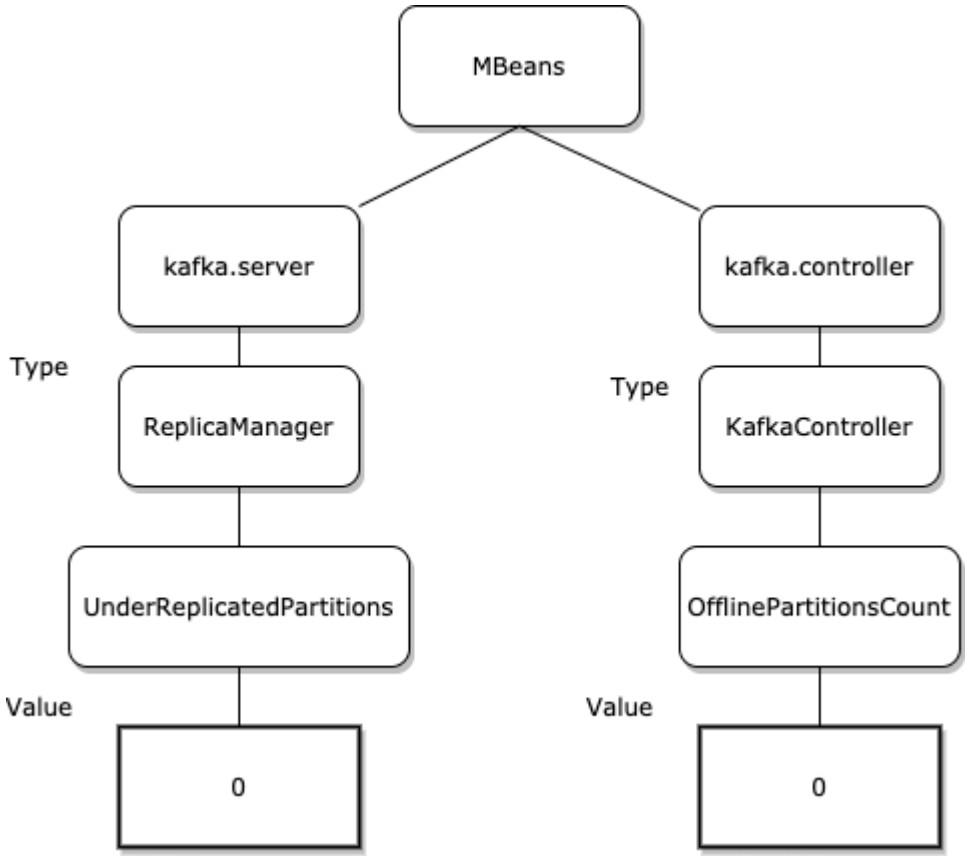

Figure 9.4 UnderReplicatedPartitions and OfflinePartitionsCount Locations

Continuing on, we will then find the type ReplicaManager with the attribute named 'UnderReplicatedPartitions'. Now that you have seen how to browse to specific values, let's go into detail about some of the most important things to look at on our servers.

9.5.2 Broker alerts

Because Kafka brokers are built as part of a cluster, some problems might be hard to spot without keeping an eye on the specific data points. Let's look at some metrics that might mean action is needed to help restore a cluster. If you are using Confluent Control Center or Confluent Cloud, most of the metrics are used in the built-in monitoring. Confluent Platform suggestions alerting on the following top three values to start with [137]. I also included `OfflinePartitionsCount` as that is a metric I find useful and recommend. As a head start, we have already located two of these in the the JMX Console section!

`UnderMinIsrPartitionCount` is located at kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount. Any value greater than zero is a cause of concern. This total is the number of partitions less than the number required for the minimum ISR.

`UnderReplicatedPartitions` is located at kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions. This is again the number of partitions and this metric being greater than 0 is an issue. We are not getting the number of replica copies that we expect and we can also handle fewer failure scenarios.

`UnderMinIsr` is located at kafka.cluster:type=Partition,topic={topic},name=UnderMinIsr,partition={partition}. This is again related to the amount of ISRs being less than the minimum number. For example, if we have three replicas and we need acknowledgment from all before our produce request is satisfied, we will not have any messages written to the topic partition.

`OfflinePartitionsCount` is located at kafka.controller:type=KafkaController,name=OfflinePartitionsCount. Because clients read and write to a leader partition on a broker-there always needs to be a leader for clients to be successful. If the number of offline partitions is greater than 0, the ability to read and write to that partition is compromised.

9.5.3 Client alerts

While our brokers should always be running, an important thing to note is that if clients are not running actively against the cluster, we will not get those metrics. Having a time-series database of past metrics (like Prometheus) might help to capture more than a point-in-time view of what our producers are doing. Let's dig into two important client metrics to keep in mind [138].

`record-error-rate` is located at kafka.producer:type=producer-metrics,client-id={YOUR_ID_HERE},name=record-error-rate. The ideal value of this attribute is zero. This is an average for the number of messages that failed per second for producer clients [139]. On a side note, we can get this metric at more than one level as well so make sure to pick the right metric!

The following consumer metric is also a good place to start monitoring consumer clients and is listed in the Confluent Platform documentation [140].

`records-lag-max` is located at kafka.consumer:type=consumer-fetch-manager-metrics,client-id={YOUR_ID_HERE},name=records-lag-max. This value is the maximum number of records for a given partition in which the consumer is behind the producer. If we notice an increasing value over time, we might consider alerting on

this. This is where having Prometheus monitoring could help show a trend over time. Let's dig into more details about *consumer lag* and also look at a different monitoring option.

9.5.4 Burrow

Although we can usually rely on Kafka being fast for event delivery, there can be times when a specific consumer client gets behind in pulling current messages. Even if a client does not stop or fail on its own, sometimes it might just have more messages to process than its normal workload. Subtracting the highest offset for a partition and the offset that the consumer client is currently processing is called consumer lag.

Burrow (github.com/linkedin/Burrow) is a tool that provides consumer lag checking and seems to have been developed with a thoughtful mindset toward its subject. I was introduced to this tool in the hallways of a Kafka Summmit conference and was excited right away to learn more. I say thoughtful because Burrow's code shows the marks of experienced operators that have worked with Kafka enough to be able to automate more meaning behind this metric. One such issue that is addressed is that the lag metric itself would be driven by and impacted by the consumer client that was doing its own reporting. Also, we might see that thresholds are hard to set for a topic if we do not know the normal lag of our consumers. My first attempt at a script for checking lag monitoring was a script similar to one shown in listing 9.12.

Listing 9.12 Kafka Consumer Group Command to find lag usage

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --describe --group alert   ❶

#Truncated example output          ❷
TOPIC   PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
alert    0          3               3               0
```

❶ By doing a describe argument with my group-I will find a number for lag.

❷ Truncated example output showing the lag value for one consumer group

Looking at Listing 9.12, we see the column heading 'Lag' and we could choose to parse this output for our custom monitoring. But after finding Burrow, I did not rely on this script as much. The ability to avoid setting a specific or hard-coded threshold number helped avoid managing various consumer-group-to-threshold values that I had no idea about regarding the normal rate of consumption. Your mileage may vary with Burrow, but it is an interesting tool to monitor your lag.

9.6 Tracing with Interceptors

The built-in metrics that we looked at so far can give us a great snapshot of current health, but what if we wanted to trace a single message through the system? What could we use to see a produced message and see its consumed status?

Let's talk about a simple but straightforward model that might work for our requirements. Let's say that we have a producer in which each event has a unique ID. Because each message is important, we do not want to miss any of these events. With one client, the business logic runs as normal and consumes the messages from the topic. It makes sense to log to a database or flat file the ID of the event that was processed. A separate consumer, let's call it an auditing consumer in this instance, would fetch data from the same topic and make sure that there were no IDs missing from the processed entries of the first application. Though this process can work well, it does require adding logic to our application, and so might not be the best choice. Figure 9.5 shows a different approach using Kafka interceptors.

In practice, the interceptor that we define is a way to add logic to the producer, consumer, or both by hooking into the normal flow of our clients, intercepting the record, and adding our custom data before it moves along its normal path. Our changes to the clients are configuration-driven and help keep our specific logic out of the clients for the most part.

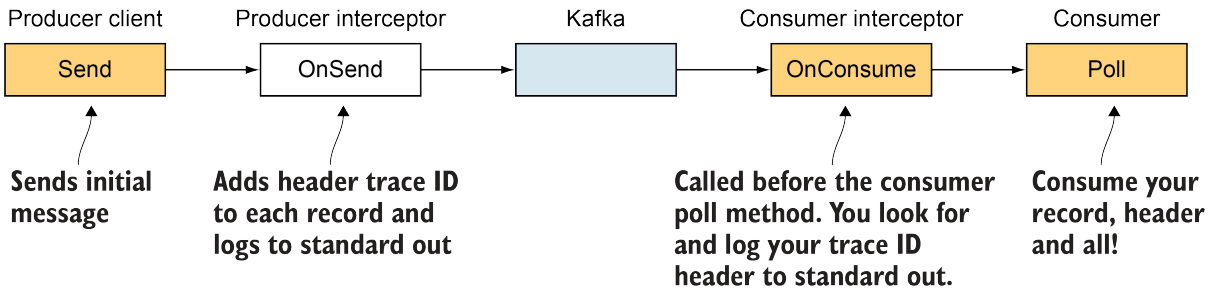

Figure 9.5 Interceptors for Tracing

Let's revisit the concept of interceptors that we touched on briefly in Chapter 4 in introducing what producer interceptors could do for our messages. By adding an interceptor on both the producer and consumer clients that we are using, we can separate the monitoring logic from the application logic itself. The crosscutting concern of monitoring can hopefully be more encapsulated by this approach.

9.6.1 Producer interceptor

It is also interesting to note that we can have more than one interceptor so we don't have to include all of our logic in one class: we can add and remove others later. The order that we list the classes is important as that is the order in which logic will run. The first interceptor gets the record from the producer client. If the interceptor modified that record, note that it might not be the exact same as the first interceptor received [141].

Let's start with looking at the Java interface `ProducerInterceptor`. We'll add this new interceptor to our Alert producer that we used in previous examples in Chapter 4. We will create a new class called `AlertProducerMetricsInterceptor` to add logic around alerts being produced, as in Listing 9.13.

Implementing the interface `ProducerInterceptor` allows us to hook into the producer's interceptor life-cycle. The logic in the `onSend` method will be called by the send from the normal producer client we have used so far. For our example, we are going to add a header called 'traceId'. Using a unique ID will help us confirm on the consumption side that we are seeing the same message at the end of its journey that we are producing in this step.

Listing 9.13 AlertProducerMetricsInterceptor Example

```
public class AlertProducerMetricsInterceptor implements ProducerInterceptor<Alert, String> {
 ❶

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

  public ProducerRecord<Alert, String> onSend(ProducerRecord<Alert, String> record) {     ❷
    Headers headers = record.headers();
    String traceId = UUID.randomUUID().toString();
    headers.add("traceId", traceId.getBytes());                                          ❸
    log.info("Created traceId: {}", traceId);
    return record;                                                                       ❹
  }

  public void onAcknowledgement(RecordMetadata metadata, Exception exception) {         ❺
    if (exception != null) {
      log.info("producer send exception " + exception.getMessage());
    } else {
      log.info("ack'ed topic = {}, partition = {}, offset = {}",
               metadata.topic(), metadata.partition(), metadata.offset());
    }
  }
    // rest of the code omitted
}
```

❶ Implement ProducerInterceptor to hook into the interceptor lifecycle.

❷ The onSend method is called by the producer client send method

❸ Add a custom header to the record to carry the generated ID across Kafka

❹ Return the record that was modified. This will include our new header

❺ `onAcknowledgement` will be called when a record is acknowledged or an error occurs

We also have to modify our existing `AlertProducer` class to register the new interceptor. The property name `interceptor.classes` will need to be added to the producer configuration with a value of our new class: `org.kafkainaction.producer.AlertProducerMetricsInterceptor`. Although I used the property name for clarity, remember that we can use the constant provided by the `ProducerConfig` class. In this case, we would use `ProducerConfig.INTERCEPTOR_CLASSES_CONFIG`. Listing 9.14 shows this required modification.

Listing 9.14 AlertProducer with Interceptor Configuration

```
Properties props = new Properties();
...
props.put("interceptor.classes",    ❶
  "org.kafkainaction.producer.AlertProducerMetricsInterceptor");   ❷

Producer<Alert, String> producer = new KafkaProducer<Alert, String>(props);
```

❶ Use the property name `interceptor.classes` to set our interceptors

❷ The value can be one as in our case or else a comma-separated list

In this example overall, we have one interceptor that logs out a unique ID for each produced message. We will add this ID as a header to the record so that when a consumer pulls this message, a corresponding consumer interceptor will log this ID that it has processed. The goal is to provide our own end-to-end monitoring that is outside of Kafka. By parsing the application logs, we will see messages like the following that came from our `AlertProducerMetricsInterceptor` class as shown, in Listing 9.15.

Listing 9.15 Alert Interceptor Output

```
Created traceId: 603a8922-9fb5-442a-a1fa-403f2a6a875d       ❶
acked topic=alert, partition=0, offset=1
```

❶ An example of our logged value added by the producer interceptor.

Now that we have completed setting up an interceptor for sending a message, we need to see how to implement similar logic on the consumer end of our system. We want to validate that we can see that same header value that we added with the producer interceptor on the consumption end. Listing 9.16 shows an implementation of the `ConsumerInterceptor` class to help retrieve this header.

9.6.2 Consumer Interceptor

Listing 9.16 AlertConsumerMetricsInterceptor Example

```
public class AlertConsumerMetricsInterceptor
  implements ConsumerInterceptor<Alert, String> {          ❶

  public ConsumerRecords<Alert, String>
    onConsume(ConsumerRecords<Alert, String> records) {
      if (records.isEmpty()) {
        return records;
      } else {
        for (ConsumerRecord<Alert, String> record : records) {
          Headers headers = record.headers();              ❷
          for (Header header : headers) {
            if ("traceId".equals(header.key())) {          ❸
              log.info("TraceId is: " + new String(header.value()));
            }
          }
        }
      }
    return records;                                        ❹
  }
}
```

❶ Implementing ConsumerInterceptor is needed to allow Kafka to recognize our interceptor.

❷ We are looping through each record's headers.

❸ Once we find the custom header we added from the producer interceptor, we will log it to standard output.

❹ We return the records to continue on the path of callers from our interceptor.

In a similar fashion as our producer, we use a consumer-specific interface, `ConsumerInterceptor`, to make our new interceptor. We are looping through all of the records and their headers to find any that have our custom 'traceId' as the key and sending this to standard output. We also have to modify our existing `AlertConsumer` class to register our new interceptor. The property name `interceptor.classes` will need to be added to the consumer configuration with a value of our new class: `org.kafkainaction.consumer.AlertConsumerMetricsInterceptor`.

Listing 9.17 shows this required step.

Listing 9.17 AlertConsumer with Interceptor Configuration

```
public class AlertConsumer {

Properties props = new Properties();
...
props.put("group.id", "alertinterceptor");                 ❶
props.put("interceptor.classes",                           ❷
  "org.kafkainaction.consumer.AlertConsumerMetricsInterceptor");   ❸
...
}
```

❶ Using a new `group.id` for this example to make sure we are starting with our current offsets (and not one from a previous group.id)

❷ The property name `interceptor.classes` is needed to add our custom interceptor

❸ Our custom class is added as the value. We can include a comma-separated list if we have more than one class we need to use.

Although I used the property name for clarity, remember that we can use the constant provided by the `ConsumerConfig` class. In this case, we would use `ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG`. While we can see the usage of interceptor on both ends of our flow, there is also another way to add functionality to client code: overriding clients.

9.6.3 Overriding Clients

If we control the source code for clients that other developers will use, we can also subclass an existing client or create our own that implements the Kafka Producer/Consumer interfaces. The Brave (github.com/openzipkin/brave) project has an example of one such client that adds tracing data at the time of writing. For those not familiar with Brave, it is a library meant to help add instrumentation for distributed tracing. It has the ability to send this data to something like a Zipkin (zipkin.io/) server, which can handle the collection and search of this data. Take a peek at the `TracingConsumer` class (github.com/openzipkin/brave/blob/master/instrumentation/kafka-clients/src/main/java/brave/kafka/clients/TracingConsumer.java) for a real-world example of tracing with Kafka.

Both the producer and consumer clients can be decorated to enable tracing (or any custom logic), but we'll focus on the consumer client in the following example. The code in Listing 9.18 is an example section of code to add custom logic to the normal Kafka consumer flow. Developers wanting to consume messages with the custom logic will use an instance of `CustomConsumer` which includes a reference to a regular consumer client named `kafkaConsumer` in this example. The custom logic functionality will be added to provide our needed behavior while still interacting with the traditional client. This code is an example of yet another way to provide tracing for Kafka messages.

Listing 9.18 Custom Consumer Client

```
final class CustomConsumer<K, V> implements Consumer<K, V> {
...
  final Consumer<K, V> kafkaConsumer;   ❶

  @Override
  public ConsumerRecords<K, V> poll(final Duration timeout) {   ❷
    //Custom logic here   ❸
    return kafkaConsumer.poll(timeout);  // Normal Kafka consumer used as normal   ❹
  }
...
}
```

❶ The normal Kafka consumer client is still being used in our custom consumer.

❷ Users of the consumer would still call the interface methods they were used to.

❸ Our custom logic is added where needed.

❹ The normal Kafka consumer client is leveraged to provide its normal duties.

This added behavior is not getting in the way of the normal client, it is adding the ability to add tracing information.

9.7 General monitoring tools

Because Kafka is a Scala application, it has the ability to leverage JMX and uses the Yammer Metrics library [142] as we have seen already. This library was used to provide JMX metrics on various parts of the application, and we have seen some of the options we can evaluate already. But as Kafka usage has expanded, there are some tools out there that leverage JMX metrics as well as administration-related commands to provide easier-to-manage clusters. Of course, this is not a complete list of options and the features of those listed might change over time. Let's take a look at a few options.

9.7.1 CMAK / Kafka Manager

Cluster Manager for Apache Kafka (CMAK) (github.com/yahoo/CMAK), once known as Kafka Manager, is an interesting project that focuses on managing Kafka as well as being a UI for various administrative activities. One key feature is the ability to manage multiple clusters. When we look at the commands that we have run so far, we have been looking at one cluster. These commands can be harder to scale the more clusters we are running at once. Other features include inspection of our overall cluster state and the ability to generate and run partition reassignment. The tool can also deal with authenticating users with LDAP, which might be needed for enterprise usage.

9.7.2 Cruise Control

Cruise Control (github.com/linkedin/cruise-control) was created by the developers at LinkedIn. Because they have thousands of brokers across their clusters, they have experience running Kafka clusters and have helped codify and automate some of their pain points over the years. A REST API can be used as well as a UI, so we have a couple of ways to interact with this tool. Some of the most interesting features are how Cruise Control watches our cluster and can generate suggestions on rebalances based on workloads.

9.7.3 Confluent Control Center

Confluent Control Center (docs.confluent.io/current/control-center/index.html) is another web-based tool that can help us monitor and manage our clusters. But one item to note is that it is a commercial feature that would need an enterprise license for a production setup. If you already have a subscription to the Confluent Platform, there is no reason not to check it out. This tool uses dashboards and can help identify message failures, network latency, and other external connectors.

9.7.4 General monitoring needs

Not all of our monitoring will be focused on the Kafka application itself. A couple of core operating system monitoring needs appear in order to provide a solid foundation for Kafka to run on. Let's look at a few specifics, which I'll touch on briefly.

File handles are something to be aware of. This number will include open file handles that Kafka has on our system. There will be a filehandle for each log file on the broker. In other words, the number can add up quickly. When installing Kafka, it is wise to adjust the file descriptors limits. While entering a large number more than 100,000 is not unheard of, we still might want an alert when that number is almost gone. If not, we will get a generic-looking error like the following: `java.io.IOException: Too many open files` [143].

We will want to make sure that we do not have unnecessary swapping of RAM. Kafka likes to be fast! Swapping can affect performance of the brokers, and the less that occurs, the better [144]. Make sure only a small amount of swap is allowed and alerted on for monitoring.

Overall, Kafka provides us with many options to not only manage but to monitor our cluster. Distributed systems are difficult and the more experience you gain, the more your monitoring skills and practices will also likely improve.

9.8 Summary

- Besides the shell scripts that are packaged with Kafka, an administration client also exists to provide API access to important tasks such as creating a topic.
- Tools such as Kafkacat and the Confluent REST Proxy API also allow ways for developers to interact with the cluster.
- While Kafka leverages a log for client data at its core, there are still various logs specific to the operation of the broker that should be maintained. Managing these logs (and ZooKeeper logs) should be addressed to provide details for troubleshooting or auditing when needed.
- Understanding advertised listeners can help determine behavior that at first appears inconsistent for client connections. Listeners alone might not determine how clients connect.
- Kafka uses JMX for metrics. You can see metrics from clients (producers and consumers) as well as from the brokers.
- Producer and consumer interceptors can be used to implement crosscutting concerns. One such example would be adding tracing ids for message delivery monitoring.

10 Protecting Kafka

This chapters covers

- Reviewing security basics and related terminology
- Setting up SSL between a cluster and clients
- Inspecting Kerberos and Java Authentication and Authorization Service (JAAS) files
- Exploring access control lists (ACLs) for permissions
- Network bandwidth and request rate quotas to limit demands on resources

This chapter will focus on keeping our data locked down to only those that need to read or write to it. Since security is a huge area to cover, we will talk about some basic concepts to get a general background on the options we have in Kafka. This will not be an complete guide of security in general, but with that foundation, we will try to discuss some of the practical actions you would take in your setup. We will have client impacts as well as brokers and ZooKeeper to make our cluster more secure.

Of course, your data might not need some protections we discuss. If you are not tracking anything of a secure nature, i.e. personal information or financial data, then you might not even need this protection. Knowing your data is key to deciding if you need the tradeoffs of managing access. For a quick example, if you are handling anything related to personal information like date of birth or credit card numbers, you will most likely want to look at most of the security options discussed in this chapter. However, if you are only handling generic information such as the number of web clicks on a series of marketing campaigns, your cluster might not need to introduce features like SSL that might slow down your throughput while at the same time increasing your CPU needs.

Let's start with an example of made up data that we would want to protect. That would be data around solving the puzzle book Masquerade written by Kit Williams and published in 1979 by

Jonathan Cape. If you have not heard of this book, let's set the context for our task. By looking at 16 paintings, readers are meant to find clues to a treasure hunt. The reward: the location of a golden hare. As a competition-wide exercise, we have broken into teams and do not want other co-teams to access our own team's work. Spoiler alert: those of you who are familiar with this work, in particular, know that the location was indeed already found. But we will pretend like it's still undiscovered for our use-case!

Starting out, each team picks their own topic names and shares that name with their team members only. Without knowing which topic name to write and read from, your data is out of sight of the other teams. Each team starts out sending in their clues to what they assume is their 'private' topics. Over time, members of various teams start to wonder about the progress of other teams and if they have any clues that they don't. This is when the trouble starts. Figure 10.1 shows the topic setup.

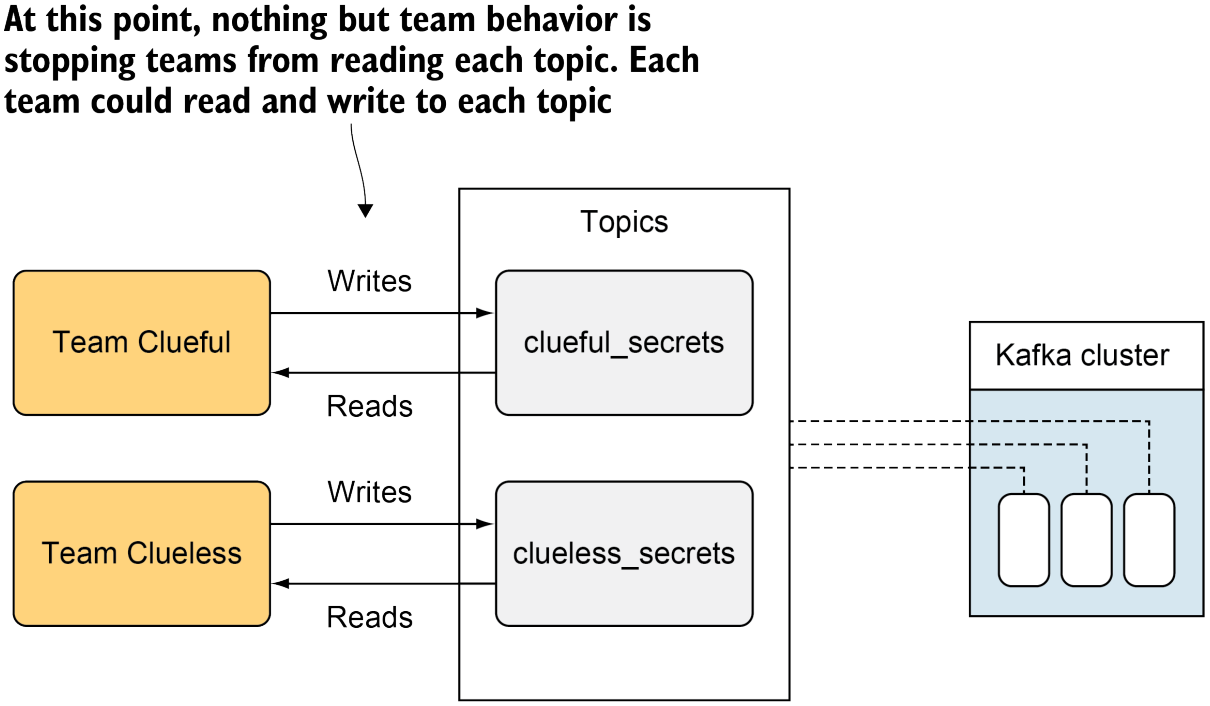

Figure 10.1 Treasure Hunt Topics

One tech-savvy competitor who has used Kafka before reaches for his command-line tools to find out the name of the other topics. After doing a list of topics, the user knows the topics of his rivals. For example, let's say that this one team member of Team Clueless looks at the topic of Team Clueful: topic clueful_secrets. With great happiness, all it takes is a consumer console command to be able to list all of the data that Team Clueful has been working on in the competition so far! But the bad actor did not stop there. In order to throw Team Clueful off of the trail, the actor also wrote false information into the channel. Now Team Clueful has bad data in

their topic that is hindering their clue-solving progress! Since they are not sure who really wrote the message on the topic, the team has to go determine the false messages and lose valuable time they could be using to work on figuring out the grand-prize location.

How could we avoid this situation? Is there a way that only those clients that have permission would be able to read from or write to a topic? There are two parts to our solution. The first is how to find out who a person is in our system. Not only who they are, but also making sure that the claimed identity of the user can be verified. The second part is that once we verify a user, we need to know what they are able to do in our system? We will dive deeper into these topics as we look at some of the solutions provided with Kafka.

10.1 Security Basics

In regards to computer application security, you will likely encounter encryption, authentication, and authorization at some point in your journey.

Encryption can be various processes that change your data in a way that only authorized parties can access the content. Encryption does not mean that others might not see your messages, but if they do, they will not be able to derive the original content that you are protecting. Many people will think of how they are encouraged to use a site that is HTTPS for online shopping on a WiFi network. Later, we are going to enable SSL for our communication not between a website and our computer, but between our clients and brokers.

Moving forward, let's talk about authentication first in a real-world example. To verify the identity of a user or application, we need to have a way to authenticate that user: a process to prove that they are who they claim they are. If you went to sign up for a library card, does the library issue the card to everyone without making sure the user is who they say they are? In most cases, the library would confirm the person's name and address with something like a government-issued id and utility bill. This process is intended to make sure that others cannot claim another identity to use for their own purposes easily. If someone had claimed your identity to borrow books and never return them, sending the fines your way, we can easily see a drawback of not confirming the user's claim.

Authorization, on the other hand, instead focuses on what the user can do. Continuing with our library example above, our card as an adult might give us different permissions than if it was for a user considered as a child. Access to online publications might be limited for that user to only terminals inside the library.

10.1.1 Encryption with SSL

So far all of our brokers have supported plain text. In effect, there has been no authentication or encryption over the network. Knowing these facts, it might make more sense when reviewing one of the broker server configuration values. If you look at the current server.properties file, you will find an entry like the following: `listeners=PLAINTEXT:localhost//:9092`. That listener is in effect providing a mapping of a protocol to a specific port on the broker. Since the brokers support multiple ports, this will allow us to keep the PLAINTEXT port up and running as we test adding SSL, or other protocols, on a new port. Having both ports will help make our transition smoother as we shift over a period of time away from PLAINTEXT [145]. Figure 10.2 shows an example of using PLAINTEXT vs SSL.

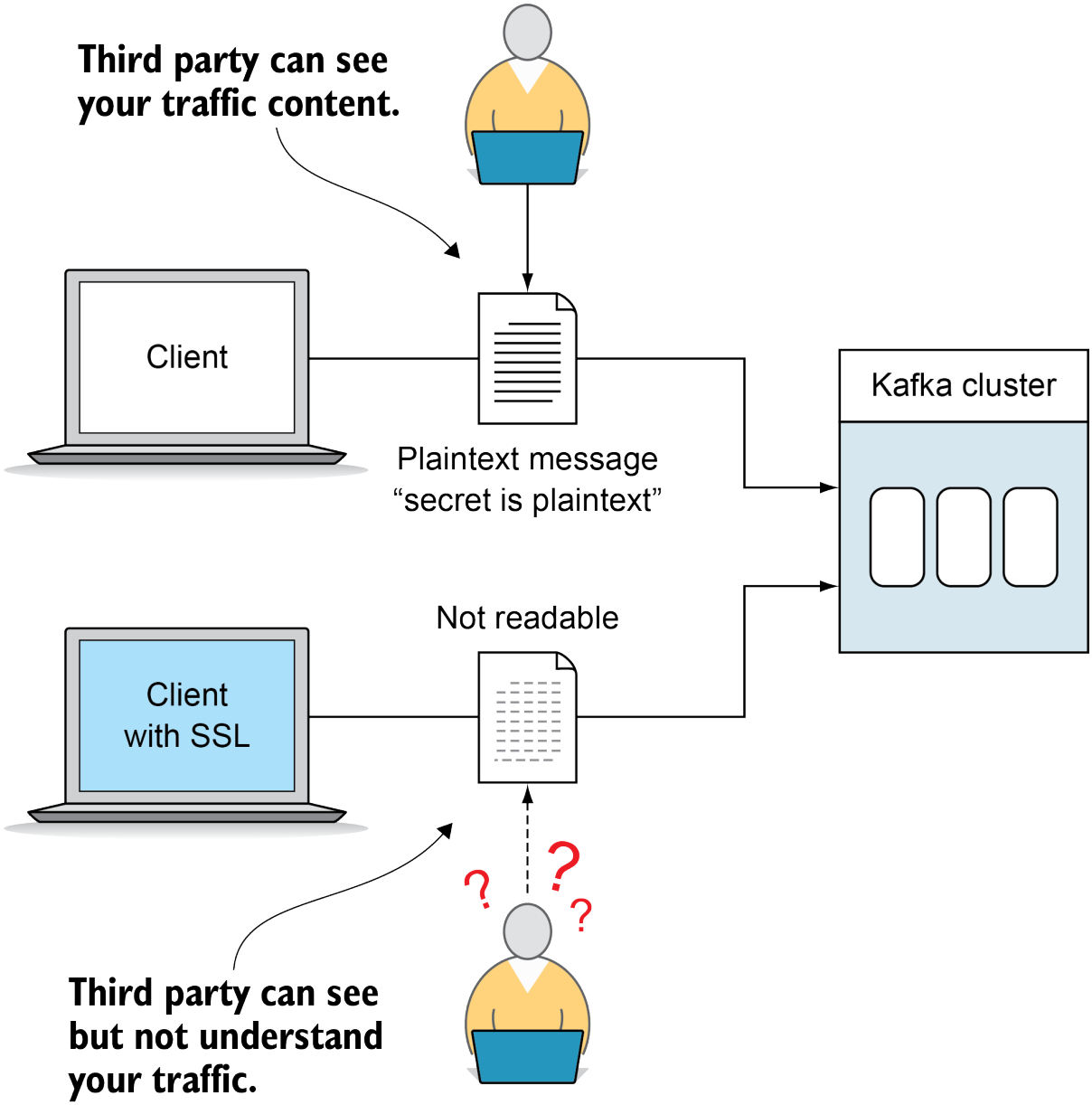

Figure 10.2 PLAINTEXT vs SSL

At this point, we are starting with a cluster without any security really baked in. Luckily, we can add various pieces to our cluster as we harden it against other teams. Setting up SSL between the brokers in our cluster and our clients is one place to start. No extra servers or directories are needed. No client coding changes should be required as the changes will be configuration driven. We don't know how advanced other teams are with listening to our traffic on the same WiFi network with WireShark, so we know we might not want to send PLAINTEXT from our brokers to our clients. While this following setup is needed for Kafka security, those that have set up SSL or HTTPS in the past (and especially with Java) will find this approach similar to other client/server trust setups.

NOTE	**While Transport Layer Security (TLS) is the successor of SSL, the label SSL has been maintained as the property name you will see in our examples. To avoid confusion between explanations and the actual values used, we will keep the term SSL throughout this material [146].**

10.1.2 SSL Between Brokers and Clients

In our previous examples of writing clients and connecting to Kafka, we have not used SSL to connect. However, now we are going to look at turning it on for the communication between our clients and our cluster to encrypt our network traffic with TLS.

Let's walk through the process and see what we are going to need to accomplish in order to get our cluster updated with this feature.

- Create a key per broker
- Create a certificate per broker
- Use a certificate authority (CA) to sign each broker's certificate
- Add the CA to the client truststore
- Sign each cluster certificate and import that back into its keystore with the CA cert
- Make sure that clients and the broker have their properties updated to use their new truststores

> **NOTE** The commands in this chapter are very specific and will not work on all operating systems the same (or even across different server domain names listed for broker setup without modification). The important thing is to follow along with the general concepts. For example, other tools can be switched out (like for openssl), so your setup and commands might be different. Once you get the concepts, head to Confluent's site at docs.confluent.io/platform/current/security/security_tutorial.html for even more resources and guides if needed. Confluent's documents are referenced below that helped us with our examples. A security professional should be consulted for the correct way to setup your own environment. Our commands are meant as a guide for getting familiar and learning, not as a production level of security. This is not a complete guide. Use at your own risk!

One of our first steps is to create a key and certificate for our brokers. Since you should already have Java on your machine-one option is to use the `keytool` utility that is part of a Java installation. The `keytool` application is used to manage a keystore of keys and trusted certificates. The important part to note is the storage. The term `broker0` is included in some file names to suggest that this is only for one specific broker - not one that is meant for every broker. It might be good to think of it as a database where our JVM programs can lookup up this information for our processes when it's needed. At this point, we are also going to leverage the ability to generate a key for our brokers in Listing 10.1 [147].

Listing 10.1 SSL Key Generation for a Broker

```
keytool -genkey -noprompt \
  -alias localhost \
  -dname "CN=kia.manning.com, OU=TEST, O=MASQUERADE, L=Portland, S=Or, C=US" \
  -keystore kafka.broker0.keystore.jks \   (1)
  -keyalg RSA \
  -storepass masquerade \   (2)
  -keypass masquerade \
  -validity 999
```

(1) This will be the name of the keystore file that will hold our newly generated key

(2) We have used a password so that the store cannot be changed without it

After running this command, we have created a new key and stored it in the keystore file: kafka.broker0.keystore.jks.

Since we have a key that in a way identifies our broker, we need something to signal that we don't just have any certificate issued by a random user. One way to verify our certificates is by signing them with a CA. You might have heard of CAs such as Let's Encrypt (letsencrypt.org/), Entrust (www.entrust.com/), or GoDaddy (www.godaddy.com/) to name a few. The role of a CA

is to act as a trusted authority that certifies the ownership and identity of the public key. In our examples, we are going to be our own CA to avoid any need of verifying our identity to a third-party. Our next step is to create our own CA as shown in Listing 10.2 [148].

Listing 10.2 Creating our own Certificate Authority

```
openssl req -new -x509 -keyout cakey.crt -out ca.crt \   ❶
  -days 999 -subj '/CN=localhost/OU=TEST/O=MASQUERADE/L=Portland/S=Or/C=US' \
  -passin pass:masquerade -passout pass:masquerade
```

❶ We create a new CA and want a key file as well as the cert file produced

After we have created our own CA, we will use it to sign our clients' truststore and brokers' truststore. This generated CA is now something that we want to let our clients know that they should trust. Similar to the term keystore, we will use a truststore to hold this new information. We will use `openssl` as a tool to help us act as this CA. Now that we have generated our CA, we will use it to sign our certificates for our brokers that we have already made. First, we will export the cert that we generated earlier for each broker from the keystore, sign that with our CA, and then import both the CA cert and newly signed cert back into the keystore [149]. Confluent also provides a shell script that can be used to help automate the following similar commands [150].

Listing 10.3 Importing the CA cert into our truststores and signing our keystore

```
keytool -keystore kafka.broker0.truststore.jks \   ❶
-alias CA -import -file ca.crt  \
-keypass masquerade -storepass masquerade

keytool -keystore kafka.broker0.keystore.jks \   ❷
-alias localhost -certreq -file cert-file \
-storepass masquerade -noprompt

openssl x509 -req -CA ca.crt -CAkey cakey.crt \   ❸
-in cert-file -out signed.crt -days 999 \
-CAcreateserial -passin pass:masquerade

keytool -keystore kafka.broker0.keystore.jks \   ❹
-alias CA -import -file ca.crt \
-storepass masquerade -noprompt

keytool -keystore kafka.broker0.keystore.jks \   ❺
-alias localhost -import -file signed.crt \
-storepass masquerade -noprompt

keytool -keystore kafka.client.truststore.jks \   ❻
-alias CA -import -file ca.crt  \
-keypass masquerade -storepass masquerade
```

❶ Adding our CA to the trusted broker0 truststore

❷ We extract the key from the broker0 keystore we created earlier

❸ We use the CA to sign this broker0 key

❹ We import the CA certificate into our broker0 keystore

❺ We import the signed certificate into the broker broker0 keystore as well

❻ Adding our CA to the client truststore

Listing 10.3 shows the group of commands that are part of adding a CA to a truststore as well as signing our keys.These changes might seem involved, but after running through this process, it is something that you can automate when you are adding more brokers in the future.

> **NOTE** **While running the above commands, your operating system or tool version may have a different prompt than that passed in above. It will likely have a user prompt appear after running your command. Our examples try to avoid these prompts but they might be a way to enter in your passwords without having them remain in your terminal's history.**

As part of our changes, we need to update the server.properties configuration file on each broker as well (broker0 shown only) as shown in Listing 10.4 [151].

Listing 10.4 Broker server properties changes

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093   ❶
ssl.truststore.location=/var/ssl/private/kafka.broker0.truststore.jks   ❷
ssl.truststore.password=masquerade
ssl.keystore.location=/var/ssl/private/kafka.broker0.keystore.jks   ❸
ssl.keystore.password=masquerade
ssl.key.password=masquerade
```

❶ We have added the SSL broker port and left the older PLAINTEXT port

❷ We need to provide the truststore location for our broker as well as the password

❸ We need to provide the keystore location for our broker as well as the password

Changes are also needed for our clients as shown in Listing 10.5. This helps set the protocol used to SSL as well as pointing to our truststore [152].

Listing 10.5 SSL Configuration for Clients

```
security.protocol=SSL   ❶
ssl.truststore.location=/var/private/ssl/client.truststore.jks   ❷
ssl.truststore.password=masquerade   ❸
```

❶ Make our protocol specific to use SSL

❷ The is the file path to the truststore we worked on creating for our client

❸ Since we used a password for our truststore, we reference it here

While testing these changes, we can also have multiple listeners setup for our broker. This also helps clients migrate over time as both ports can serve traffic before we drop the older

PLAINTEXT port option for clients. One of the simplest ways to check our SSL setup is to quickly use the console producers and consumer clients to connect to a topic on our SSL port 9093 as shown in Listing 10.6 [153].

Listing 10.6 Using SSL Configuration for Command-line Clients

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_test_ssl \
  --producer.config custom-ssl.properties   ❶
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic kinaction_test_ssl \
  --consumer.config custom-ssl.properties   ❷
```

❶ Use the producer.config option to let our producer know about the SSL details

❷ Use the consumer.config option to leverage our SSL config for consumers

The `custom-ssl.properties` files help our clients provide the information needed to interact with the broker that is secured now!

One of the nicest features is that we can use the same configuration for both producers and consumers. As you look at the contents of this configuration, one issue that might spring into mind is the use of passwords in these files. The most straightforward option is to make sure that you are aware of the permissions around this file. Limiting the ability to read, as well as the ownership of the file, would be important to note before placing this configuration. Of course, consult your security experts for ever better options that might be available in your environment.

10.1.3 SSL Between Brokers

Another detail to research is since we also have our brokers talking to each other, we might want to determine if we need to use SSL for those interactions. `security.inter.broker.protocol = SSL` should be used in the server properties if we do not want to continue using plaintext for communication between brokers as well as a potential port change [154]. More details can be found at kafka.apache.org/24/documentation.html#security_rolling_upgrade.

10.2 Simple Authentication and Security Layer (SASL)

10.2.1 Kerberos

If you have a security team that already has a Kerberos server, your likely have some security experts to ask for help! When I first started working with Kafka, it was part of a suite of Big Data tools that mostly all leveraged Kerberos. Kerberos is often found in organizations as a method to provide single sign-on (SSO) that is secure. If you have a Kerberos server set up already, you will need to work with a user with access to that Kerberos environment to create a principal for each broker and also for each user (or application id) that will access the cluster. Since this setup might be too involved for local testing, please follow along to see the format of JAAS files.

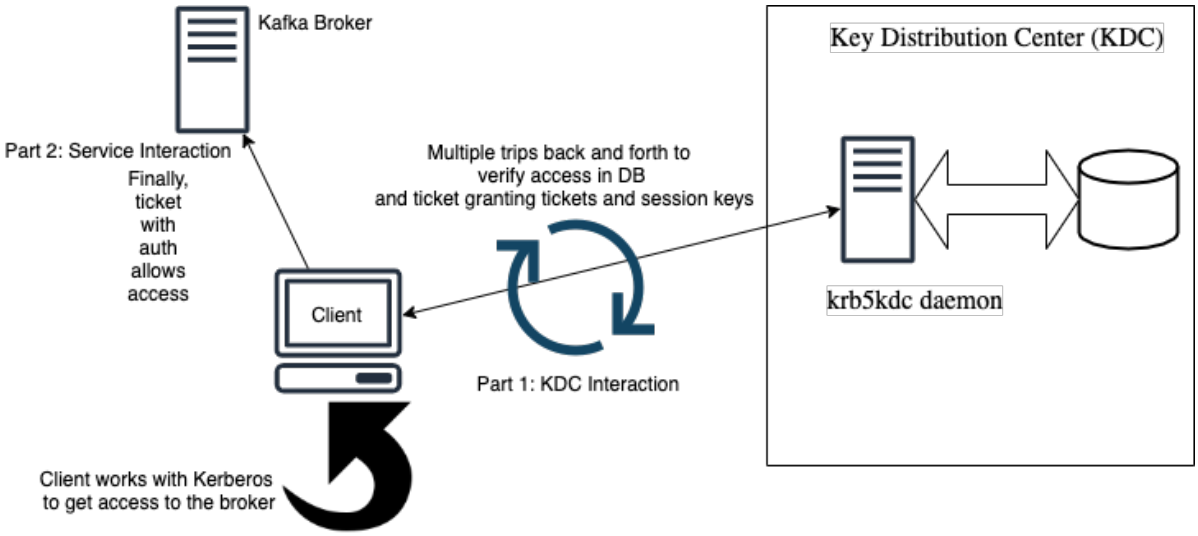

Figure 10.3 Kerberos Overview

Figure 10.3 shows a very simple concept-level view of a Kerberos setup. There are great resources like an overview found at tldp.org/HOWTO/Kerberos-Infrastructure-HOWTO/overview.html if you want to gain more details [155]. One common file type that we will talk about for the brokers and clients are Java Authentication and Authorization Service (JAAS) files. This file will help us provide to Kafka the principal we will use as well as credentials like a keytab file. A keytab will likely be a separate file that has a principal and encrypted keys. This keytab file will be used to authenticate to the Kafka brokers without requiring a password. However, it is important to note that you need to treat your keytab file with the same security and care that you would for any credential.

To get our brokers set up, let's look at some of the server property changes needed as well as an example JAAS configuration. To start, it is important to note that each broker will need its own keytab file. Our JAAS file will help our brokers find the keytab's location on our server as well as declare the principal to use [156].

Listing 10.7 Broker SASL JAAS File

```
KafkaServer {    ❶
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab=”/var/kafka_server0.keytab”
    principal=”kafka/kafka0.kia.manning.com@KAFKAINACTION.ORG”;
};

 -Djava.security.auth.login.config=/var/broker0_jaas.conf    ❷
 -Djava.security.krb5.conf=/var/krb.conf
 -Dsun.security.krb5.debug=true
```

❶ Kafka broker JAAS file

❷ Java parameters need for Kafka server startup

Listing 10.7 shows a JAAS file example and some extra parameters brokers would use on startup.

We are going to add another port as well in order to test out SASL_SSL together before we remove the older ports as shown in Listing 10.8 [157].

Listing 10.8 Broker SASL properties changes

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093,
  <linearrow /> SASL_SSL://localhost:9094    ❶
sasl.kerberos.service.name=kafka    ❷
```

❶ We have added the SASL_SSL broker port and left the older ports

❷ We will use the same primary name for all brokers

Depending on what port you use to connect, the protocol will either be PLAINTEXT, SSL, or SASL_SSL.

The setup from a client is very similar. We add a JAAS file as well as a parameter on start-up as shown in Listing 10.9 [158].

Listing 10.9 Client SASL JAAS File

```
KafkaClient {    ❶
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/var/kafkaclient.keytab"
    principal="kafkaclient@KAFKAINACTION.ORG";
};

-Djava.security.auth.login.config=/var/kafkaclient_jaas.conf    ❷
```

❶ Client SASL JAAS file entry

❷ Java parameter for Kafka server startup to point to the above JAAS file

We also add the configuration for the values if not already included for SASL as shown in listing 10.10 [159].

Listing 10.10 Client property chanes

```
security.protocol=SASL_SSL    ❶
sasl.kerberos.service.name=kafka
ssl.truststore.location=/var/kafka.client.truststore.jks    ❷
ssl.truststore.password=masquerade
ssl.key.password=masquerade
```

❶ Every broker configuration should include the SimpleAclAuthorizer

❷ We add a superuser that will have access to all resources with or without ACLs

The client file is similar to Listing 10.5, but this one defines the SASL_SSL protocol. Make note to make sure the truststore location is correct for your own setup.

After testing that things are not broken on port 9092 or 9093, we can attempt to use our new configuration by validating the same result as before when we use our new SASL_SSL protocol.

10.2.2 HTTP Basic Auth

One common method for developers that use REST APIs or web services in general, is HTTP basic authentication. The reason why we do not discuss this option in-depth is that it is limited to the following components:

- KSQL
- Schema Registry
- Connect REST API
- REST Proxy
- Control Center REST API

Most of these options rely on Java Authentication and Authorization Service (JAAS) configuration files and the creation of a password properties file. Overall, if you have Kerberos already in use, that is probably your most supported option at the time of publication.

10.3 Authorization in Kafka

Now that we have seen how to use authentication with Kafka, let's take a look at how we can start using that information about the user to apply access.

10.3.1 Access Control Lists

As we quickly reviewed, authorization is the process that controls what a user can do. One way to do that is with Access Control Lists (ACLs). While most Linux users are familiar with permissions on a file such as read, write and execute, one drawback is that the permissions are set for an owner, one group, and all others. ACLs can provide permissions for multiple individuals and groups as well as more types of permissions. ACLs are often used when different levels of access for a shared folder are needed [160]. One example is a permission to let a user edit a file but not allow that same user to delete it (delete is a separate permission altogether). Figure 10.4 shows Franz's access.

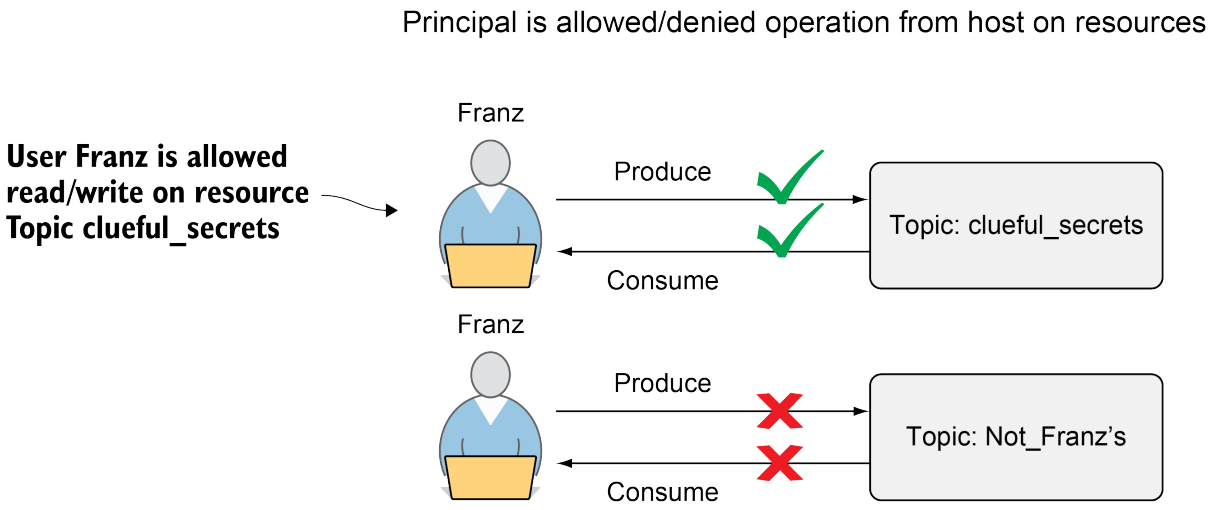

Figure 10.4 ACLs

Kafka designed their authorizer to be pluggable to allow users to make their own logic if desired. Kafka does have a class `SimpleAclAuthorizer` that we will use in our example. With this authorizer, ACLs are stored in ZooKeeper. Brokers asynchronously get this information and cache this metadata in order to make subsequent processing quicker [161].

Listing 10.11 ACL Authorizer and Super Users

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer    ❶
super.users=User:Franz    ❷
```

❶ Every broker configuration should include the SimpleAclAuthorizer

❷ We add a superuser that will have access to all resources with or without ACLs

Listing 10.11 shows adding the authorizer class and super user as Franz to the broker's `server.properties file in order to use ACLs.

The Kafka Authorizer CLI tool is included with the other Kafka scripts in our install. `kafka-acls.sh` allows us to add, delete, or list current ACLs. An important item to note is that

once we configure an Authorizer, ACLs will need to be set or only those considered super users will have access.

Let's figure out how to make only Team Clueful have access to produce and consume from their own topic: clueful_secrets.

For brevity, we will have 2 users in our example team, Franz and Hemingway. Since we already have the keytabs of the users created, we know the principal information that we need.

Listing 10.12 Kafka ACLs To Read and Write to a Topic

```
bin/kafka-acls.sh --authorizer-properties \
  --bootstrap-server localhost:9092 --add \
  --allow-principal User:Franz --allow-principal User:Hemingway \   ❶
  --operation Read --operation Write --topic clueful_secrets   ❷
```

❶ We are identifying the 2 users we want to grant permissions

❷ We allow the named principals to both Read and Write to the specific topic mentioned

As you may have noticed in Listing 10.12, the operation READ is allowing consumers the ability to get data from the topic [162]. The second operation WRITE allows the same principals to produce data into the topic.

While a commercial component of the Confluent Platform, the Confluent LDAP Authorizer is one option to look into if you want to leverage existing Group and/or User principal types.

10.3.2 Role-based access control

Role-based access control (RBAC) is an option that the Confluent Platform supports. RBAC is a way to control access based on roles. Users are then assigned to their role according to their needs such as a job duty. Instead of granting each and every user permissions, with RBAC, you manage the privileges assigned to predefined roles [163]. Figure 10.5 shows how adding a user to a role gives them a new permission assignment.

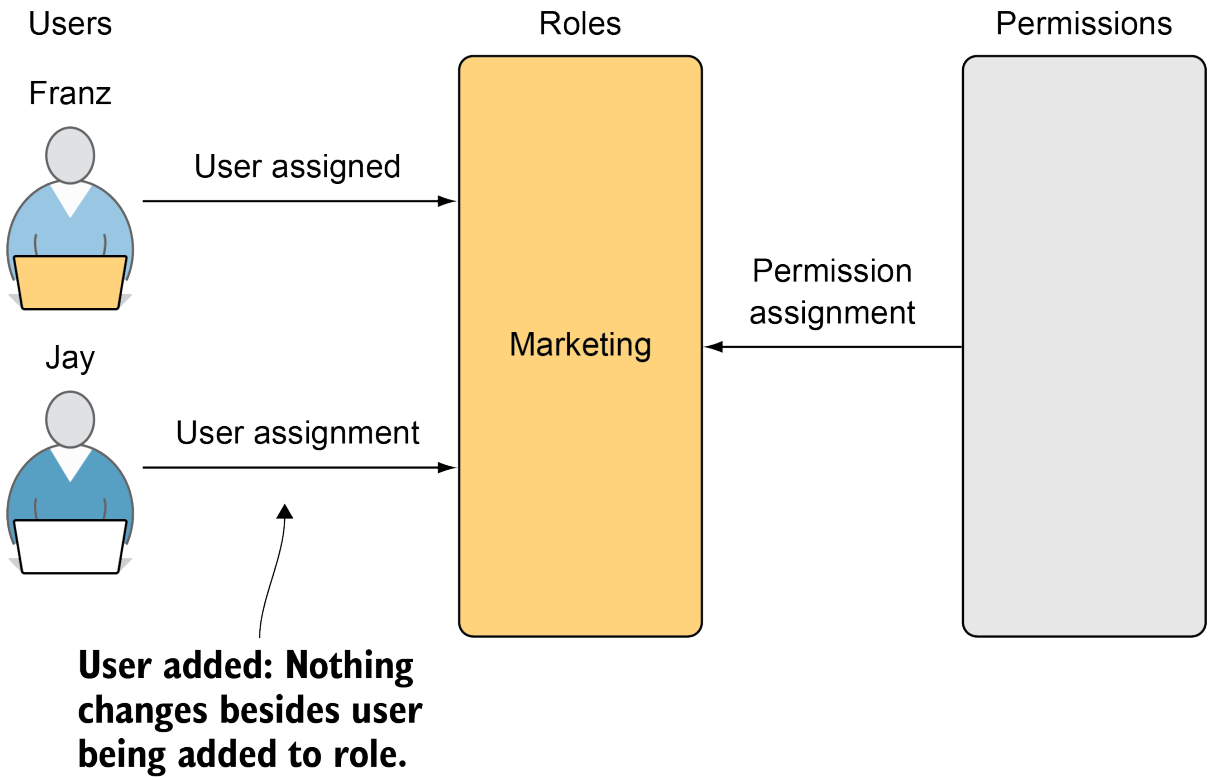

Figure 10.5 Role-based access

RBAC can be used as a layer on top of ACLs that we discussed earlier. One of the biggest reasons to consider both RBAC and ACLS is if you need more granular access control in certain areas. While a user might be able to access a resource like a topic due to RBAC, you could still use an ACL to deny access to specific members of that group.

For our treasure hunting teams, it might make sense to have a specific Role per team. This might mirror how a team in a Marketing area would have a role vs a team from Accounting. If some user moved departments, they would be reassigned their Role and not individual permissions. Since this is a newer option that might change as it matures and is limited to a Confluent Platform environment, the option is mentioned for awareness even though we will not dig further into it here.

10.4 ZooKeeper

Part of securing Kafka is looking at how we can secure all parts of our cluster: including ZooKeeper. Not only can someone really miss up our cluster by modifying broker data stored in ZooKeeper, but another concern is the ACLs themselves. If we protect the brokers but not the system that holds that security-related data, it would be possible for those with knowledge to update those accesses without much effort. While the data in ZooKeeper is not usually sensitive and znodes can be read by anyone, we are trying to prevent changes to the znodes.

We will need to set the value `zookeeper.set.acl` to true per broker as shown in Listing 10.3 [164].

Listing 10.13 ACL ZooKeeper

```
zookeeper.set.acl=true    (1)
```

(1) Every broker configuration would include this ZooKeeper related value

Without this setting (or set to false), ACLs would not be created. Note that these ACLs are ZooKeeper specifc, not the ACLs we talked about earlier which applied to Kafka reources only [165].

10.4.1 Kerberos Setup

Making sure that ZooKeeper works with Kerberos requires a variety of configuration changes.

For one, in the zookeeper.properties configuration we will add values that let ZooKeeper know that SASL should be used for clients and which provider should be used and would likely look similar to those shown in Listing 10.14 [166].

Listing 10.14 ZooKeeper properties updates for Kerberos

```
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider    (1)
requireClientAuthScheme=sasl
jaasLoginRenew=3600000
```

(1) SASLAuthenticationProvider allows us to enable the use of SASL

Start-up parameters and JAAS files would also be created at this time. Refer to docs.confluent.io/platform/current/security/zk-security.html for more details if needed [167].

While we were busy looking at the other options for setup so far in this chapter, some of the users on our system were still up to no good. Let's see if we can dig into the subject of quotas to help.

10.5 Quotas

Let's say that users of our web application start to notice that they don't have any issues with trying over and over to request data. While this is often a good thing for end-users who want to be able to use a service as much as they want without their progress being limited, the cluster might need some protection from users who might use that to their advantage. In our instance, since we made it so the data was locked down to members of our team only, some users have thought of a way to try to prevent others from using the system. In effect, they are trying to use a distributed denial-of-service (DDoS) attack against our system. A targeted attack against our cluster can overwhelm our brokers and their surrounding infrastructure. In practice, the other teams are requesting reads from their topics over and over while reading from the beginning. We can use quotas to prevent this behavior. One detail to know is that quotas are defined on a per-broker basis. The cluster does not look across each broker to calculate a total, so a per-broker definition is needed.

Our quota metadata is written to ZooKeeper which allows the brokers to get this updated information quickly. This update path is also a benefit in that we do not have to restart our brokers in order for this new configuration to take effect. The ability to react quickly to our quota changes is a plus if we are addressing clients that are suddenly a problem in our ecosystem. Also, by default, quotas are unlimited. While there is a way to set producer and consumer defaults with the keys: `quota.producer.default` and `quota.consumer.default` this technique is noted in the project documentation as being deprecated [168]. Figure 10.6 shows an example of using a request percentage quota.

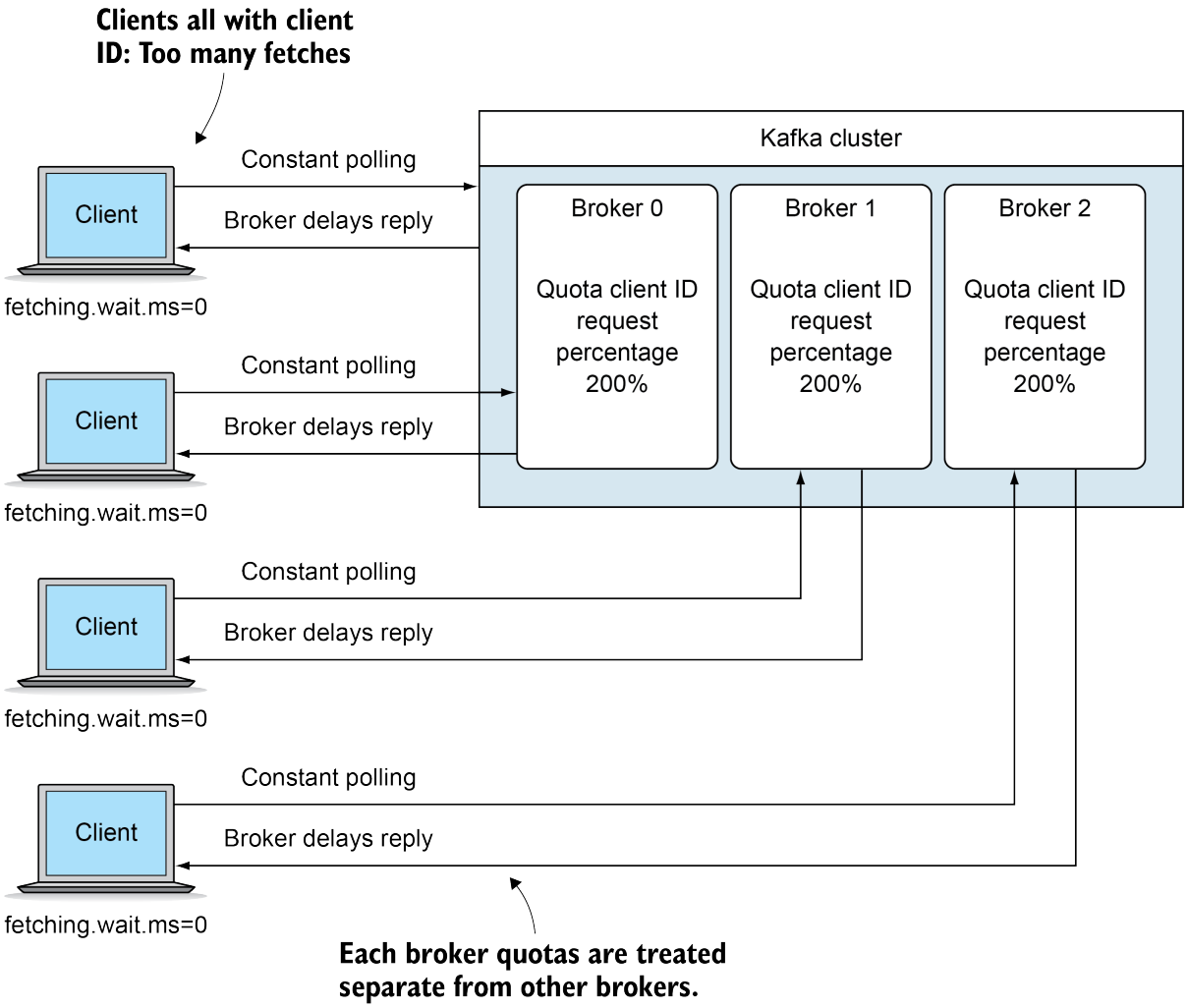

Figure 10.6 Quotas

To set our own custom quotas, we need to know how to identify the "who" to limit and the limit we want to set. Whether or not we have security impacts what options we have for defining who we are limiting. Without security, we are able to use the `client.id` property. With security enabled, we can also add the `user`, and thus, any `user` and `client.id` combinations as well.

There are a couple of types of quotas that we can look at defining for our clients: network bandwidth and request rate quotas. Let's take a look at the network bandwidth option first.

10.5.1 Network Bandwidth Quota

Network bandwidth is measured by the number of bytes per second [169]. In our example, we want to make sure that each client is respecting the network and not flooding it to prevent others from using it. Each user in our competition uses a client id that is specific to their team for any producer or consumer requests from their clients. In our example in Listing 10.15, we are going to limit the clients using the client id `clueful` by setting a `producer_byte_rate` and a `consumer_byte_rate` [170].

Listing 10.15 Creating a Network Bandwidth Quota for Client Clueful

```
bin/kafka-configs.sh  --bootstrap-server localhost:9092 --alter \
  --add-config 'producer_byte_rate=1048576,consumer_byte_rate=5242880' \   ❶
  --entity-type clients --entity-name clueful   ❷
```

❶ We are allowing producers 1 MB per second and Consumers 5 MB per second

❷ The entity-name for a client-type is our client.id clueful

The `add-config` parameter is used to set both the producer and consumer rate. The `entity-name` applies the rule to our specific clients `clueful`.

As is often the case, we might need to list our current quotas as well as delete them if they are no longer needed. All of these commands can be completed by sending different arguments to the `kafka-configs.sh` script as shown in Listing 10.16 [171].

Listing 10.16 Listing and Deleting a Quota for Client Clueful

```
bin/kafka-configs.sh  --bootstrap-server localhost:9092 \
  --describe \   ❶
  --entity-type clients --entity-name clueful

bin/kafka-configs.sh  --bootstrap-server localhost:9092 --alter \
  --delete-config 'producer_byte_rate,consumer_byte_rate' \   ❷
  --entity-type clients --entity-name clueful
```

❶ We are listing the existing configuration of our client.id

❷ We are using the delete-config option to remove those we just added

The `describe` helps us get a look at the existing configuration. We can then use that information to decide if we need to modify or even delete the config as shown by using the `delete-config` parameter.

As we start to add quotas we might end up with more than one quota applied to a client. We will need to be aware of the precedence in which various quotas are applied. While it might seem like the most restrictive (lowest bytes allowed) of the quotas would be applied, that is not always the case. The following is the order in which quotas are applied with the highest precedence listed at the top [172]:

- User and client.id provided quotas
- User quotas
- Client.id quotas

For example, if a user named Franz had a user quota limit of 10 MB and a client.id limit of 1 MB, the consumer he used would be allowed 10 MB per second due to the User-defined quota having higher precedence.

10.5.2 Request Rate Quotas

The other quota to examine is on request rate. Why the need for a second quota? While a DDoS attack is often thought of as a network issue, clients making lots of connections could still overwhelm the broker by making CPU-intensive requests (anything related to SSL or message compressed/decompression are good examples). Consumer clients that poll continuously with a setting of `fetch.max.wait.ms=0` are also a concern that can be addressed with this quota [173].

The simplest way to think about this quota is that it represents the total percentage of CPU that a group of related clients is allowed to use. To set this quota, we use the same entity types and add-config options as we did with our other quotas. The biggest difference is setting the configuration for `request_percentage`. A formula that we can use is the number of I/O threads (num.io.threads) + the number of network threads (num.network.threads) * 100% and was listed in the KIP-124 - Request rate quotas wiki [174]. For our example in Listing 10.17, we are setting a request percentage of 100. This is setting each thread quota at .5.

Listing 10.17 Creating a Network Bandwidth Quota for Client Clueful

```
bin/kafka-configs.sh  --bootstrap-server localhost:9092 --alter \
  --add-config 'request_percentage=100' \   ❶
  --entity-type clients --entity-name clueful   ❷
```

❶ We are allowing producers 100 request percentage. (.5 + .5) * 100 = 100

❷ The entity-name for a client-type is our client.id clueful

Using quotas is a good way to protect our cluster as well as react to clients that suddenly might start putting a strain on our brokers.

10.6 Data at Rest

Another thing to consider is whether you need to encrypt the data that Kafka writes to disk. By default, Kafka does not encrypt the events it adds to the log. There have been a couple of Kafka Improvement Proposals that have looked at this feature, but at the time of publication, you will still need to make sure you have a strategy that meets your requirements. One of the first attempts at this feature could be volume encryption. However, those that have access to the broker file system, like OS administrators, might then have access to all of the topic data. Depending on your business needs as well, you might want to only encrypt specific topics or even specific topics with unique keys.

10.6.1 Managed Options

Of course, if you are using a managed option for your cluster, it might be good to check out what features the service provides. Amazon's Managed Streaming for Apache Kafka (aws.amazon.com/msk/) is one example of a cloud provider that handles a large part of your cluster management including some security pieces as well. Having your brokers and ZooKeeper nodes updated with automatically deployed hardware patches and related updates addresses one major source of keeping issues at bay. The other benefit of these updates is that you are not providing access to even more administrators to your cluster. Amazon MSK also provides encryption for your data at rest using either the AWS Key Management Service (KMS) Customer Master Key (CMK) or your own CMK. Additional features that we covered in our examples are the ability to use SSL between your clients and cluster and ACLs.

Confluent Cloud (www.confluent.io/confluent-cloud/) also is an option that can be deployed across various public cloud offerings. Support for Data encryption at rest & in-motion as well as ACL support matches options that you should be aware of when matching your security requirements to the actual provider.

Also sticking with the Confluent stack, Confluent Platform 5.3 has a commercial feature called Secret Protection (docs.confluent.io/platform/current/security/secrets-tutorial.html). When we looked at our SSL configuration files earlier, we had clear text passwords in certain files. However, Secret Protection is meant to address that issue by encrypting the secrets in the file and keeping those exposed values out of log files as well. Since this is a commercial offering, I will not go into depth on how it works, but just be aware there are options available.

10.7 Summary

- PLAINTEXT, while fine for prototypes, needs to be evaluated before production usage. SSL can help protect your data between clients and brokers and even between brokers.
- Kerberos can be used to provide a principal identity and might allow you to leverage existing Kerberos environments that exist in an infrastructure already.
- Access control lists (ACLs) help define which users have specific operations granted. Role-based access control (RBAC) is also an option that the Confluent Platform supports.
- Quotas can be used with network bandwidth and request rate limits to protect the available resources of a cluster. These quotas can be changed and fine-tuned to allow for normal workloads and peak demand over time.

11 Schema registry

This chapters covers

- Developing an ideal Kafka Maturity Model
- Focusing on the value schemas can provide for your data as it changes
- Reviewing Avro and its use in data serialization
- Determining compatibility rules for schema changes over time

As we have discovered the various ways to use Apache Kafka, it might be an interesting experiment to think through how you view Kafka the more you utilize it.

As enterprises (or even tools) grow, they can sometimes be modeled with maturity levels. Martin Fowler provides a great explanation at this site: martinfowler.com/bliki/MaturityModel.html [175]. Fowler also has a clear example of explaining the Richardson Maturity Model which looks at REST in such a model [176]. For even furher reference, the original talk writeup, "Justice Will Take Us Millions Of Intricate Moves: Act Three: The Maturity Heuristic" by Leonard Richardson can by found at www.crummy.com/writing/speaking/2008-QCon/act3.html [177]. The following model is our opinions for maturity levels specific to Kafka. For a comparison to a different experienced perspective, check out the Confluent whitepaper titled, "Five Stages to Streaming Platform Adoption", which has five stages of their streaming maturity model with different criteria for each stage [178].

Let's look at our first level: of course, as programmers we're starting with **Level 0**.

- **Level 0:** Kafka is used as an enterprise service bus (ESB) or publish/subscribe system. We use events to provide asynchronous communication between applications, whether we are replacing a different message broker like RabbitMQ or just starting with this pattern. One example use case is a user submitting a text document to be converted into a pdf. Once a user submits a document, the application stores the document and then sends a message to a Kafka topic. A Kafka consumer then reads these messages to determine which documents need to be converted into a pdf. In this example, the drive might be to offload work to a backend system that a user knows will not send a response right away. Figure 11.1 shows this message bus in action.

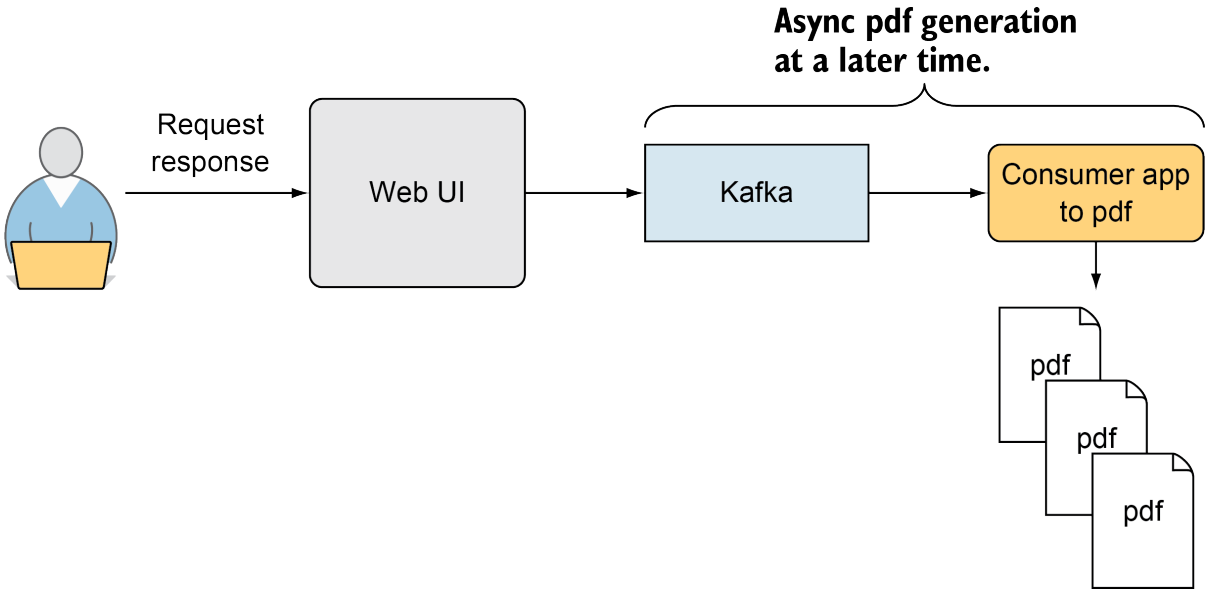

Figure 11.1 Level 0 Example

This level alone brings us the benefit of allowing us to decouple a system so that a failure of our front-end text submission system would not impact our backend system. Also we don't need to rely on both to maintain operations at the same time to be successful.

- **Level 1:** Batch processing may still be present in areas of our enterprise-but most data produced is now being brought into Kafka.

  Whether with extract, transform, load (ETL), or change data capture (CDC) processes, Kafka is starting to gather events from more and more systems in our enterprise. An example of this might be a vendor database that holds customer information. We do not want our marketing folks to run complex queries that could slow down our production traffic. In our case, we use Kafka Connect to write the data from those tables into Kafka topics that we can leverage on our terms.

  Figure 11.2 shows Kafka Connect capturing data from a relational database and moving that data into a Kafka topic.

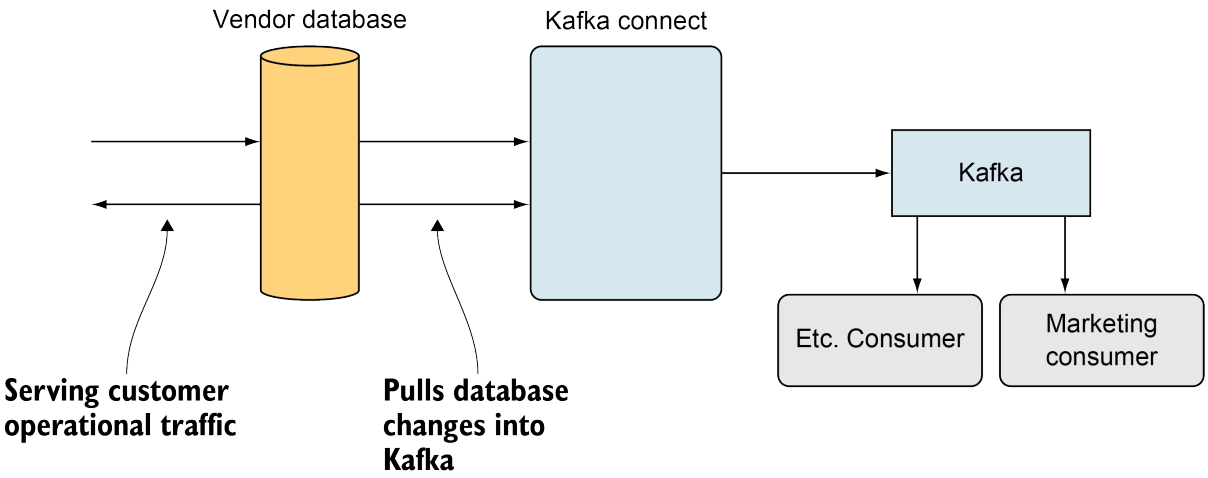

Figure 11.2 Level 1 Example

Level 1 allows us to have an operational, real-time data flow and the ability to feed data quickly into analytical systems.

- **Level 2:** We realize that data will change over time and that schemas are needed.
  Although our producers and consumers might be decoupled, they still need a way to understand the data itself. For this we'll take advantage of schemas and a schema registry. And even though it would have been ideal to start with schemas, the reality is the need often presents itself a couple of application changes later after initial deployments. One example is changing the data structure of an event we used to receive from our order processing system. New data is added, but the new fields are optional and this works fine since our schema registry is configured to support backward compatibility. Figure 11.3 shows our Kafka clients referencing a schema registry. We will look more into these details as we progress through chapter 11.

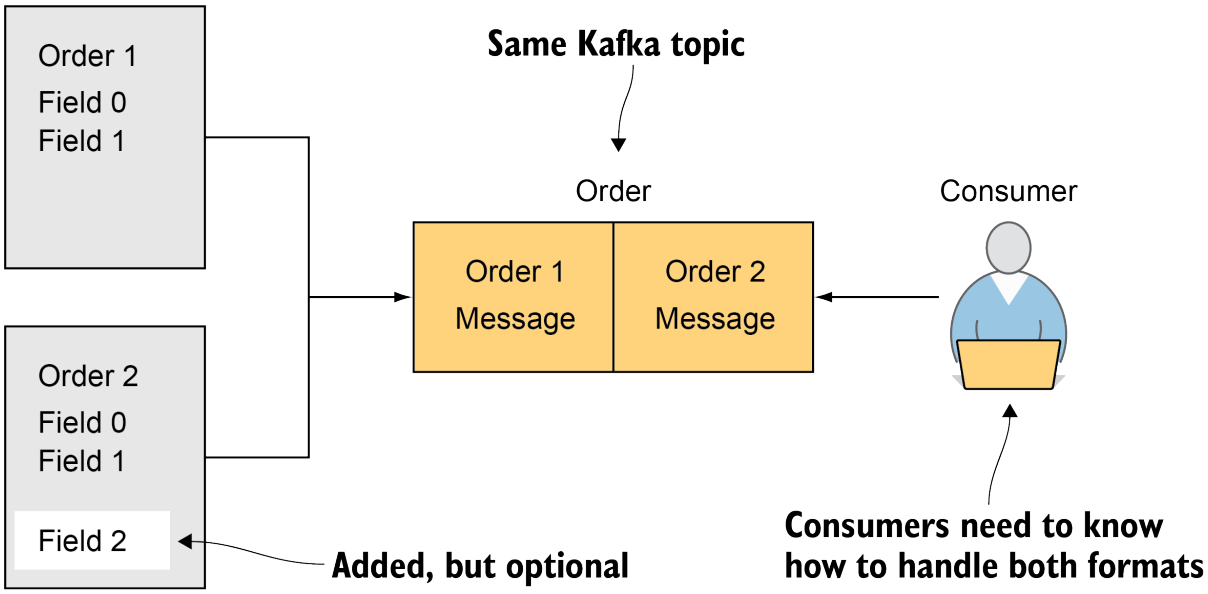

Figure 11.3 Level 2 Example

- **Level 3:** Everything is an event stream.
  Events are infinite and will not end. Kafka is the central data system of our enterprise for our event-based applications. In other words, we don't have customers waiting for recommendations or status reports that used to be produced by an overnight batch

processing run. Customers are alerted in milliseconds of a change to their account when an event happens, not in minutes. Instead of pulling data from other data sources, applications are producing data to Kafka. User-facing applications can derive state and materialized views to customers depending on our core Kafka infrastructure needs. Figure 11.4 shows an example Kafka streams application.

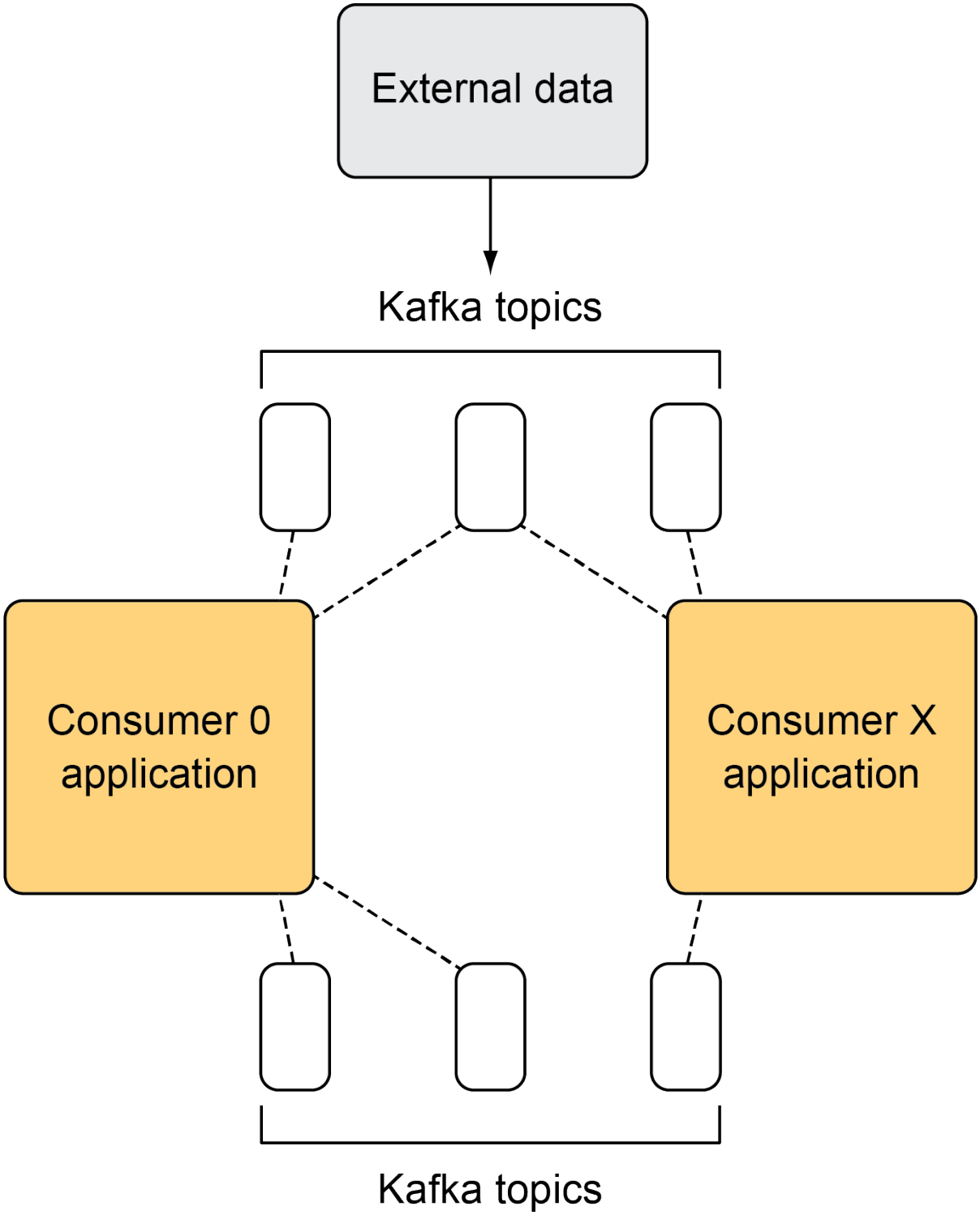

Figure 11.4 Level 3 example

This exercise was used to think about how Kafka can be a powerful tool for one application or even evolve into the foundation of all of your enterprise's applications rather than as a simple message broker alone. This isn't meant to be a step-by-step required path, but rather a way to think about how you might start with and progress with Kafka. These steps are debatable of course but offer an example path.

11.1 Schema Registry

As part of our work in this chapter, we will focus on Level 2, looking at how we can plan for data to change over time. We have become very good at sending data into and out of Kafka. However, despite a small mention of schemas in chapter 3, we left out some important details. Let's dive into what the schema registry provides for us.

First, it might be helpful to start with a comparison of schemas and how they compare with APIs. APIs can act as a contract between two different applications to determine how both sides will interact. Schemas can be thought of similarly, except as the contract for our producer and consumer clients [179].

The schema registry stores our named schemas and allows us to maintain multiple versions. This is somewhat analogous to the Docker registry, which stores and distributes Docker images. Why is this storage needed? Producers and consumers are not tied together, but they still need a way to discover the schema involved in the data from all clients. Also, by having a remotely hosted registry, users do not have to run their copy locally or attempt to build their own based on a list of schemas.

While schemas can provide a contract for applications, we can also leverage them to prevent breaking changes. Why should we care about data that is moving fast throughout our system? Kafka's storage and retention capabilities make it so that consumers can go back to process older messages. These messages might be from months ago, and our consumers would need to handle these various versions of the data.

Another added benefit of using a registry is that the schemas do not have to be stored with every message. In other words, we could avoid using a registry by adding the entire schema itself with every message we receive. By not doing this, however, we can save significant memory and disk space. The same schemas are usually used by many messages, and the desire is not to replicate the storage.

For Kafka, we can leverage the Confluent Schema Registry. Confluent provides an excellent option to consider as we look into how to take advantage of schemas. If you installed Kafka via Confluent Platform before this chapter, you should have all of the tools available to explore further.

11.1.1 Installing Schema Registry

The Schema Registry is a community software offering under the Data Compatibility section of the Confluent Platform and does not currently require a commercial license at this time. The registry itself uses Kafka as its storage layer. Just as with offset storage, the topic is named with an underscore, in this case `_schemas`, and is considered an internal topic [180]. It is vital not to delete this topic accidentally!

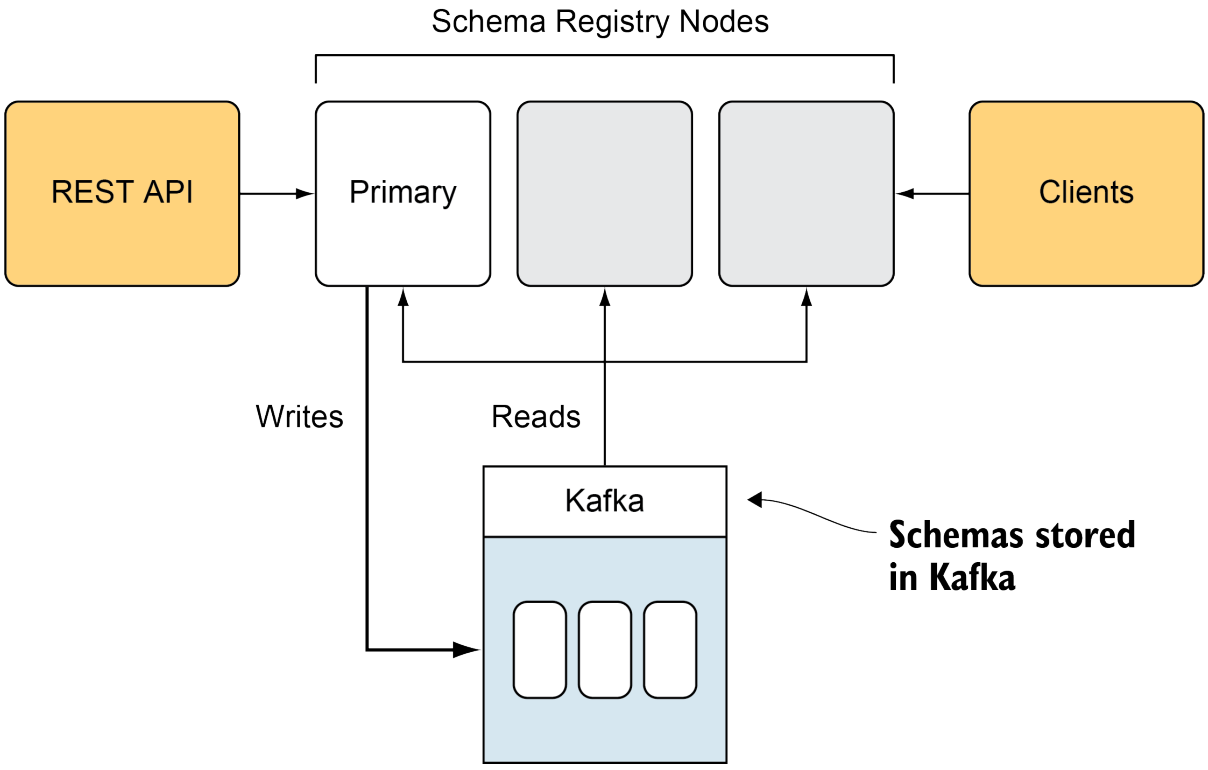

Figure 11.5 Schema Registry infrastructure

When thinking about production usage, the schema registry should be hosted on a separate server from your brokers as Figure 11.5 shows. Since we deal with a distributed system and have learned to expect failures, you can provide multiple registry instances. Since all nodes can handle lookup requests from clients and route write requests to the primary node, the clients of the registry do not have to maintain a list of specific nodes but can provide a VIP for the cluster.

11.1.2 Registry configuration

Similar to the other components of Kafka, you can set several configuration parameters in a file. If you have installed Kafka on your disk, you can see the defaults located at `etc/schema-registry/schema-registry.properties`.

For the registry to be successful, it will need to know which topic to store its schemas in and how to work with your specific Kafka cluster. In this example, we are using ZooKeeper to help complete the election of the primary node. Important to note that since only the primary node

writes to the Kafka topic. If your team is trying to move away from ZooKeeper dependencies, we also can use Kafka-based primary election (using the configuration `kafkastore.bootstrap.servers`) [181].

Listing 11.1 Important Schema Registry configuration

```
listeners=http://localhost:8081      ❶

kafkastore.connection.url=localhost:2181      ❷

kafkastore.topic=_schemas      ❸

debug=true      ❹
```

❶ We are going to serve our registry at 8081.

❷ We are pointing to our Kafka bootstrap server.

❸ The default topic for schema storage is used - but we could change it if needed.

❹ This debug flag can be flipped to get or remove extra error information.

Let's go ahead and start up the schema registry. We want to make sure that our ZooKeeper and Kafka brokers are already started for our examples. After confirming that those are up and running, we can use the command line to run the starting script for the registry [182].

Listing 11.2 Starting Schema Registry

```
bin/schema-registry-start.sh \      ❶
  ./etc/schema-registry/schema-registry.properties      ❷
```

❶ Run the schema-registry-start script as part of the install's `bin` directory.

❷ The script takes in a properties file that we can modify.

We can check that the process is still running or use `jps` to verify as this is a Java application just like the brokers and ZooKeeper. Now that we have the registry running, we need to look at how to use the components of the system. Since we have a place to store our data format, let's revisit a schema that we used in chapter 3.

11.2 Defining a schema

In our context, a schema defines the structure of the data. We will be using Avro for our schemas as we did in earlier chapters. It is important to note that there is no rule that the topic name matches the schema name. Another term you might see in documentation is `subject`. The subject is the name with which the schema registry uses to refer to a specific schema [183]. Practically, I like to think of the subject as a generic bucket. When you update a schema, the same subject is still used, but a new version and ID are created for your specific schema. Compatibility checks are also scoped to the subject level. As for naming the subject, the default is to use the topic name as the start of the subject name.

Let's look at an example of a topic and its related subject for an application using a topic named `clues` in the spirit of our scavenger hunt from chapter 10. Our messages will use a schema named `Evidence.` In our schema registry, we would have a subject called `clues-value` as we use the default of basing the name off of our current topic name. If we are using a schema for the message key as well, we would also have a subject called `clues-key.` Notice that the key and value are treated as different subjects. Why is this needed? It ensures that we can independently version and change our schemas since the key and value are serialized separately.

11.2.1 A new schema

Listing 11.3 shows the proposed Avro schema for our Alert class that we were using in chapter 3. Since we are just starting our project, we only have four fields: sensor_id, time, status, and alert_status. We assume our format might change as we learn more about what our troubleshooting teams would find helpful, so starting with a schema is the right approach.

Listing 11.3 Alert Schema: alert.avsc

```
{"namespace": "org.kafkainaction",   ❶
 "type": "record",
 "name": "Alert",   ❷
 "fields": [   ❸
     {"name": "sensor_id",  "type": "long",
       "doc":"The unique id that identifies the sensor"},
     {"name": "time", "type": "long",
       "doc":"Time the alert was generated as UTC milliseconds from the epoch"},
     {"name":"status",
     "type":{"type":"enum",
         "name":"alert_status",
             "symbols":["Critical","Major","Minor","Warning"]},
     "doc":"The allowed values that our sensors will use to emit current status"}
 ]
}
```

❶ A namespace will be generated.

❷ Alert will be the name of the Java class we will interact with.

❸ The fields we want in our data alone with their data type and documentation notes.

Overall, the schema will be what our data structure looks like for our clients as they produce and read the messages from Kafka.

11.3 Schema features

The Confluent Schema Registry contains two important components:

- A REST API (and the underlying application) for storing and fetching schemas.
- Client libraries for retrieving and managing a local cache of schemas.

Let's look at each of these two components starting with the REST API.

11.3.1 REST API

The REST API is there to help us manage the following resources related to our schemas in general [184]:

- Schemas - The ability to retrieve a schema by ID is provided.
- Subjects - Schemas are registered under a specified subject. We can create, retrieve, and delete versions as well as subjects themselves.
- Compatibility - The ability to test schemas for compatibility is presented with the resource. We will go into more detail concerning the types of compatibility later in this chapter.
- Config - Provides the ability to retrieve and update the global and subject-level configuration for our cluster.

To confirm the registry is started, we are going to submit a GET against the REST API to see it in action using a tool like `curl`. In this case, we'll use it to list the current configuration.

Listing 11.4 Listing Schema Registry Configuration

```
curl -X GET http://localhost:8081/config      ❶
```

❶ List all of the configs in the registry using REST

Also, we should add a `Content-Type` header for our REST interactions with the schema registry. In the following examples, I will be using `application/vnd.schemaregistry.v1+json`. The content type tells us that we want to use `v1` as the API version. Also, JSON is the serialization format for the request/response. As we do with the schemas themselves, we're planning for API changes by declaring which API version we are using. This helps ensure that our clients are using the version we intended.

While the REST API is great for administrators of the subjects and schemas, the client library is where most developers will spend their time interacting with the registry.

11.3.2 Client library

Let's drill into the producer client's interaction with the schema registry. As a reminder, we are going to focus on the Java clients in our examples. Let's say that we have a producer that is configured to use an Avro serializer for our messages. The code within the serializer itself calls out to the registry to find the schema it needs. In the case where the schema exists, the client acquires an ID from the registry. Any messages that use that same schema will store the message and the ID together. Caching is involved for already seen schemas, so our clients do not have to connect to the registry for every message.

We should already have a registry started locally, so now we need to configure our producer client to use it. With our use case from chapter 3, we had created a schema for an Alert that would be the value of our message. The `value.serializer` property needs to be set to use the KafkaAvroSerializer in our case. This class knows how to serialize the custom object using the registry.

Listing 11.5 Producer using Avro Serialization

```
...
props.put("key.serializer","org.apache.kafka.common.serialization.LongSerializer");
props.put("value.serializer","io.confluent.kafka.serializers.KafkaAvroSerializer");   ❶
props.put("schema.registry.url", "http://localhost:8081");   ❷

Producer<Long, Alert> producer = new KafkaProducer<Long, Alert>(props);
Alert alert = new Alert();   ❸
alert.setSensorId(12345L);
alert.setTime(Calendar.getInstance().getTimeInMillis());
alert.setStatus(alert_status.Critical);
log.info(alert.toString());

ProducerRecord<Long, Alert> producerRecord = new ProducerRecord<Long, Alert>("avrotest",
                                                                alert.getSensorId(), alert);   ❹

producer.send(producerRecord);
```

❶ We are sending the Alert as a value and using the `KafkaAvroSerializer`.

❷ This property points to the URL of our registry, which has a versioned history of schemas and will help with schema validation and evolution.

❸ Creating an alert to show using the Java classes generated from the schema.

❹ Notice the generic is now Alert instead of a string.

> **NOTE** **Since the default `TopicNameStrategy` is used, the schema registry should have a subject: `avrotest-value` with our schema for Alert registered [185]. To use a different strategy, the producer client could set the following configuration to override the value (and key) strategy: `value.subject.name.strategy` and/or `key.subject.name.strategy`.**

On the consumer side, the client will perform a lookup using the ID stored with the message

against the registry. The client can again cache the schema to avoid repeated calls to the schema registry. Once the client has successfully found the schema, it can now deserialize the messages it reads. Let's look at using the same schema we produced to a topic and retrieve it with a consumer to see if we can get that value back without error [186].

Listing 11.6 Consumer using Avro Deserialization

```
props.put("key.deserializer","org.apache.kafka.common.serialization.LongDeserializer");
props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");   ❶
props.put("schema.registry.url", "http://localhost:8081");   ❷
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);

KafkaConsumer<Long, Alert> consumer = new KafkaConsumer<Long, Alert>(props); //C   ❸

consumer.subscribe(List.of("avrotest"));   ❹

while (keepConsuming) {
  ConsumerRecords<Long, Alert> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<Long, Alert>  record : records) {
      log.info("Alert Content = {}, record.value().toString());   ❺
    }
}
```

❶ We use the `KafkaAvroDeserializer` class in consumer config.

❷ This property points to the URL of our registry, which has a versioned history of schemas and will help with schema validation and evolution.

❸ Creating our consumer as standard with our configuration using the Alert type to show using the Java classes generated from our schema.

❹ We are subscribing to the same topic that we produced our schema messages to

❺ We should be able to have our Alert object successfully deserialized!

So far, we have worked on only one version of a schema with our producer and consumer clients. However, planning for data changes can save you a lot of headaches down the line. Next, we are going to look at some of the rules that will help us think about the changes we can make and their impact on our clients.

11.4 Compatibility rules

One important thing to decide is what compatibility strategy we plan to support. These compatibility rules are here to help guide our schemas as they change over time. While it may seem like a large number of types available, it is nice to know that, in general, those marked as transitive follow the same rules as those without that suffix. However, the non-transitive types are only checked against the last version of the schema, whereas transitive types are checked against all previous versions. The only type that stands out is the NONE type. Since compatibility is not really checked and all changes are accepted, you are in a situation where you could have incompatible changes.

Here is a list of types noted by Confluent [187]:

- BACKWARD - This is the default type.
- BACKWARD_TRANSITIVE
- FORWARD
- FORWARD_TRANSITIVE
- FULL
- FULL_TRANSITIVE
- NONE

Let's look at what the BACKWARD type implies for our applications. Backward compatible changes might involve adding non-required fields or removing fields. Another critical aspect to consider when choosing the compatibility type is the order in which we want clients to change. For example, we will likely want our consumer clients to upgrade first for the `BACKWARD` type. Consumers will need to know how to read the messages before new variations are produced. Using an older schema on reading when a newer version was used to produce messages might leave our application in a mess if we do not update in this order.

On the reverse end of the types, `FORWARD` compatible changes are the opposite of `BACKWARD`: adding new fields or removing non-required fields. Keeping up with the reverse theme, we will likely want to update our producer clients first.

As you can reason by now, `FULL` means that changes both backward and forward compatible. Since clients can read and write messages with old or new schemas, our clients can be updated in any order. NONE in effect means that schema compatibility checks are not done. The work is entirely up to us to determine if schema changes will break clients. To pull changes off in this type, we will likely want to update both the producer and consumer clients at the same time or even migrate to a new topic entirely.

Let's look at how we can change our schema for `Alert` to maintain backward compatibility. Listing 11.7 shows the addition of a new field, `recovery_details`. A default value was also used with this new field.

Listing 11.7 Alert schema change

```
{"namespace": "org.kafkainaction",
 "type": "record",
 "name": "Alert",
 "fields": [
     {"name": "sensor_id",  "type": "long",
       "doc":"The unique id that identifies the sensor"},
     {"name": "time", "type": "long",
       "doc":"Time the alert was generated as UTC milliseconds from the epoch"},
     {"name":"status",
     "type":{"type":"enum",
         "name":"alert_status",
             "symbols":["Critical","Major","Minor","Warning"]},
     "doc":"The allowed values that our sensors will use to emit current status"},
    {"name": "recovery_details", "type": "string", "default": "Analyst recovery needed"}   ❶
 ]
}
```

❶ New field, `recovery_details`

Adding a new field with a default value should allow an updated consumer to read data produced with the old and new schema. Any older messages with version 1 of the schema will have a default value populated for the field added later and read by a consumer using schema version 2.

11.4.1 Validate schema modifications

If we have tests that exercise our API endpoints or even Swagger (swagger.io/), it is important to think about how we can automate testing changes to our schemas.

To check and validate our schema changes, we have a couple of options: * Use the REST API Compatibility resource endpoints * For JVM-based applications, using a Maven plugin.

Let's look at an example REST call that will help us check our compatibility for a schema change in Listing 11.8. [188]

Listing 11.8 Checking Compatibility with the Schema Registry REST API

```
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \   ❶
  --data '{"schema": "{\"type\": \"string\"}"}' \   ❷
  http://localhost:8081/compatibility/subjects/topics-value/versions/latest   ❸


{"is_compatible":true}
```

❶ We are using a custom `content-type` in our header.

❷ We are passing the schema content on the command line.

❸ We are listing the `topics-value` subject lastest version to check our new schema against.

We can also use a Maven plugin if we are willing to use Maven and already on a JVM-based

platform [189]. Listing 11.9 shows the `pom.xml` entry needed.

Listing 11.9 Checking Compatibility with the Schema Registry Maven plugin

```
<plugin>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-schema-registry-maven-plugin</artifactId>   ❶
    <version>6.1.1</version>
    <configuration>
        <schemaRegistryUrls>
            <param>http://localhost:8081</param>   ❷
        </schemaRegistryUrls>
        <subjects>   ❸
            <Topics-key>src/main/avro/Topics-Key.avsc</Topics-key>
            <Topics-value>src/main/avro/Topics-Value.avsc</Topics-value>
        </subjects>
    </configuration>
    <goals>
        <goal>test-compatibility</goal>   ❹
    </goals>
</plugin>
```

❶ Coordinates that Maven will need to download this plugin.

❷ The URL to our schema registry.

❸ We are listing the topics-value and key subjects we want to validate with our new schemas at the provided file path.

❹ The maven goal can be invoked with the following: `mvn schema-registry:test-compatibility`

If you already have a build pipeline, you can run this goal as an additional step in your build. In essence, it takes the schemas located in your file path and connects to your schema registry to check against the schemas already stored.

11.5 Alternative to a Schema Registry

Since not all projects start with schemas or with data changes in mind, there are some simple steps that we can take to work around data format changes. One such option is to produce data on a different topic with a breaking change. After consumers have consumed the old format, they can be updated if needed and then start to read from another topic. This works if we do not have a long retention period and do not reprocess our data.

11.5.1 Create new topics

Suppose we do plan on reprocessing the data across formats. In that case, we could also create a new topic that exists to hold the transformed topic messages that existed in the initial topic and, of course, the new messages streaming in as shown in Figure 11.6. Kafka Streams, which we discuss in chapter 12, can help in this topic-to-topic transformation.

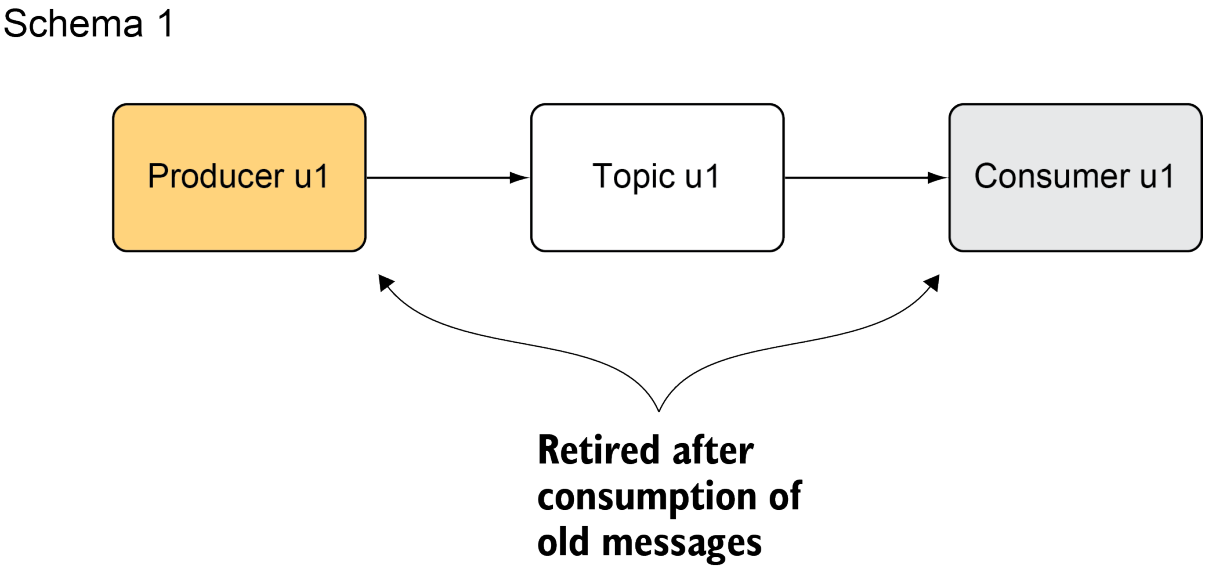

Figure 11.6 Alternative stream

11.5.2 Confluent Cloud Schema Registry

If your team does not have the operations staff or is running primarily in the cloud already, then Confluent Cloud Schema Registry (docs.confluent.io/cloud/current/cp-component/schema-reg-cloud-config.html) is another option to look into. In this setup, schemas are stored in Confluent managed Kafka clusters versus having that data in your own Kafka clusters.

11.6 Summary

- Kafka has many features and can be used for simple use cases all the way up to being the central system of an enterprise.
- Schemas can be thought of as APIs or contracts for our data. APIs can change over time, and schemas help version those changes.
- The Schema Registry, a separate component from Kafka, provides a way to store and retrieve schemas.
- Avro provides data serialization by leveraging schemas that can be created in a JSON format.
- As schemas change, compatibility rules help users know whether the changes are BACKWARD or FORWARD (to name a few examples) compatible.
- If schemas are not an option, different topics can be used to handle different versions of data.

12 *Stream processing with Kafka Streams and ksqlDB*

This chapter covers

- Getting started with Kafka Streams.
- Using basic Kafka Streams APIs.
- Using State Stores for persistent storage.
- Enriching transaction streams.

So far in your journey in learning about Kafka, you've focused on the parts that made a complete event-streaming platform, including the Kafka brokers, producer clients, and consumer clients.

With this foundation, we can expand our toolset and understand the next layer of the Kafka ecosystem - stream processing using Kafka Streams and ksqlDB. These technologies offer abstractions, APIs, and DSLs based on the foundation that we have been building on in the previous chapters.

This chapter introduces a simple banking application - the processing of funds movement in and out of the accounts.

In our application, we will implement a Kafka Streams topology to process transaction requests submitted to the `transaction-request` topic *atomically*.

> **NOTE** **About atomicity**
>
> Our business requirement states that we must check whether the funds are sufficient for every request received before updating the account's balance being processed. As per requirements, our application can't process two transactions simultaneously for the same account. This would create a race condition in which we cannot guarantee we can enforce the balance check before withdrawing funds. We will leverage Kafka's inter-partition ordering guaranties to implement serializable (ordered) processing of transactions for a particular account.

We will have a *data generator* program that writes simulated transaction requests to the Kafka topic with a key equal to the transaction's account number. Therefore, we can ensure all transactions will be processed by a single instance of our Transaction Service, no matter how many applications are concurrently running. Kafka Streams won't commit any message offset until it completes our business logic of managing a transaction request.

Let's introduce the *Processor API* by implementing *a transformer* component from Kafka Streams. This utility allows us to process events one by one while interacting with a `State Store` – another element of Kafka Streams that helps us persist our account balance in a local instance of an embedded database - RocksDB.

Next, we will demonstrate how to load reference account details into an `account` topic using ksqlDB's `INSERT INTO` statement - an alternative way to produce data to Kafka topics. Finally, we will write a second Stream Processor to generate a detailed transaction statement enriched with account details. Rather than creating another Kafka Streams application, we will leverage ksqlDB to declare a stream processor that will enrich our transactional data in real-time with our referential data coming from the `account` topic. This section aims to show how we can use an SQL-like query language to create stream processors (with similar functionality to Kafka Streams) without compiling and running any code.

Let's dig into the Kafka Streams API's details after refreshing the concepts of stream processing applications.

12.1 Kafka Streams

In general, stream processing (or streaming) is a process or application you implement that deals with a uninterrupted flow of data and performs work as soon as that data arrives. This application does not execute on a regular schedule or even query a database for this data. Views can be created from this data, but otherwise, we are not limited to a point-in-time view.

Enter Kafka Streams!

Kafka Streams is a library and not a standalone cluster [190]. Notice that this description includes the word *library*. However, this alone can help us create stream processing for our applications.

Figure 12.1 shows an example application setup that is *NOT* deployed on the same hardware as the Kafka cluster. No other infrastructure is required besides the need to leverage an existing Kafka cluster. The Kafka Streams library is a part of your JVM-based application.

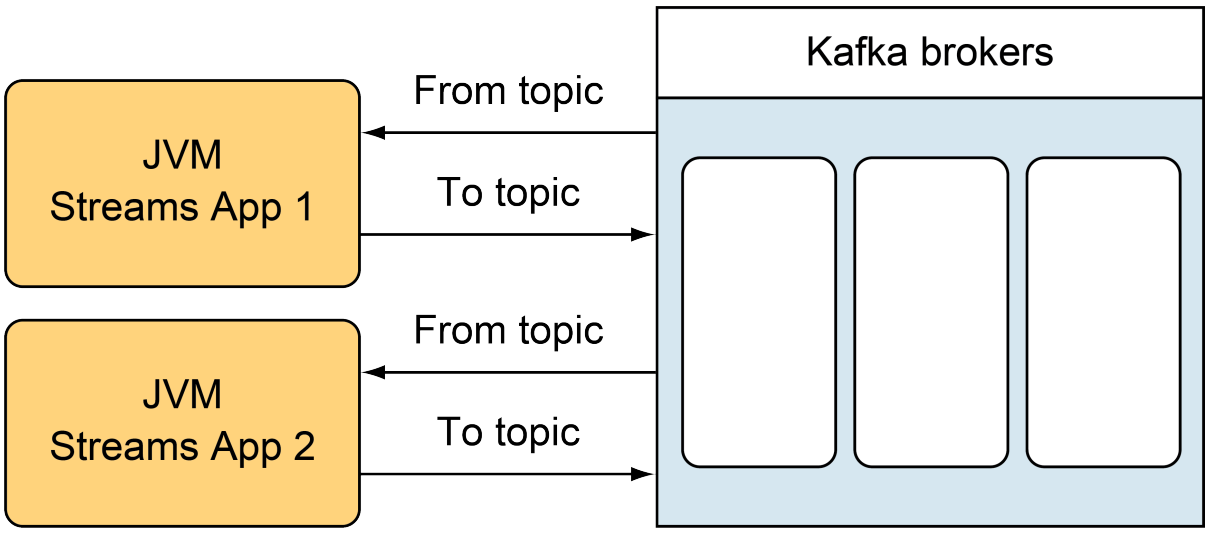

Figure 12.1 Streams as a library

Not having additional components makes this API one that can be easily tested when starting with a new application. Though other frameworks might require more cluster management components, Kafka Streams applications can be built and deployed using any tool or platform that allows JVM-based applications to run.

> **NOTE** **Our application won't run on the brokers of our cluster - we will run our application outside of the Kafka cluster. This approach would guarantee the separation of concerns of resource management of Kafka brokers and stream processors.**

Kafka Streams is different from some frameworks that use micro-batches (a small group of records together). The Streams API performs *per-record* (or *per message*) processing. You don't want to wait for a batch to form or delay that work if you're concerned about your system reacting to the events as soon as they are received.

As we consider how to implement our applications, one of the first questions that come to mind is choosing a producer/consumer client over using the Kafka Streams library. Although the Producer API is excellent for taking precise control of how your data gets to Kafka, and the Consumer API for consuming events, sometimes you might not want to implement every aspect

of the stream processing framework yourself. Instead of using lower-level APIs for stream processing, we want to use an abstraction layer that will allow us to work with our topics more efficiently.

Kafka Streams might be a perfect option if our requirements include data transformations, with potentially complex logic, consuming and producing data back into Kafka. Streams offer a choice between a functional DSL (domain-specific language) and the more imperative Processor API. Let's take a look at starting to use the Kafka Streams DSL.

> **NOTE** **Domain-specific language**
>
> Domain-specific languages (DSLs) are meant to provide a language that makes it easier to work with a specific domain. The language can use constructs and even method names that match the domain you are exposing. SQL, which is used with databases commonly, and HTML, used for creating webpages, are examples of languages to consider DSLs. In comparison, programming languages like Java or Python are general in that you can use generic building blocks to provide a wide range of outcomes. Though the official Kafka Streams documentation refers to the high-level Kafka Streams API as a DSL, we like to refer to it as fluent API or Fluent Interface.

12.1.1 KStreams API DSL

The first API that we're going to look at is the KStreams API. Kafka Streams is a data processing system designed around the concept of a graph - one that doesn't have any cycles in it [191]. It has starting node and an ending node. Data flows from the start node to the end node. This graph has nodes, or processors, along the way to process and transform data. Let's take a look at a scenario where we can model a data processing process as a graph.

We have an application that gets transactions from a payment system. At the beginning of our graph, we have to have a *source* for this data. Since we're using Kafka as a source of data, a Kafka topic will be our starting point. This origin point is often referred to as a *source processor* (or *source node*). This is the start of the processing; there aren't any previous processors.

Our first example is an existing service that captures transactions from an external payment system and places transaction request events into a topic.

> **NOTE** **We will simulate this behavior with a simple data gen application.**

This transaction request event is needed to update the balance for a particular account. The results of the transaction processor will go into two Kafka topics - successful transactions will

land in "transactions-success" and unsuccessful transactions in "transaction-failure." Because this is the end of the road for our small application, we will create a pair of *sink processors* (or *sink nodes*) to write to our *success* or *failure* topics.

> **NOTE** Some processor nodes may not have a connection to sink nodes. In this case, those nodes create side effects elsewhere and do not require sending data back to Kafka, e.g., printing information to the console or writing data to the state stores.

Figure 12.2 shows a DAG representation of how data flows.

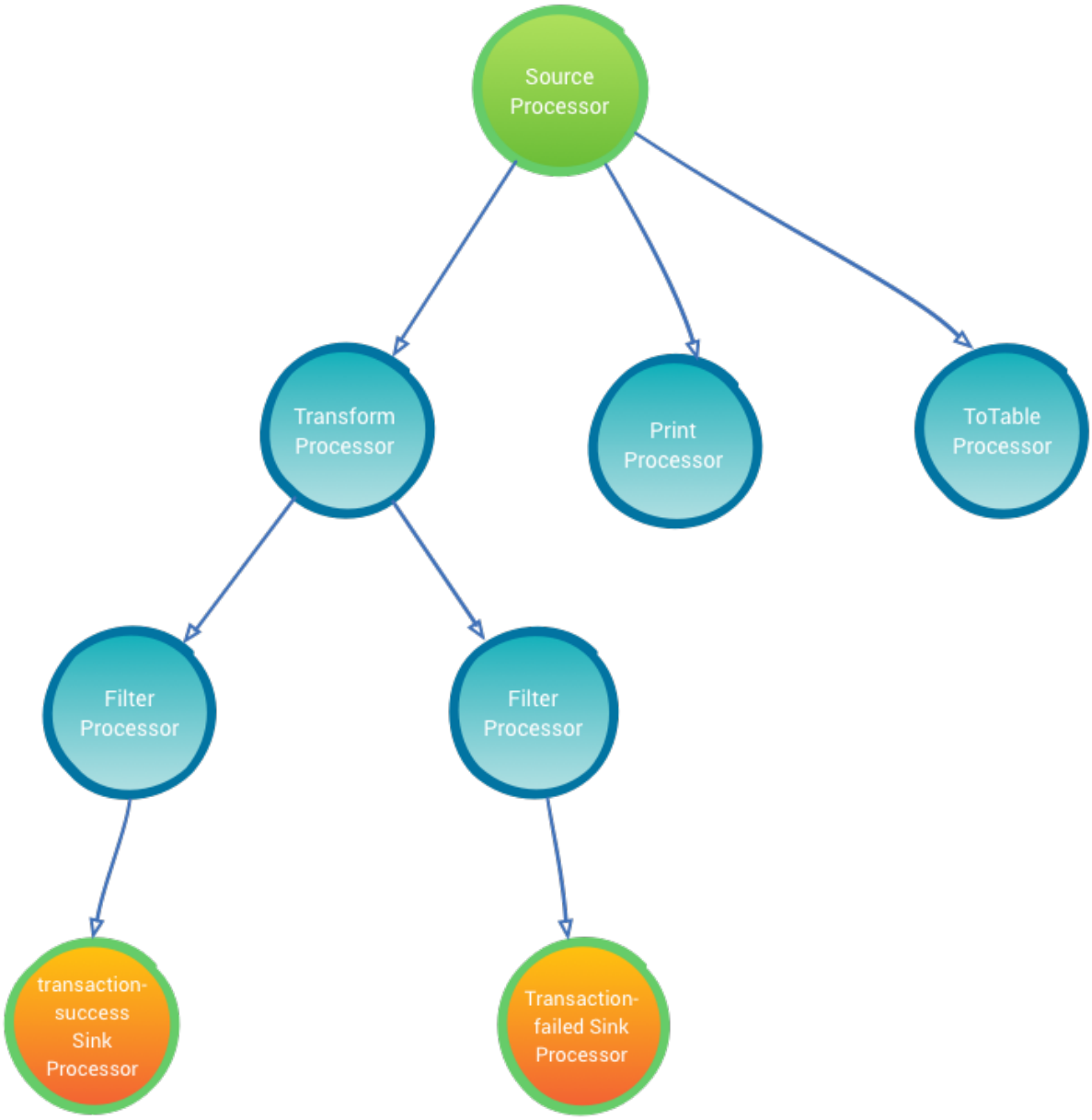

Figure 12.2 Directed Acyclic Graph of our stream processing application

Figure 12.3 shows you how this DAG maps out to Kafka Streams topology.

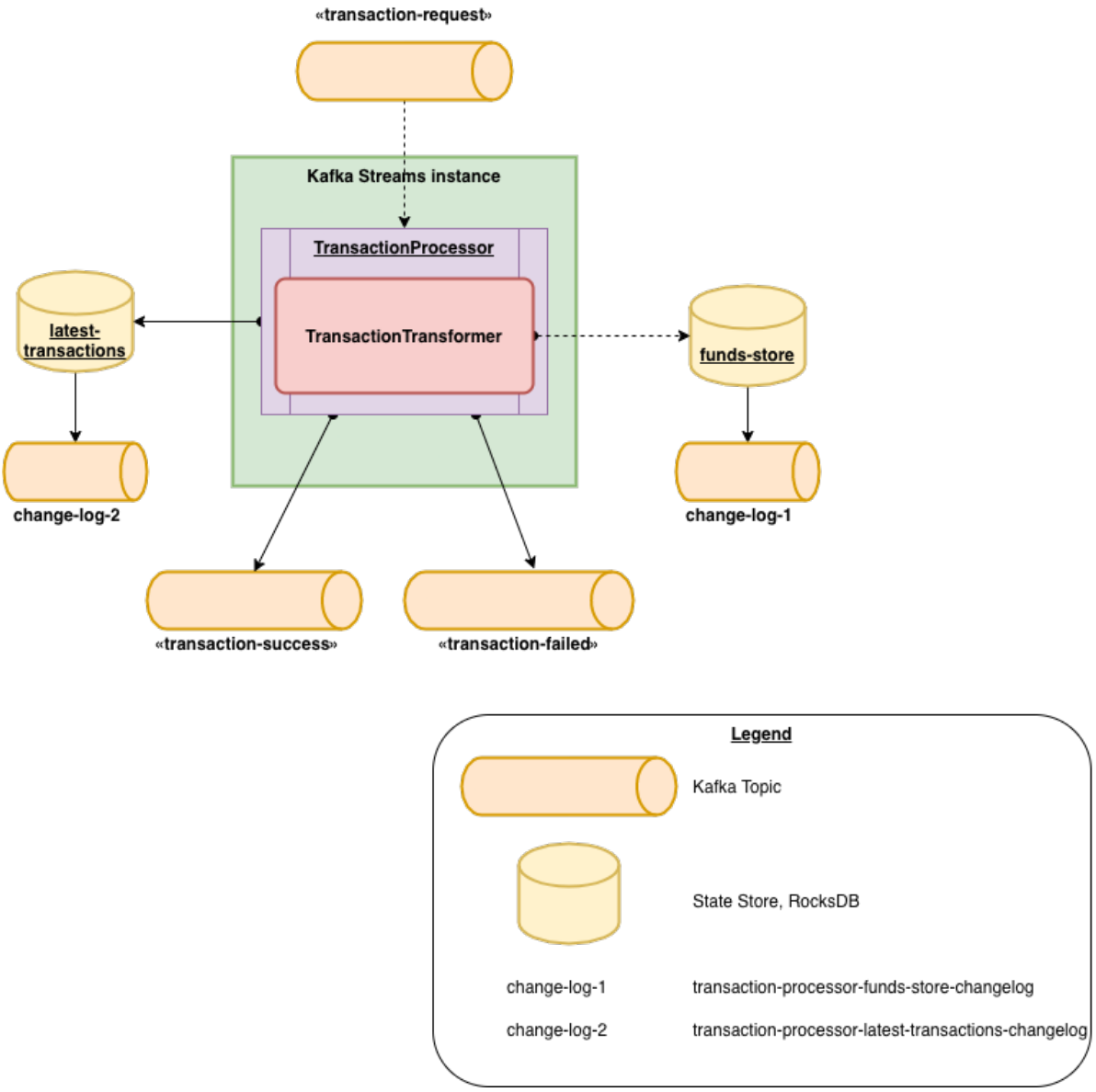

Figure 12.3 Topology for your Transaction Processing Application

Now that we have a map for a guide, let's look at what this application looks like with the DSL code. Unlike our earlier examples, when using this API, we don't need to reach a consumer directly to read our messages, but can leverage a builder to start creating our stream.

Listing 12.1 shows the creation of a source processor.

NOTE	At this point you're defining our topology but not invoking it (e.g. processing isn't started yet).

We use the `StreamsBuilder` object in order to create a stream from the Kafka topic

`transaction-request`. `transaction-request` will be the source of our data and the logical starting point of our processing.

Listing 12.1 Source topic definition

```
StreamsBuilder builder = new StreamsBuilder()     ❶

KStream<String, Transaction> transactionStream =
  builder.stream("transaction-request",
                 Consumed.with(stringSerde, transactionRequestAvroSerde));      ❷
```

❶ `StreamsBuilder` is the starting point of building our topology.

❷ We're defining a `KStream` object from the topic named *"transaction-request"* since we're going to start our processing from this topic.

The next step is to add to our topology using the KStream that we created from the source processor.

Listing 12.2 Processor and sink topic definition

```
final KStream<String, TransactionResult> resultStream =
        transactionStream.transformValues(() -> new TransactionTransformer());   ❶

    resultStream
        .filter(TransactionProcessor::success)
        .to(this.transactionSuccessTopicName,
          Produced.with(Serdes.String(), transactionResultAvroSerde));

    resultStream
        .filterNot(TransactionProcessor::success)
        .to(this.transactionFailedTopicName,
          Produced.with(Serdes.String(), transactionResultAvroSerde));   ❷

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), props);    ❸
kafkaStreams.start();    ❹
Thread.sleep(000);
kafkaStreams.close();    ❺
```

❶ We use the stream created from the previous source processor to continue to build our topology.

❷ Depending on the transaction success criteria, our sink processor writes to one of two topics "transaction-success" or "transaction-failed".

❸ We pass our built topology and configuration to create a `KafkaStreams` object.

❹ Finally, we start our stream application! This is meant to continue in the same way we have consumer clients poll in an infinite loop.

❺ We will need to close the stream to stop processing.

Although we had only one processing node that wasn't involving reading and writing data, it is easy to see how we could chain multiple nodes on our path.

Looking over the code in Listing 12.2, you might notice the lack of direct usage of the following:

- A consumer client to read from the source topic.
- A producer client to send our messages at the end of the flow.

This layer of abstraction allows us to focus on our logic rather than the details. Let's look at another practical example. Imagine we simply want to log transaction requests in the console without processing them.

Listing 12.3 Price Tracker KStream

```
KStream<String, Transaction> transactionStream =
  builder.stream("transaction-request",
                 Consumed.with(stringSerde, transactionRequestAvroSerde));    ❶

    transactionStream.print(Printed.<String, Transaction>toSysOut().
      withLabel("transactions logger"));     ❷

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), props);
kafkaStreams.cleanUp();     ❸
kafkaStreams.start();
Thread.sleep(1000);
kafkaStreams.close();
```

❶ We're going to source our data from the topic: `transaction-request`. Also, we are using a custom object `Transaction` to hold our data

❷ We're going to print out information as we get prices, so we can more easily follow along with the example.

❸ Doing a cleanup of the local data store ensures that we're running without past state.

Listing 12.3 shows the reading of price events from a topic `transaction-request`. This flow is so simple that we just write out the prices to the console, although it could have been an API call to send an SMS or email. Notice the added call to `cleanup` before starting the application. This method provides a way to remove the local state stores for our application. Just remember to only do this before the start or after closing the application.

Despite the ease of use of `KStreams`, they are not the only way we can process our data. The KTable API provides us with an alternative to always adding events to our view by instead representing data as updates.

12.1.2 KTable API

Although a KStream can be thought of as event data always being appended to our log, a KTable will allow us to think about a log-compacted topic [192]. In fact, we can also draw a parallel to a database table that deals with updates in place. Recall from working with compacted topics in Chapter 7, that our data needs to have a key for this to work. Without a key, the value to be updated won't really make practical sense.

Listing 12.4 Price KTable

```
StreamsBuilder builder = new StreamsBuilder();

KTable<String, Transaction> transactionStream =
  builder.stream("transaction-request",   ❶
                Consumed.with(stringSerde, transactionRequestAvroSerde),
                Materialized.as("latest-transactions"));   ❷

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), props);
```

❶ A method `StreamsBuilder.table()` is used to create a `KTable` from *transaction-request* topic.

❷ `KTable` records will also be materialized locally in *latest-transactions* state store.

Running the code from Listing 12.4, we see that not every order event shows up. Instead, we see only the four distinct orders.

What is familiar is the way we build the stream. We use a builder to create the steps and then once it is defined, we call `start`. Until that moment, nothing is processing in our application.

12.1.3 GlobalKTable API

Although similar to `KTable`, the `GlobalKTable` is populated with data from all partitions of a topic [193]. The foundational knowledge about topics and partitions pays off in understanding these abstractions, as shown in how the `KafkaStreams` instances consume each partition of a topic.

Listing 12.5 is an example of using a join with a `GlobalKTable`. Imagine a stream that gets updated with details about a mailed package for a customer. These events contain the customer id, and we can then join on a customer table to find their associated email and send a message.

Listing 12.5 Mailing Notification GlobalKTable

```
...
StreamsBuilder builder = new StreamsBuilder();

final KStream<String, MailingNotif> notifiers = builder.stream("mailingNotif");       ❶
final GlobalKTable<String, Customer> customers = builder.globalTable("customers");    ❷

lists.join(customers, (mailingNotifID, mailing) -> mailing.getCustomerId(),
  (mailing, customer) -> new Email(mailing, customer))
  .peek((key, email) -> emailService.sendMessage(email));   ❸

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), props);
kafkaStreams.cleanUp();
kafkaStreams.start();
Thread.sleep(1000);
kafkaStreams.close();
```

❶ The notification stream is listening for new messages about mailings to send to a customer.

❷ The `GlobalKTable` holds a list of `Customer` information, including email information.

❸ The join matches the customer that needs to be notified to send an email.

As shown in Listing 12.5, a new `GlobalKTable` can be built using the method `globalTable`. Whereas a table that is not global might not consume all of the input topics data due to multiple partitions, the global table consumes all partitions for your running code. Usage of `GlobalKTable` is ideal for use cases where we deal with data that can fit on a processing node easily, as each partition's data is replicated to the local node.

IMPORTANT **The idea of a global table is to make the data available to our application regardless of which partition it is mapped to.**

Even though the Streams DSL has been excellent for quick use cases, sometimes we might need more control as we send data along our logic paths. Developers can use the Processor API alone or with the Streams DSL to provide even more options.

12.1.4 Processor API

It's important to note that when reviewing code for another streaming application or even looking at getting into lower abstraction levels in our own logic, we might run into examples from the Processor API. This is considered not as easy to use as the DSL discussed in the previous sections—but at the benefit of giving us more options and power over our logic [194]. Let's look at an example in Listing 12.6, where we create a topology and highlight the differences from our previous Streams applications.

Listing 12.6 Processor API Source

```
import static org.apache.kafka.streams.Topology.AutoOffsetReset.LATEST;

public static void main(String[] args) throws Exception {
//...
final Serde<String> stringSerde = Serdes.String();
Deserializer<String> stringDeserializer = stringSerde.deserializer();
Serializer<String> stringSerializer = stringSerde.serializer();

Topology topology = new Topology();     ❶

topology = topology.addSource(LATEST,     ❷
  "input",      ❸
  stringDeserializer,     ❹
  stringDeserializer,     ❺
  "input-topic");     ❻
}
```

❶ Start with the Topology object to create our flow.

❷ We're setting the offset to LATEST as we usually do.

❸ This is the name of the node that we can refer to in later steps.

❹ This is a deserializer for our key.

❺ This is the deserializer for our value.

❻ This is the Kafka topic that we will read from to start.

First, we start building our graph using the `Topology` object. Setting the offset to `LATEST` and listing our key and value deserializers should be familiar from when we set configuration properties for our client consumers in chapter 5. In the example, we're naming the node as `input` and reading from the topic `input-topic`. Our next step is to add a processing node.

Listing 12.7 Processor API Processor Node

```
topology = topology.addProcessor("testProcessor",    ❶
  () -> new TestProcessor(),    ❷
  "input");    ❸
```

❶ This is the name we're giving to our new processor node.

❷ This is used to create a processor instance from a ProcessorSupplier.

❸ This can be one or a list of nodes that will send data to this node.

Listing 12.7 shows that when we define a processing node, we give it a name, `testProcessor`, and associate the logic with the step, we also list which nodes will provide the data.

To finish out our simple example, let's look at Listing 12.8. The sink is where we place our data at the end of processing. The topic name and the key and value serializers should be familiar from our earlier work with producer clients. As we did with the other parts of the topology, we define `testProcessor` as one of the nodes from which we will get data in our flow. To end our example, Listing 12.8 shows how we define two separate sinks to complete our topology.

Listing 12.8 Processor API Processor Sink

```
topology = topology.addSink("Output-Sink1",    ❶
  "sink-topic1",    ❷
  stringSerializer,    ❸
  stringSerializer,    ❹
  "testProcessor");    ❺

topology = topology.addSink("Output-Sink2",    ❻
  "sink-topic2",
  stringSerializer,
  stringSerializer,
  "testProcessor");

KafkaStreams kafkaStreams = new KafkaStreams(topology, props);
kafkaStreams.cleanUp();
kafkaStreams.start();
Thread.sleep(1000);
kafkaStreams.close();
```

❶ We're adding a sink node name.

❷ This is the name of the output topic we plan to use.

❸ This is a serializer for our key.

❹ This is a serializer for our value.

❺ This is the node that will feed us data to write to the sink.

❻ We add a second sink to our topology.

In our processor code, we're going to show how we can use logic to direct the flow of our data. Our `TestProcessor` enables us to forward the flow, including the key and value, to the sink named `Output-Sink2`.

Although this is hard-coded in Listing 12.9, we can use logic to determine when to send data to the other sink.

Listing 12.9 Processor Custom Code

```
public class TestProcessor extends AbstractProcessor<String, String> {    ❶
    @Override
    public void process(String key, String value) {
      context().forward(key, value, To.child("Output-Sink2"));    ❷
    }
}
```

❶ Extending `AbstractProcessor` allows us to quickly implement the process method for our custom logic.

❷ While we hardcoded the next node, we could also direct the forward with additional logic.

Even though it's easy to see that the code is more verbose than our DSL examples, the important thing is the control we now have in our logic that was not shown in our simple flow above. If we want to control the schedule of when processing occurs or even the commit time of results, we'll need to dig into more complex Processor methods.

12.1.5 Kafka Streams Architecture

While our example application only used a single instance, streaming applications can scale by increasing the number of threads and deploying more than one instance. As with the number of instances of a consumer in the same consumer group, our application's parallelism is related to *the number of partitions* in its source topic. For example, if our starting input topic has eight partitions, we would plan to scale to eight instances of our application. Unless we wanted to have an instance ready in case of failure, we wouldn't have more instances because they wouldn't take any traffic. Figure 12.3 shows an example of having the same Streams application deployed in a single-threaded manner across two different instances. In this model, with four total partitions, we end up with each task being assigned two partitions [195].

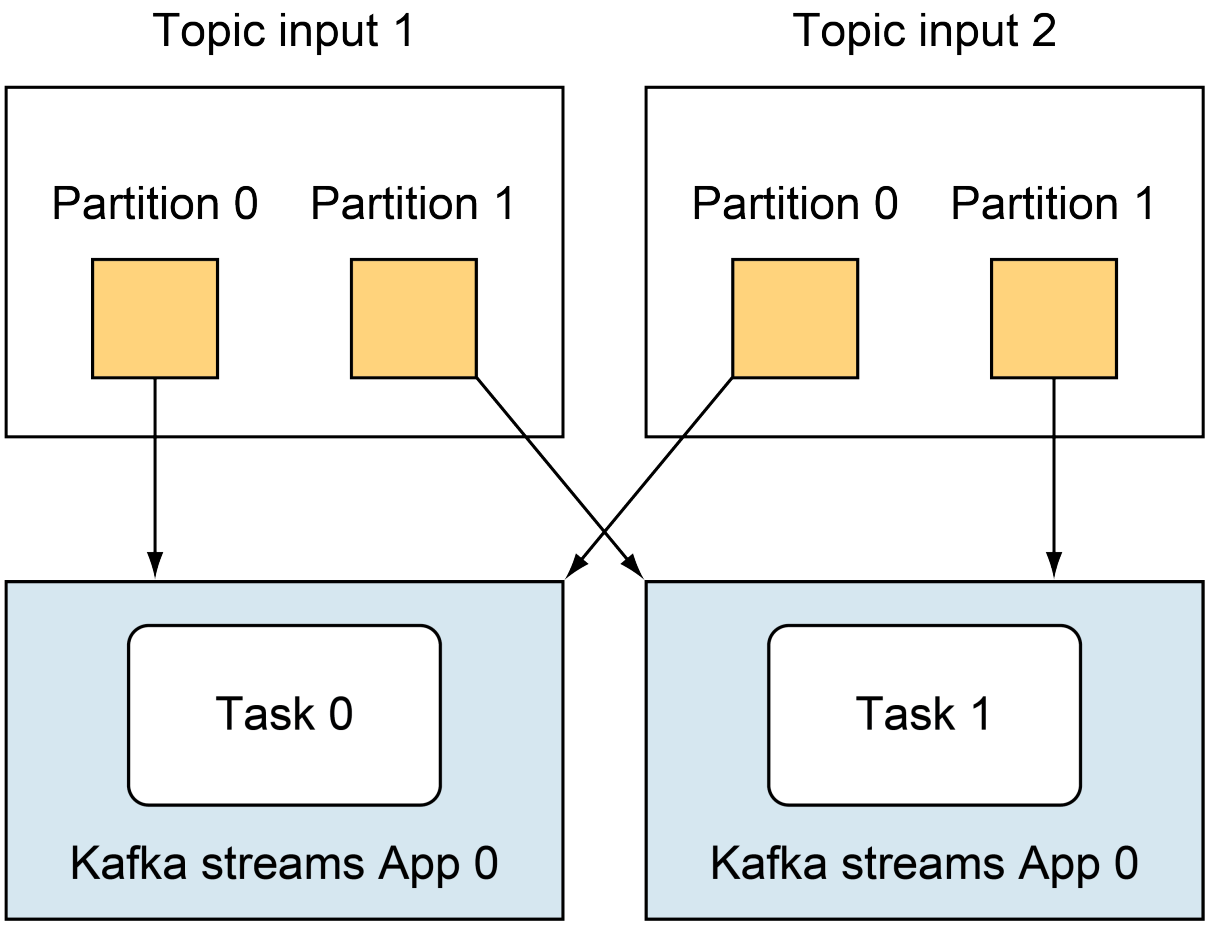

Figure 12.4 Instances Limited by Topic Partition

Whenever we think about our application's design, it is crucial to mention the processing guarantees that our use case requires. Kafka Streams supports `at-least-once` and `exactly-once` processing semantics.

> **NOTE** In version 2.6.0, "exactly-once beta" was introduced. This version enables higher throughput and scalability by attempting to reduce resource utilization [196].

If your application logic depends on `exactly-once` semantics, having your Kafka Streams application within the walls of the Kafka ecosystem helps ensure this possibility. As soon as you send data outside into external systems, you have to look at how they achieve any promised delivery options. Whereas the Streams API can treat retrieving topic data, updating the stores, and writing to another topic as one atomic operation, external systems cannot. System boundaries become significant when they impact your guarantees.

As a reminder, with `at-least-once` delivery, it is crucial to note that although data should not be lost, you might have to prepare for the situation where your messages are processed more than once. At the time of writing, `at-least-once` delivery is the default mode, so make sure you're okay with addressing duplicate data in your application logic.

Kafka Streams is designed with fault-tolerance in mind. And it does so in the ways that we have seen before in our Kafka cluster. The state stores used are backed by a replicated Kafka topic that is partitioned. Due to Kafka's ability to retain messages and replay what has happened before a failure, users can successfully continue without manually re-creating their state.

If you're interested in continuing deeper into what Kafka Streams can offer, we recommend *Kafka Streams in Action* (www.manning.com/books/kafka-streams-in-action) by William P. Bejeck Jr. because it dives further into the details.

12.2 ksqlDB: An event streaming database

ksqlDB (ksqldb.io) is an event streaming database. This product was first introduced as KSQL, but the ksqlDB project underwent a name change in November 2019. The Apache Kafka® community developed various clients to help make our data work easier. ksqlDB exposes the power of Kafka to anyone who has ever used SQL. ksqlDB opens the door for those familiar with relational databases to jump into the world of stream processing and create these applications using a toolset that they have used for years. Suddenly, no Java or Scala code is needed to leverage the topics and data inside our clusters.

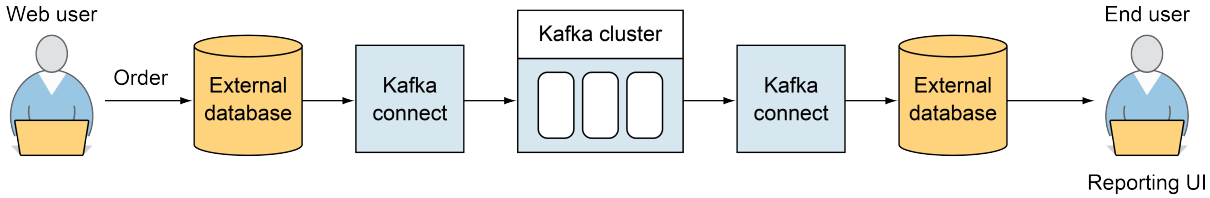

Figure 12.5 Example Kafka Application Flow

Another primary driver is that as users worked with the entire application lifecycle, it was often the case that Kafka provided a part of the flow and not the whole architecture needed.

Figure 12.4 shows an example of one way that we could leverage Kafka. Notice that to serve users, the data from Kafka is moved into an external datastore. For example, imagine an application that would trigger an order occurring in an e-commerce system. An event is triggered for each stage of the order process and acts as a status for the purchaser to know what is happening with their order. Before ksqlDB, it was often the case that the order events would be stored in Kafka (and processed with Kafka Streams or Spark) and then be moved to the external system using Kafka Connect API. The application would then read from that database the view created from the event stream to show the user as a point-in-time state. With the pull query and connector management features added to ksqlDB, developers have a path to remain within the ecosystem to show users these materialized views.

Figure 12.5 shows a high-level overview of how the Kafka ecosystem can provide a more consolidated application.

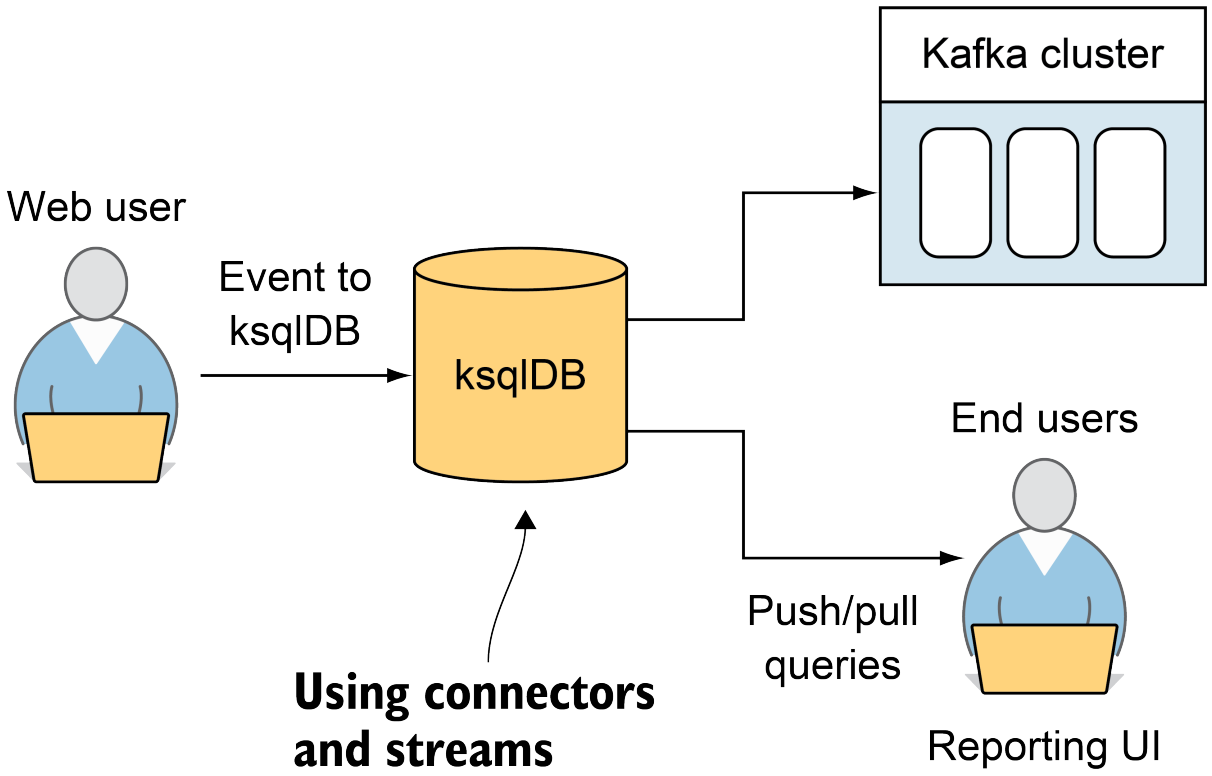

Figure 12.6 ksqlDB Example Kafka Application Flow

Let's dig into the types of queries that ksqlDB supports, starting with pull queries that we just introduced.

12.2.1 Queries

Pull queries and *push queries* can help us build applications. Pull queries fit well when used in a synchronous flow like request-and-response patterns [197]. We can ask for the current state of the view that has been materialized by events that have arrived. The query returns a response and is considered completed. Most developers are familiar with this pattern and should know that the data was a point-in-time snapshot of their events when the query was retrieved.

Push queries, on the other hand, fit in well when used in asynchronous patterns [198]. In effect, we subscribe much like we did when using a consumer client. As new events arrive, our code can respond with the necessary actions. The connection stays alive and accepts changes over time. The term `EMIT CHANGES` is used in our SQL statement to reflect a push query.

12.2.2 Local development

Though I've tried to avoid bringing in extra technologies besides Kafka proper, the easiest way to go with ksqlDB local is with Confluent's Docker images. Located at ksqldb.io/quickstart.html, you can download images that include a complete Kafka setup or just `ksqldb-server` and `ksqldb-cli`.

If you're using the Docker images, you can start those images with `docker-compose up`. Now, you should be able to use the `ksqldb-cli` to create an interactive session with your command terminal to your ksql server. As users know, after you have your database server, you need to define your data. For more information on running examples, refer to Appendix A.

Listing 12.10 Create an interactive session with ksqlDB

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088     ❶
> SET 'auto.offset.reset'='earliest';     ❷
```

❶ This will allow you to connect to the ksqlDB server and run commands from your terminal

❷ Set offset reset policy to `earlies` allows ksqlDB to process data already available in Kafka topics.

Here is an example of a situation where we can start to discover ksqlDB with an extension of our transaction processor. We'll learn how, using existing data from processed transactions, we can generate a statement report. The statement report will include extended (or enriched) information about the transaction's account. We will achieve this by joining successful transactions with account data. Let's start with creating a stream of a successful transaction from Kafka's topic.

NOTE **Because data was previously available in a Kafka topic (from our Kafka Streams application), we may need to reset the offset with the command `SET 'auto.offset.reset' = 'earliest';` so ksqDB will be able to work with the existing data. We'll need to run this command before we execute the `CREATE` statement.**

Listing 12.11 Create an interactive session with ksqlDB

```
CREATE STREAM TRANSACTION_SUCCESS (
  numkey string KEY,    ❶
  transaction STRUCT<guid STRING, account STRING,
                  amount DECIMAL(9, 2), type STRING,
                  currency STRING, country STRING>,    ❷
  funds STRUCT<account STRING,
               balance DECIMAL(9, 2)>,
  success boolean,
  errorType STRING
) WITH (
  KAFKA_TOPIC='transaction-success',    ❸
  VALUE_FORMAT='avro');    ❹
```

❶ We need to tell ksqlDB about the record key.

❷ ksqlDB supports work with nested data. In our Kafka Streams example, we used a nested type `Transaction` in the `TransactionResult` class. Using the `STRUCT` keyword, we define a structure of nested type.

❸ using the `KAFKA_TOPIC` attribute of the `WITH` clause, we can specify which topic to read data from.

❹ ksqlDB integrates with Confluent Schema Registry and natively supports schemas in Avro, Protobuf, JSON, and JSON-Schema formats. Using Schema Registry integration, ksqlDB, in many cases, can use schemas to infer or discover stream or table structure. This is a tremendous help for enabling effective collaboration between microservices, for example.

As mentioned, we need to use comprehensive information about accounts. In contrast to the history of successful transactions, we are not interested in a complete history of account information changes. We need to have a lookup of accounts by account id. For that purpose, we can use `TABLE` in ksqlDB.

Listing 12.12 Create a table

```
CREATE TABLE ACCOUNT (number INT PRIMARY KEY)    ❶
WITH (KAFKA_TOPIC = 'account', VALUE_FORMAT='avro');    ❷
```

❶ We need to choose a primary key for our table. The account number field will be used as a key.

❷ Using Avro schema, ksqDB can learn about fields of the account table.

Listing 12.13 Creating transaction statement stream enriched with account information

```
CREATE STREAM TRANSACTION_STATEMENT AS
    SELECT *
    FROM TRANSACTION_SUCCESS
    LEFT JOIN ACCOUNT
        ON TRANSACTION_SUCCESS.numkey = ACCOUNT.numkey
    EMIT CHANGES;
```

Despite the SQL statement in Listing 12.10 looking similar to SQL statements you may have run in the past, I want to draw your attention to a small but mighty difference. The use of `EMIT CHANGES` creates what we had previously discussed as a *push query*. Instead of returning to our command prompt, this stream will be running in the background! To test our query, we need a new instance of the `ksqldb-cli` to insert data into our stream to continue producing test transactions. The Kafka Streams application will process those transactions. In case of success, the Kafka Streams processor will write the result to `transaction-success` topic, where it will be picked up by ksqlDB and used in `TRANSACTION_SUCCESS` and `TRANSACTION_STATEMENT` streams.

12.2.3 ksqlDB architecture

By using the Docker images, we glossed over the architecture that is part of kslDB. But it's important to know that unlike the Streams API, ksqlDB requires additional components to run. The main component is called the *ksqlDB server* [199]. The server is responsible for running the SQL queries submitted to it and getting data to and from our Kafka cluster. In addition to the query engine, a REST API also is provided. This API is used by the `ksqldb-cli` that you used in the examples.

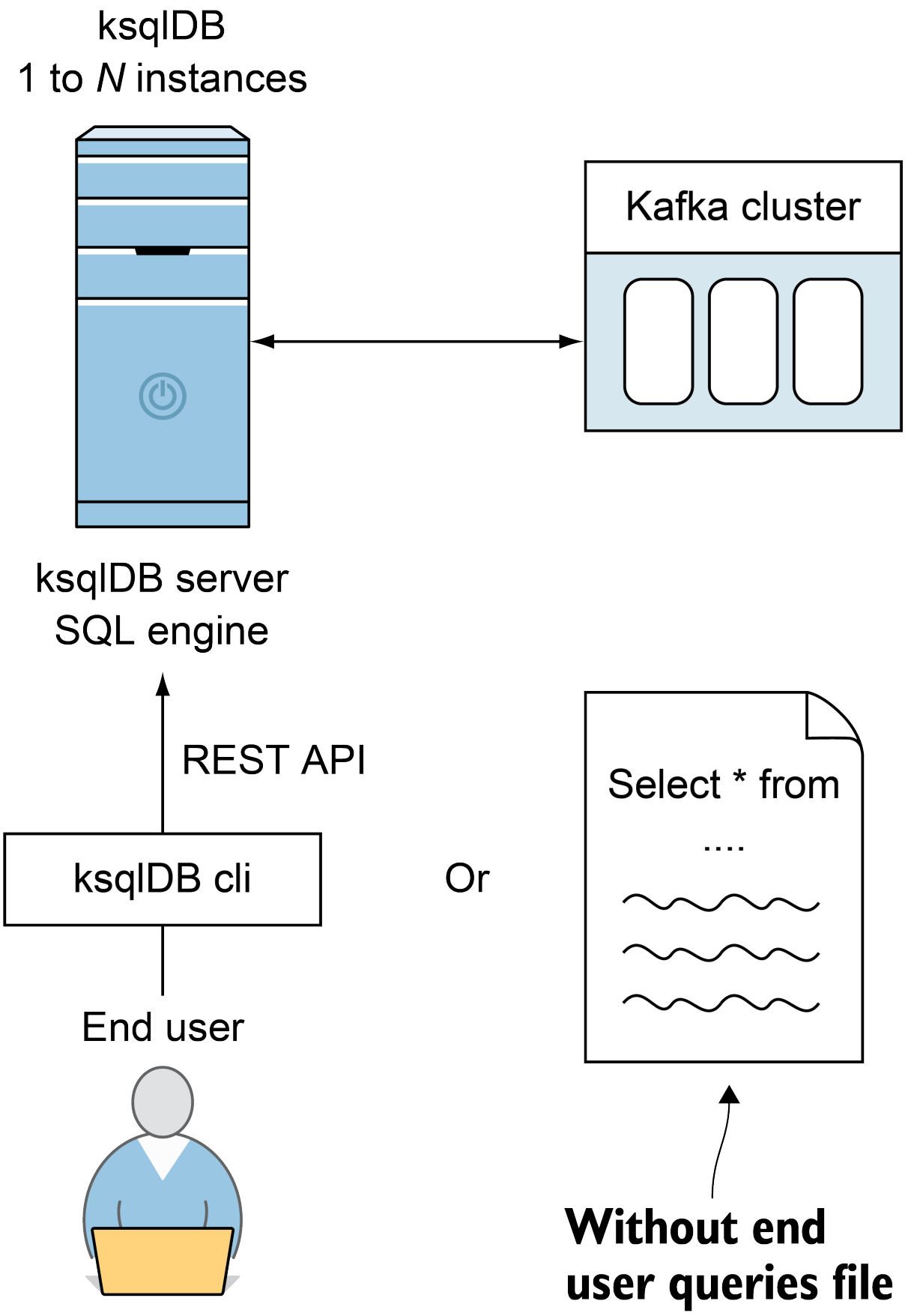

Figure 12.7 ksqlDB Architecture

The server component can (and likely should) be deployed on different machines than our existing Kafka cluster as we move toward production. Another item that we should consider is locking out users from running the ksqlDB CLI in production. Called *headless*, this will prohibit developers from running interactive queries through the command-line interface.

To configure this mode, we can either start the ksqlDB server with the `--queries-file` command-line argument or update the `ksql-server. properties` file [200]. Of course, this means that a query file is also required!

Listing 12.14 Start ksqlDB in headless mode

```
/bin/ksql-server-start.sh etc/ksql/ksql-server.properties --queries-file ksql.sql    ❶
```

❶ This will start ksqlDB in a non-interactive in which the CLI will not work

Now that we have used Kafka Streams and ksqlDB, how do we know which one to reach for as we approach new tasks? Though not a read–eval–print loop (REPL) directly, running some quick prototype tests and trials quickly with the `ksqldb-cli` might be a great way to start new applications. Another key for ksqlDB is that users who are not running Java or Scala (JVM languages) might find Kafka Streams' feature set available with this SQL option. Users looking to build micro-services would likely find the Streams API a better fit.

12.3 Going further

Even though we just introduced Kafka Streams and ksqlDB, there are still many more resources to help you continue your Kafka journey.

12.3.1 Kafka Improvement Proposals

While it might not seem the most exciting option, following Kafka Improvement Proposals (KIPs) is really one of the best ways to keep current with Kafka. Even though not everything that gets proposed will be implemented, it is interesting to see what other users of Kafka think is worth exploring as use cases change over time. As we saw in chapter 8, KIP 32 (cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica) was motivated by the need for users to fetch data when the partition leader was in a non-local
data center.

If Kafka existed only in on-premise data centers without separate data centers for disaster recovery, the proposal might not have gained acceptance. Reviewing these new KIPs allows everyone to understand the issues or features fellow Kafka users experience in their day-to-day life. KIPs are important enough to be addressed and discussed in keynotes like at Kafka Summit 2019, where KIP 500 (cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum) was discussed, which deals with the replacement of ZooKeeper.

12.3.2 Kafka projects you can explore

In addition to Kafka source code, searching GitHub or Gitlab public repositories for real-world uses of Kafka can help you learn from those projects. Although not all code is equal in quality, hopefully, the previous chapters have given you enough information to understand how the required pieces fall into place.

This book pointed out a couple of projects that use Kafka in some part to help power software and have made their source code publicly viewable. Examples included are Apache Flume (github.com/apache/flume) and Brave Tracing (github.com/openzipkin/brave).

12.3.3 Community Slack Channel

If you like a more interactive way to gather information, and a great place to search for or ask questions, visit the Confluent Community page (www.confluent.io/community/). You can find a Slack group with channels focusing on Kafka's specific parts, such as clients, connect, streams, and many other Kafka topics. The amount of detailed questions that others have posted (and that you can post) shows the depth of experiences users are willing to explore and share. There is also a community forum where you can introduce yourself and meet other vibrant community members.

Throughout this chapter, we have expanded our knowledge to learn about the further abstractions of KStreams and ksqlDB and how they relate to our core knowledge of Kafka. As the Kafka ecosystem evolves and changes or even adds new products, I have confidence that Kafka's foundations will help understand what is going on under the hood. Good luck on your continuing Kafka journey!

12.4 Summary

- Kafka Streams provides stream processing in applications with per-record (or per message) processing. It is an abstraction layer over the producer and consumer clients.
- Kafka Streams offers a choice between a functional DSL (domain-specific language) and the more imperative Processor API. Streams can be modeled as a topology using the Kafka Streams DSLs.
- ksqlDB is an event streaming database that exposes the power of Kafka to those who already know SQL. ksqlDB queries run continuously and can help us quickly prototype streaming applications.
- Kafka Improvement Proposals are a great way to see what changes are being requested and implemented in future Kafka versions.

Appendix A: Installation

Despite being a sophisticated feature set, the Apache Kafka installation process is straightforward. There are just a couple of things that you need to get started, but you may have questions about which version or operating system you should use before that.

A.1 Operating system requirements

Linux is the most likely home for Kafka and seems to be where many user support forums continue to focus their questions and answers in that context. We used macOS with Bash (a default terminal before macOS Catalina) or Zsh (default terminal since macOS Catalina). Though it's totally fine to run Kafka on Windows [201] for development, it's not recommended production environment.

NOTE	Later in here, we also explain installation using Docker [202]

A.2 Kafka versions

Apache Kafka® is one of the most active Apache Software Foundation projects. And over time, the versions of Kafka will be updated.

Kafka releases have, in general, taken backward compatibility very seriously. If you want to use a new version, please do so and update any parts of the code marked as deprecated.

TIP	Running on more than one machine for production Generally, Zookeeper and Kafka should not be running on one physical server in a production environment if you want fault tolerance. We want to make sure you can focus on learning Kafka features instead of managing multiple servers while you're learning.

A.3 Installing Kafka on your local machine

From our personal experience, you can install Kafka on a workstation with the following minimum requirements:

- A minimum number of CPUs (physical or logical): 2.
- A minimum amount of RAM: 4Gb.
- A minimum free space of hard drive: 10GB.

Use the following instructions to install Java and Apache Kafka (which includes Zookeeper) on your workstation.

A.3.1 Prerequisite: Java

Java is a prerequisite that you should install first. At the time of writing, Kafka requires the Zookeeper, which is bundled with the Kafka download.

For the examples in this book, you will use the Java Development Kit (JDK) 11. You can download Java versions from jdk.dev/download/. We recommend using SDKMAN CLI at sdkman.io to install and manage Java versions on your machine.

A.3.2 Prerequisite: Zookeeper

Even with the reduced dependency on ZooKeeper from the client side in recent versions, Kafka needs a running installation of ZooKeeper to work at the time of this writing. The Apache Kafka distribution includes a compatible version of Zookeeper. You don't need to download and install it separately. The required scripts to start and stop Zookeeper are also included in the Kafka distribution.

A.3.3 Prerequisite: Kafka download

At the time of this book's release, Kafka `2.7.1` is the latest version used in our examples. The Apache project has mirrors, and you can search for the version to download in that way. To be automatically redirected to the nearest mirror, use this URL: www.apache.org/dyn/closer.cgi?path=/kafka/2.7.1/kafka_2.13-2.7.1.tgz.

After downloading the file, take a look at the actual binary filename. It might seem a little confusing at first. For example, `kafka_2.13-2.7.1` means the Kafka version is `2.7.1` (the information after the hyphen) that uses Scala `2.13`. Compatibility issues between Scala versions led to versions being part of the package name, but this should not concern your new installation if you are using Java for your examples.

To get the most out of the examples while still making things easy to get started, you will set up a three-node cluster on a single machine. This is not a recommended strategy for production, but

it will allow you to understand critical concepts without the overhead of spending a lot of time on the setup.

> **NOTE** **Why even bother to use a three-node cluster?**
>
> Kafka's various parts as a distributed system lend themselves to more than one node. Our examples simulate a cluster without the need for different machines in the hope of clarifying what you are learning.

Next, you'll unpack the Kafka binary and locate the bin directory. Then you'll configure the cluster.

A.3.4 Unpacking Kafka

After you install Kafka, you need to configure a three-node cluster. First, you need to unpack the binary and locate the `bin` directory.

Figure 12.1 shows the tar command used to unpack the jar file, but you might need to use unzip or other tools, depending on your downloaded compression format. It's a good idea to include the Kafka scripts `bin` directory in your `$PATH` environment variable. In this case, the commands will be available without specifying a full path to the commands.

Listing A.1 Unpacking the Kafka binary

```
$ tar -xzf kafka_2.13-2.7.1.tgz
$ mv kafka_2.13-2.7.1 ~/
$ cd ~/kafka_2.13-2.7.1
$ export KAFKA_HOME=~/kafka_2.13-2.7.1
$ export PATH=$PATH:$KAFKA_HOME/bin     ❶
```

❶ Add the `bin` directory to your shell `$PATH`.

> **NOTE** **Windows script location**
>
> For Windows users, you'll find `bat` scripts under the `bin/windows` folder with the same names as the shell scripts used in the following examples. You can use Windows Subsystem for Linux 2 (WSL 2) [203] and use the same commands as you would use on Linux.

A.3.5 Starting a ZooKeeper server

The examples in this book use a single local ZooKeeper server. Chapter 5 covers what we might consider for production settings for ZooKeeper as well as Kafka.

The command in Figure 12.2 starts a single ZooKeeper server. You want to start Zookeeper before you begin any Kafka brokers.

Listing A.2 Starting ZooKeeper

```
$ cd $KAFKA_HOME
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

The next step is to create and configure a three-node cluster.

A.4 Creating and configuring a cluster

To create your cluster, you set up three servers (brokers) that form your Kafka cluster: `server0`, `server1`, `server2`. We will modify property files for each server. Kafka comes with a set of predefined defaults. Run the commands in Figure 12.3 to create configuration files for each server in our cluster. We will use the default `server.properties` as a starting point.

Listing A.3 Creating Multiple Kafka Broker servers

```
$ cd $KAFKA_HOME
$ cp config/server.properties config/server0.properties      ❶
$ cp config/server.properties config/server1.properties
$ cp config/server.properties config/server2.properties
```

❶ Make three copies of the default server properties files.

Run the following command to open each config file, respectively, to make the following changes to properties.

> **NOTE** In our examples, we use `vi` as our text editor, but you can edit those files with a text editor of your choice.

Listing A.4 Configure server 0

```
$ vi config/server0.properties      ❶

broker.id=0
listeners=PLAINTEXT://localhost:9092
log.dir=/tmp/kafka-logs-0

$ vi config/server1.properties      ❷

broker.id=1
listeners=PLAINTEXT://localhost:9093
log.dir=/tmp/kafka-logs-1

$ vi config/server2.properties      ❸
broker.id=2
listeners=PLAINTEXT://localhost:9094
log.dir=/tmp/kafka-logs-2
```

❶ Update `id`, `port`, and `log` directory for broker id 0.

❷ Update `id`, `port`, and `log` directory for broker id 1.

❸ Update `id`, `port`, and `log` directory for broker id 2.

> **TIP** **Unique broker ids**
>
> **Each Kafka broker will run on its port and use a separate log directory. It is also critical each configuration file has a unique `id` for each broker. Broker will use this id to register itself in You will usually see your broker `ids` start at `0` following a zero-based array indexing scheme.**

After this, you can start each broker. You start each broker server using the built-in scripts that are part of the initial install, along with the configuration files that you updated in Figure 12.3. If you wish to observe Kafka broker output in the terminal, we recommend starting each process in a separate terminal tab or window, and leave them running.

Listing A.5 Starting Kafka - Run the following in their console

```
$ bin/kafka-server-start.sh config/server0.properties      ❶
$ bin/kafka-server-start.sh config/server1.properties      ❷
$ bin/kafka-server-start.sh config/server2.properties      ❸
```

❶ Starts the broker process for server0.

❷ Starts the broker process for server1.

❸ Starts the broker process for server2.

> **TIP** **Java process information**
>
> **If you close a terminal or have a hanging process, do not forget about running the `jps` command. That command will help you find the java processes you might need to kill.**

Figure 12.4 shows an example from my machine where you can see the brokers' PIDs and ZooKeeper (`QuorumPeerMain`) in the output from three brokers and one running ZooKeeper instance. Your process ids for each instance are on the left and will likely be different each time you run the start scripts.

Listing A.6 `jps` output for Zookeeper (`QuorumPeerMain`) and 3 Brokers

```
2532 Kafka     ❶
2745 Kafka     ❶
2318 Kafka     ❶
2085 QuorumPeerMain   ❷
```

❶ Kafka JVM process label and id for each broker.

❷ ZooKeeper JVM process label and `id`.

Now that you know how to configure a local installation manually let's look at using the Confluent Platform.

A.5 Confluent Platform

Confluent Inc. (www.confluent.io/) makes Confluent Platform—an event-streaming platform based on Apache Kafka. Confluent Platform is an enterprise-ready packaging option that complements Apache Kafka with essential development capabilities. It includes packages for Docker, Kubernetes, Ansible, and various others. Confluent actively develops and supports Kafka clients for C++, C#/.NET, Python, and Go. It also includes Schema Registry, which you'll learn about in Chapter 11.

Confluent also provides a fully managed service cloud-native Kafka, which might come in handy for later projects. Managed service provides Apache Kafka® experience without requiring knowledge on how to operate it. This is a characteristic that keeps developers focused on what matters, which is coding.

The Confluent version 6.1.1 download includes Apache Kafka version 2.7.1, which is used throughout this work.

In chapters 3 and 11, we are about Schema Registry, which is included in Confluent Platform Community Edition. In Chapter 12, you will learn about stream processing with ksqlDB, which also included in Confluent Platform Community Edition.

You can follow easy installation steps from official Confluent documentation docs.confluent.io/current/installation/installing_cp/zip-tar.html#prod-kafka-cli-install

A.5.1 Confluent CLI

Confluent has command-line tools to quickly start and manage its Confluent Platform from the command line. A README.md on github.com/confluentinc/confluent-cli has more details on the convenience script usage and can be installed with instructions guide. The CLI is helpful in that it starts up multiple parts of your product as needed.

A.5.2 Docker

The Apache Kafka® community doesn't provide official docker images, but Confluent does. Those images are tested, supported, and used by many customers and users in production. In the repository of examples for this book, you can find a `docker-compose.yaml` file with preconfigured Kafka, Zookeeper, and other components. To get all components up and running, you run the command `docker-compose up -d' in the directory with the YAML file.

> **NOTE** **If you are not familiar with Docker or don't have it installed, please, follow official documentation www.docker.com/get-started.**

Listing A.7 filename.sh

```
$ git clone https://github.com/Kafka-In-Action-Book/Kafka-In-Action-Source-Code.git     ❶
$ cd ./Kafka-In-Action-Source-Code
$ docker-compose up -d      ❷

Creating network "kafka-in-action-code_default" with the default driver     ❸
Creating Zookeeper... done
Creating broker2   ... done
Creating broker1   ... done
Creating broker3   ... done
Creating schema-registry ... done
Creating ksqldb-server   ... done
Creating ksqldb-cli      ... done

$ docker ps --format "{{.Names}}: {{.State}}"     ❹

ksqldb-cli: running
ksqldb-server: running
schema-registry: running
broker1: running
broker2: running
broker3: running
zookeeper: running
```

❶ Clone a repository with book examples from Github.

❷ Start docker-compose in the examples directory.

❸ Observe the following output.

❹ Validate that all components are up and running.

A.6 How to work with book examples

You can use any IDE to open and run companion code from the book. Here are few suggestions that you can use:

- IntelliJ IDEA Community Edition www.jetbrains.com/idea/download/
- Apache Netbeans netbeans.org
- VS Code for Java code.visualstudio.com/docs/languages/java
- Eclipse STS spring.io/tools

A.6.1 Building from the command line

If you wish to build from the command line, you will have a few more steps involved. The Java 11 examples in this book built with Maven 3.6.3. You should be able to create the jar for each chapter by running from the root of the chapter directory in the folder that contains the `pom.xml` file and issuing the following command: `./mvnw verify` or `./mvnw --projects KafkaInAction_Chapter2 verify` from the root project level. We use the Maven Wrapper tool, so if you don't have Maven installed, this command will download and run Maven for you.

To run the specific class, you will need to supply a Java class that contains a `main` method as an argument after the path to your jar.

NOTE	You must use a jar that has been built with all dependencies to run the command successfully.

Listing A.8 Running Producer of Chapter 2 command line

```
java -cp target/chapter2-jar-with-dependencies.jar org.kafkainaction.producer.HelloWorldProducer
```

Listing A.15 shows running the producer Java client example from Chapter 2 for the command line.

A.7 Troubleshooting

- Make sure you have a cluster started **before** running the code and command-line examples in this book.
- If you did not get a chance to shut down your cluster correctly, you might have an old process holding onto a port you want to use the next time you attempt to start up. Use tools like `jps` or `losf` to help identify which processes are running and which might need to be killed.
- You should be located inside your installation directory when you run commands unless otherwise noted. If you are more comfortable with the command line, you can, of course, complete your setups, such as adding environment variables and aliases.
- If you are having trouble with commands not being found - check the setup of your installation directory. Do you have the files as being marked as executable? Does a command like `chmod -R 755` help? Or is the installation bin part of your `PATH` variable? If nothing else works, using the absolute path to the command should hopefully work.
- Check the source code for each chapter for a `Commands.md` file. This is a markdown file that should include most commands used throughout that specific chapter. Please look for README.md files for more notes as well.

The latest copy to reference will be at: github.com/Kafka-In-Action-Book/Kafka-In-Action-Source-Code.

Appendix B: Client example

This appendix covers:

- Example of Kafka clients in Python
- Notes on testing

While the code samples in this book focus on the Java Kafka clients, one of the easiest ways to quickly draw parallels for new users might be to look at examples in programming languages that they are more familiar with. The Apache Kafka Clients site (cwiki.apache.org/confluence/display/KAFKA/Clients) has a list that you can check out that includes options for clients in various programming languages. The Confluent Platform also has a list of included clients that it supports [204].

B.1 Python Kafka client

For this example, we are going to look at the Confluent Python Client [205]. The benefit of using a Confluent client is that you can have a high-level of confidence that the clients will be compatible not only with Apache Kafka itself, but also with the whole Confluent's platform offering. Let's take a look at how to get started using Python for the producer and consumer client examples.

B.1.1 Install

Assuming you are a Python user, you should probably already have moved to Python 3 by now! Otherwise, you will still need to install `librdkafka`. If you are using Homebrew you can use the following command: `brew install librdkafka`. Next, you will need the client package that your code will use as a dependency. The binary wheels package for Confluent Kafka can be installed with pip: `pip install confluent-kafka`. With these prerequisites on your workstation, let's look at building a simple Python producer client.

B.1.2 Python Producer Example

Listing B.1 shows a Python producer client that sends two messages to a topic called `kinaction-python-topic`.

Listing B.1 Python Producer Example

```
from confluent_kafka import Producer    ❶

producer = Producer({'bootstrap.servers': 'localhost:9092'})   ❷

def result(err, message):   ❸
    if err:
        print('%% Producer failure: %s\n' % err)
    else:
        print('Producer info: topic=%s, partition=[%d], offset=%d\n' %
        <linearrow /> (message.topic(), message.partition(), message.offset()))

messages = ["hello python", "hello again"]   ❹

for msg in messages:
    producer.poll(0)
    producer.produce("python-topic", value=msg.encode('utf-8'), callback=result)   ❺

producer.flush()   ❻

# Output   ❼
#Producer info: topic=kinaction-python-topic, partition=[0], offset=8

#Producer info: topic=kinaction-python-topic, partition=[0], offset=9
```

❶ The Confluent package is imported first

❷ The Producer client is configured to connect to a specific Kafka broker

❸ The result function will act as a callback for success and failure handling

❹ The array containing the messages to send

❺ The producer sends all messages to Kafka

❻ Flush is used to make sure that the messages are sent and not only buffered

❼ Sample output showing the metadata about the two sent messages

To use the Confluent package, you first need to make sure that the dependency `confluent_kafka` is imported. A `Producer` client can then be set up with a set of configuration values, including the address of the broker to connect to. The `result` callback is triggered to run

some logic after each call to the `produce` method, whether the call succeeded or failed. The sample code then iterates over the `messages` array to send each message in turn and then calls `flush()` to make sure that the messages are actually sent to the broker as opposed to only being queued to be sent at a later time. Finally, some sample output is printed to the console. Let's now turn to the consuming side and see how it works with Python.

B.1.3 Python Consumer

Listing B.2 shows a sample Kafka consumer client that can be used to read the messages produced by the Python Kafka client in Listing B.2.

Listing B.2 Python Consumer Example

```
from confluent_kafka import Consumer, KafkaError   ❶

consumer = Consumer({
    'bootstrap.servers': 'localhost:9092',   ❷
    'group.id': 'testgroup',
    'auto.offset.reset': 'earliest'
})

consumer.subscribe(['kinaction-python-topic'])   ❸

try:
    while True:
        message = consumer.poll(1.0)   ❹

        if message is None:
            continue
        if message.error():
            print('Error on read: %s' % message.error())
            continue
        else:
            print('Message: %s at offset: %d\n'  %
                (message.value().decode('utf-8'), message.offset()))

except KeyboardInterrupt:
    print('Shutting down consumer\n')
finally:
    consumer.close()   ❺

# Output   ❻
# Message: hello python at offset: 14
```

❶ The Confluent package is imported first

❷ The Consumer client is configured to connect to a specific Kafka broker

❸ The Consumer subscribes to a list of desired topics

❹ Message polling inside an infinite loop

❺ Some clean up to free up resources

❻ The consumed message that is printed out to the console

Similarly to the Producer example in Listing B.1, we first need to make sure that the `confluent_kafka` dependency is declared. A `Consumer` client can then be set up with a set of

configuration values, including the address of the broker to connect to. The consumer client then subscribes to an array of topics it wants to consume messages from, in this case just the single topic named 'python-topic'. The same way as we did with the Java client, we then leverage an infinite loop in which the consumer regularly polls Kafka for new messages! The sample output shows a successful message value as well as the offset of that message. In the event that the consumer is shut down, the `finally` block attempts to gracefully close the client by leaving the consumer group after committing any offsets consumed.

The Python examples provided in this section are very simple but aim at showing non-Java developers that interacting with Kafka can be done with various programing languages. Just remember that not all clients support the same level of features as the Java clients do.

B.2 Client Testing

Testing with `EmbeddedKafkaCluster` was touched on in chapter 7 briefly. Now, we are going to explore a few different alternatives to test Kafka code before deploying it to production.

B.2.1 Unit Level Testing in Java

Unit level testing focuses on testing a single unit of software. This isolated testing should ideally not depend on any other components. However, how is it possible to test such a Java class without connecting to an actual Kafka cluster? If you are familiar with testing frameworks like Mockito, you might decide to create a mock producer object to stand in for the real one. Luckily, the official Kafka client library already provides such a mock named `MockProducer` that implements the `Producer` interface [206]. JUnit 4 can be used and no real Kafka cluster is actually needed in order to verify that the producer logic is working! The mock producer also features a `clear` method that can be called in order to clear the messages that have been recorded by the mock producer, so that other subsequent tests can be run. Luckily, the consumer also has a mocked implementation to leverage as well [207].

B.2.2 Kafka Testcontainers

As also mentioned in chapter 7, Testcontainers (www.testcontainers.org/modules/kafka/) is an option. Whereas the `EmbeddedKafkaCluster` option depended on a process running the Kafka brokers and ZooKeeper nodes in-memory, Testcontainers depends on Docker images.

Notes

1. Moffatt, Robin. The Changing Face of ETL. (2018, September 17). Confluent Blog. Retrieved May 10, 2019, www.confluent.io/blog/changing-face-etl/

2. Introduction. (n.d.). Apache Software Foundation. Retrieved May 30, 2019, from kafka.apache.org/intro

3. Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2020, from kafka.apache.org/documentation/#semantics

4. Narkhede, N. (2017, June 30). Exactly-once Semantics is Possible: Here's How Apache Kafka® Does it. Confluent Blog. Retrieved December 27, 2017, from www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it

5. Narkhede, Neha. Apache Kafka Hits 1.1 Trillion Messages Per Day – Joins the 4 Comma Club. (2015, September 1). Confluent Blog. Retrieved October 20, 2019, from www.confluent.io/blog/apache-kafka-hits-1-1-trillion-messages-per-day-joins-the-4-comma-club/

6. Dauber, L. (2017, May 4). The 2017 Apache Kafka® Survey: Streaming Data on the Rise. Confluent Blog. Retrieved December 23, 2017, from www.confluent.io/blog/2017-apache-kafka-survey-streaming-data-on-the-rise/

7. Waehner, Kai. How to Build and Deploy Scalable Machine Learning in Production with Apache Kafka. (2017, September 29). Retrieved December 11, 2018, from www.confluent.io/blog/build-deploy-scalable-machine-learning-production-apache-kafka/

8. Matsuda, Yasuhiro. Apache Kafka, Purgatory, and Hierarchical Timing Wheels. (2015, October 28). Confluent Blog. Retrieved December 20, 2018, from www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels

9. Use cases. (n.d.). Apache Software Foundation. Retrieved May 30, 2017, from kafka.apache.org/uses

10. Flume 1.9.0 User Guide. (n.d.). Apache Software Foundation. Retrieved May 27, 2017, from flume.apache.org/FlumeUserGuide.html

11. Stopford, Ben. Building a Microservices Ecosystem with Kafka Streams and KSQL. (2017, November 9). Confluent Blog. Retrieved May 1, 2020, www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/

12. Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2020, from kafka.apache.org/documentation/#brokerconfigs_message.max.bytes

13. Documentation. (n.d.). Apache Software Foundation. Retrieved May 22, 2019, from kafka.apache.org/documentation.html#intro_concepts_and_terms

14. Apache Kafka Quickstart. Apache Software Foundation. Retrieved July 15, 2020, from kafka.apache.org/quickstart

15. Apache Kafka Quickstart. Apache Software Foundation. Retrieved July 15, 2020, from kafka.apache.org/quickstart

16. See github.com/apache/kafka/blob/1.0/connect/runtime/src/main/java/org/apache/kafka/connect/runtime/WorkerSourceTas

17. See github.com/apache/kafka/blob/1.0/connect/runtime/src/main/java/org/apache/kafka/connect/runtime/WorkerSinkTask.

18. KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum. (2020, July 09). Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 22, 2020, from cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+

19. A Guide To The Kafka Protocol. (n.d.). Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 21, 2019, from cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-Som

20. Documentation. (n.d.). Apache Software Foundation. Retrieved May 25, 2020, kafka.apache.org/23/documentation.html#intro_topics

21. Svingen, B. (2017, September 6). Publishing with Apache Kafka at The New York Times. Confluent Blog. Retrieved September 25, 2018, from www.confluent.io/blog/publishing-apache-kafka-new-york-times/

22. Microservices Explained by Confluent. (2017, August 23). Conlfuent. Web Presentation. youtu.be/aWI7iU36qv0

23. Clients. Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 21, 2019, from cwiki.apache.org/confluence/display/KAFKA/Clients

24. Class KafkaConsumer<K,V>. (2019, November 09). Apache Software Foundation. Retrieved November 20, 2019, from kafka.apache.org/24/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html

25. Streams Concepts. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/concepts.html

26. MSV, J. (2017, February 15). Apache Kafka: The Cornerstone of an Internet-of-Things Data Platform. Retrieved August 10, 2017, from thenewstack.io/apache-kafka-cornerstone-iot-data-platform/

27. JDBC Source Connector. Confluent Documentation. (n.d.). Retrieved November 22, 2019, from docs.confluent.io/3.1.2/connect/connect-jdbc/docs/source_connector.html

28. Apache Avro™ 1.8.2 Specification. (n.d.). Apache Software Foundation. Retrieved November 19, 2019, from avro.apache.org/docs/1.8.2/spec.html#Schema+Resolution

29. Apache Avro™ 1.8.2 Getting Started (Java). (n.d.). Apache Software Foundation. Retrieved November 19, 2019, from avro.apache.org/docs/1.8.2/gettingstartedjava.html#download_install

30. Apache Avro™ 1.8.2 Getting Started (Java). (n.d.). Apache Software Foundation. Retrieved November 19, 2019, from avro.apache.org/docs/1.8.2/gettingstartedjava.html#Serializing+and+deserializing+without+code+generation

31. Apache Avro™ 1.8.2 Getting Started (Java). (n.d.). Apache Software Foundation. Retrieved November 19, 2019, from avro.apache.org/docs/1.8.2/gettingstartedjava.html#download_install

32. Application Development. Confluent Documentation. (n.d.). Retrieved November 20, 2019, from docs.confluent.io/platform/current/app-development/index.html#java

33. Installation. Confluent Documentation. (n.d.). Retrieved November 20, 2019, from docs.confluent.io/3.1.2/installation.html#maven-repository-for-jars

34. RecordAccumulator.java. (n.d.). Apache Kafka. Github. Retrieved June 17, 2020, from github.com/apache/kafka/blob/299eea88a5068f973dc055776c7137538ed01c62/clients/src/main/java/org/apache/kafka

35. Documentation. (n.d.). Apache Software Foundation. Retrieved May 25, 2020, kafka.apache.org/documentation/#producerconfigs_max.in.flight.requests.per.connection

36. Producer Configurations.Confluent Documentation. (n.d.). Retrieved May 29, 2020, from docs.confluent.io/current/installation/configuration/producer-configs.html#max.in.flight.requests.per.connection

37. KafkaProducer. (n.d.). Apache Software Foundation. Retrieved July 7, 2019, from https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html

38. Documentation. (n.d.). Apache Software Foundation. Retrieved May 9, 2018, from kafka.apache.org/documentation.html#producerconfigs_bootstrap.servers

39. Documentation. (n.d.). Apache Software Foundation. Retrieved May 9, 2018, from kafka.apache.org/documentation.html#producerconfigs_acks

40. Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2020, from kafka.apache.org/documentation/#semantics

41. Documentation. (n.d.). Apache Software Foundation. Retrieved July 22, 2020, from kafka.apache.org/documentation/#topicconfigs_message.timestamp.type

42. Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2020, from kafka.apache.org/documentation/#compression.type

43. KIP-42: Add Producer and Consumer Interceptors. Wiki for Apache Kafka - Apache Software Foundation. Retrieved April 15, 2019, from cwiki.apache.org/confluence/display/KAFKA/KIP-42%3A+Add+Producer+and+Consumer+Interceptors

44. DefaultPartitioner.java. (n.d.). Apache Software Foundation. Github. Retrieved March 22, 2020, from github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/internals/DefaultPartition

45. Documentation. (n.d.). Apache Software Foundation. Retrieved June 19, 2019, from kafka.apache.org/documentation/#consumerconfigs

46. Documentation. (n.d.). Apache Software Foundation. Retrieved December 10, 2019, from kafka.apache.org/23/documentation.html#intro_topics

47. KIP-392: Allow consumers to fetch from closest replica. Wiki for Apache Kafka - Apache Software Foundation. Retrieved December 10, 2019, from cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica

48. Gustafson, J. (2016, January 21). Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client. Confluent Blog. Retrieved June 01, 2020, from www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

49. Rao, Jun. How to choose the number of topics/partitions in a Kafka cluster? (2015, March 12). Confluent Blog. Retrieved May 19, 2019, www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/

50. kafka 2.1.0 API. (n.d.). Apache Software Foundation. Retrieved June 22, 2020, from kafka.apache.org/21/javadoc/?org%2Fapache%2Fkafka%2Fcommon%2Ferrors%2FWakeupException.html

51. Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from kafka.apache.org/documentation/#consumerconfigs_heartbeat.interval.ms

52. Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from kafka.apache.org/documentation/#consumerconfigs_group.id

53. Gustafson, J. (2016, January 21). Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client. Confluent Blog. Retrieved June 01, 2020, from www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

54. Documentation. (n.d.). Apache Software Foundation. Retrieved December 11, 2019, from kafka.apache.org/23/documentation.html#intro_consumers

Gustafson, Jason. Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client. Figure 1. (2016, January 21). Confluent Blog. Retrieved October 11, 2019, from
55. www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

Gustafson, Jason. Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client. (2016, January 21). Confluent Blog. Retrieved October 11, 2019, from
56. www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

Gustafson, Jason. Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client. (2016, January 21). Confluent Blog. Retrieved October 11, 2019, from
57. www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

Documentation. (n.d.). Apache Software Foundation. Retrieved December 22, 2020, from
58. kafka.apache.org/documentation/#consumerconfigs_partition.assignment.strategy

Release Plan 0.11.0.0. Wiki for Apache Kafka - Apache Software Foundation. Retrieved December 14,
59. 2019, from cwiki.apache.org/confluence/display/KAFKA/Release+Plan+0.11.0.0

Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from
60. kafka.apache.org/documentation/#consumerconfigs_enable.auto.commit

Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from
61. kafka.apache.org/documentation/#consumerconfigs_auto.offset.reset

Post Kafka Deployment. Confluent Documentation. (n.d.). Retrieved September 15, 2019, from
62. docs.confluent.io/platform/current/kafka/post-deployment.html#balancing-replicas-across-racks

KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum. (2020, July 09). Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 22, 2020, from
63. cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+

Junqueira, Flavio; Narkhede, N. "Distributed Consensus Reloaded: Apache ZooKeeper and Replication in Apache Kafka." Confluent Blog. (2015, August 27). Retrieved September 15, 2019, from
64. www.confluent.io/blog/distributed-consensus-reloaded-apache-zookeeper-and-replication-in-kafka/

Kafka data structures in Zookeeper. Wiki for Apache Kafka - Apache Software Foundation. Retrieved
65. January 19, 2020, from cwiki.apache.org/confluence/display/KAFKA/Kafka+data+structures+in+Zookeeper

ZooKeeper. (n.d.). Apache Software Foundation. Retrieved September 15, 2019, from
66. zookeeper.apache.org/doc/r3.2.2/zookeeperOver.html

KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum. (2020, July 09). Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 22, 2020, from
67. cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+

A Guide To The Kafka Protocol. (2017, June 14). Wiki for Apache Kafka - Apache Software Foundation. Retrieved September 15, 2019, from
68. cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol

A Guide To The Kafka Protocol. (2017, June 14). Wiki for Apache Kafka - Apache Software Foundation. Retrieved September 15, 2019,
69. cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#GuideToTheKafkaProtocol-Fetch/

Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from
70. kafka.apache.org/documentation/#brokerconfigs

Documentation. (n.d.). Apache Software Foundation. Retrieved September 15, 2019, from
71. kafka.apache.org/documentation/#brokerconfigs

Kafka Controller Internals. (2014, January 26). Wiki for Apache Kafka - Apache Software Foundation. Retrieved September 15, 2019, from
72. cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals

Kafka Controller Internals. (2014, January 26). Wiki for Apache Kafka - Apache Software Foundation. Retrieved September 15, 2019, from
73. cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals

Post Kafka Deployment. Confluent Documentation. (n.d.). Retrieved July 10, 2019, from
74. docs.confluent.io/platform/current/kafka/post-deployment.html#rolling-restart

Narkhede, Neha. "Hands-free Kafka Replication: A lesson in operational simplicity." Confluent Blog. (2015, July 1). Retrieved October 02, 2019, from
75. www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/

KIP-392: Allow consumers to fetch from closest replica. Wiki for Apache Kafka - Apache Software Foundation. Retrieved December 10, 2019, from
76. cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica

Documentation. (n.d.). Apache Software Foundation. Retrieved October 02, 2019, from
77. kafka.apache.org/documentation/#replication

Documentation. (n.d.). Apache Software Foundation. Retrieved October 02, 2019, from
78. kafka.apache.org/documentation/#design_uncleanleader

The Grafana Word Mark and Grafana Logo are either registered trademarks/service marks or trademarks/service marks of Coding Instinct AB, in the United States and other countries and are used with Coding Instinct's permission. We are not affiliated with, endorsed or sponsored by Coding Instinct, or the
79. Grafana community.

80. Documentation. (n.d.). Apache Software Foundation. Retrieved October 02, 2019, from kafka.apache.org/documentation/#monitoring

81. Running Kafka in Production. Confluent Documentation. (n.d.). Retrieved May 16, 2021, from docs.confluent.io/platform/current/kafka/deployment.html

82. Post Kafka Deployment. Confluent Documentation. (n.d.). Retrieved September 15, 2019, from docs.confluent.io/current/kafka/post-deployment.html#scaling-the-cluster

83. Documentation. (n.d.). Apache Software Foundation. Retrieved May 11, 2018, from kafka.apache.org/documentation/#basic_ops_restarting

84. Post Kafka Deployment. Confluent Documentation. (n.d.). Retrieved September 15, 2019, from docs.confluent.io/current/kafka/post-deployment.html#rolling-restart

85. McCabe, Colin. "Upgrading Apache Kafka Clients Just Got Easier." Confluent Blog. (2017, July 18). Retrieved October 02, 2019, from www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier

86. Documentation. (n.d.). Apache Software Foundation. Retrieved September 15, 2019, from kafka.apache.org/documentation/#georeplication

87. Apache Kafka's MirrorMaker. Confluent Documentation. (n.d.). Retrieved April 22, 2019, from docs.confluent.io/4.0.0/multi-dc/mirrormaker.html

88. Release Notes - Kafka - Version 2.4.0. (n.d.). Apache Software Foundation. Retrieved May 12, 2020, from archive.apache.org/dist/kafka/2.4.0/RELEASE_NOTES.html

89. connect-mirror-maker.properties. Github for Apache Kafka. Retrieved January 15, 2020, from github.com/apache/kafka/blob/trunk/config/connect-mirror-maker.properties

90. KIP-382: MirrorMaker 2.0. (2019, December 11). Wiki for Apache Kafka - Apache Software Foundation. Retrieved January 15, 2020, from cwiki.apache.org/confluence/display/KAFKA/KIP-382%3A+MirrorMaker+2.0

91. Shapira, Gwen. (2018). Recommendations_for_Deploying_Apache_Kafka_on_Kubernetes. White paper. Retrieved December 15, 2019 from Confluent, Inc.: www.confluent.io/resources/recommendations-for-deploying-apache-kafka-on-kubernetes

92. Replication tools. Wiki for Apache Kafka - Apache Software Foundation. Retrieved January 19, 2019, from cwiki.apache.org/confluence/display/kafka/replication+tools

93. Rao, Jun. How to choose the number of topics/partitions in a Kafka cluster? (2015, March 12). Confluent Blog. Retrieved May 19, 2019, www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/

Documentation. (n.d.). Apache Software Foundation. Retrieved May 19, 2018, from
94. kafka.apache.org/documentation/#basic_ops_modify_topic

Documentation. (n.d.). Apache Software Foundation. Retrieved December 11, 2019, from
95. kafka.apache.org/documentation/#basic_ops_add_topic

Kafka Broker Configurations. Confluent Documentation. (n.d.). Retrieved January 15, 2021, from
96. docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_delete.topic.enable

Documentation. (n.d.). Apache Software Foundation. Retrieved December 19, 2019, from
97. kafka.apache.org/documentation/#brokerconfigs_auto.create.topics.enable

Documentation. (n.d.). Apache Software Foundation. Retrieved December 18, 2019, from
98. kafka.apache.org/documentation/#basic_ops_modify_topic

KIP-279: Fix log divergence between leader and follower after fast leader fail over. Wiki for Apache Kafka - Apache Software Foundation. Retrieved August 22, 2019, from
99. cwiki.apache.org/confluence/display/KAFKA/KIP-279%3A+Fix+log+divergence+between+leader+and+follower+aft

Documentation. (n.d.). Apache Software Foundation. Retrieved December 11, 2019, from
100. kafka.apache.org/documentation/#design_compactionbasics

System Tools. (2017, October 27). Wiki for Apache Kafka - Apache Software Foundation. Retrieved September 15, 2019, from
101. cwiki.apache.org/confluence/display/KAFKA/System+Tools#SystemTools-DumpLogSegment

Documentation. (n.d.). Apache Software Foundation. Retrieved May 19, 2018, from
102. kafka.apache.org/documentation/#basic_ops_modify_topic

Replication tools. Wiki for Apache Kafka - Apache Software Foundation. Retrieved January 15, 2018,
103. cwiki.apache.org/confluence/display/KAFKA/Replication+tools#Replicationtools-1.PreferredReplicaLeaderElection

Documentation. (n.d.). Apache Software Foundation. Retrieved December 11, 2019, from
104. kafka.apache.org/documentation.html#upgrade_240_notable

For reference see:
105. github.com/apache/kafka/blob/9af81955c497b31b211b1e21d8323c875518df39/streams/src/test/java/org/apache/kafk

106. Bejeck Jr., William P. Kafka Streams in Action. Shelter Island, NY: Manning, 2018

Documentation. (n.d.). Apache Software Foundation. Retrieved May 29, 2018, from
107. kafka.apache.org/documentation/#compaction

Topic Configurations. Confluent Documentation. (n.d.). Retrieved November 22, 2020, from
108. docs.confluent.io/platform/current/installation/configuration/topic-configs.html#topicconfigs_cleanup.policy

Documentation. (n.d.). Apache Software Foundation. Retrieved December 11, 2019, from
109. kafka.apache.org/documentation/#design_compactionbasics

Kafka Design. Confluent Documentation. (n.d.). Retrieved November 19, 2020, from
110. docs.confluent.io/platform/current/kafka/design.html#log-compaction

Documentation. (n.d.). Apache Software Foundation. Retrieved May 29, 2018, from
111. kafka.apache.org/documentation/#brokerconfigs_log.roll.hours

Svingen, B. (2017, September 6). Publishing with Apache Kafka at The New York Times. Confluent Blog.
112. Retrieved September 25, 2018, from www.confluent.io/blog/publishing-apache-kafka-new-york-times/

Kafka Broker Configurations. Confluent Documentation. (n.d.). Retrieved December 14, 2020, from
113. docs.confluent.io/platform/current/installation/configuration/broker-configs.html

Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2018, from
114. kafka.apache.org/documentation/#brokerconfigs_log.retention.ms

Flume 1.9.0 User Guide. (n.d.). Apache Software Foundation. Retrieved October 10, 2019, from
115. flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#kafka-sink

Pinterest. “Pinterest Secor.” GitHub. Retrieved June 1, 2020, from
116. github.com/pinterest/secor/blob/master/README.md

Tiered Storage. Confluent Documentation. (n.d.). Retrieved June 2, 2021 from,
117. docs.confluent.io/platform/current/kafka/tiered-storage.html

Marz, Nathan, and James Warren. Big Data: Principles and best practices of scalable real-time data systems.
118. Shelter Island, NY: Manning, 2015

Kreps, Jay. Questioning the Lambda Architecture. (2014, July 2). O’Reilly Radar. Visited October 11, 2019,
119. from www.oreilly.com/radar/questioning-the-lambda-architecture/

Wang, Allen. Multi-Tenant, Multi-Cluster and Hierarchical Kafka Messaging Service. Confluent. Kafka Summit San Francisco 2017 Presentation.
120. www.confluent.io/kafka-summit-sf17/multitenant-multicluster-and-hieracrchical-kafka-messaging-service/

Fowler, Martin. CQRS. (2011, July 14). Retrieved December 11, 2017, from
121. martinfowler.com/bliki/CQRS.html

Architecture Patterns for Multi-Region Clusters. Confluent Documentation. (n.d.). Retrieved September 15, 2019, from docs.confluent.io/platform/current/multi-dc-deployments/multi-region-architectures.html

Architecture Patterns for Multi-Region Clusters. Confluent Documentation. (n.d.). Retrieved September 15, 2019, from docs.confluent.io/platform/current/multi-dc-deployments/multi-region-architectures.html#region-active-active

Loddengaard, Alex. Design and Deployment Considerations for Deploying Apache Kafka on AWS. (2016, July 28). Confluent Blog. Retrieved June 11, 2021, from www.confluent.io/blog/design-and-deployment-considerations-for-deploying-apache-kafka-on-aws/

KIP-392: Allow consumers to fetch from closest replica. Wiki for Apache Kafka - Apache Software Foundation. Retrieved December 10, 2019, from cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica

cp-helm-charts. (n.d.). Confluent Inc. Github. Retrieved June 10, 2020, from github.com/confluentinc/cp-helm-charts

AdminClient. Confluent Documentation. (n.d.). Retrieved November 17, 2020, from docs.confluent.io/5.3.1/clients/javadocs/index.html?org/apache/kafka/clients/admin/AdminClient.html

Confluent REST APIs. Confluent Documentation. (n.d.). Retrieved February 20, 2019, from docs.confluent.io/platform/current/kafka-rest/index.html#features

REST Proxy Quick Start. Confluent Documentation. (n.d.). Retrieved February 22, 2019, from docs.confluent.io/platform/current/kafka-rest/quickstart.html

Using Confluent Platform systemd Service Unit Files. Confluent Documentation. (n.d.). Retrieved January 15, 2021, from docs.confluent.io/platform/current/installation/scripted-install.html#overview

RollingFileAppender. Apache Logging. Retrieved April 22, 2020, from logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/RollingFileAppender.html

log4j.properties. (n.d.). Apache Kafka. Github. Retrieved June 17, 2020, from github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/config/log4j.properties

ZooKeeper Administrator's Guide. (n.d.). Apache Software Foundation. Retrieved June 10, 2020, from zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html

ZooKeeper Getting Started Guide. Apache Software Foundation. Retrieved August 19, 2020, from zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html

Moffatt, Robin. Kafka Listeners – Explained. (2019, July 1). Confluent Blog. Retrieved June 11, 2020, from www.confluent.io/blog/kafka-listeners-explained/

136. Kafka Monitoring and Metrics Using JMX. Confluent Documentation. (n.d.). Retrieved June 12, 2020, from docs.confluent.io/platform/current/installation/docker/operations/monitoring.html

137. Monitoring Kafka. Confluent Documentation. (n.d.). Retrieved May 1, 2020, from docs.confluent.io/5.4.0/kafka/monitoring.html#broker-metrics

138. Monitoring Kafka. Confluent Documentation. (n.d.). Retrieved May 1, 2020, from docs.confluent.io/platform/current/kafka/monitoring.html

139. Monitoring Kafka. Confluent Documentation. (n.d.). Retrieved April 22, 2019, docs.confluent.io/platform/current/kafka/monitoring.html#producer-metrics

140. Monitoring Kafka. Confluent Documentation. (n.d.). Retrieved May 1, 2020, from docs.confluent.io/5.4.0/kafka/monitoring.html#fetch-metrics

141. Interface ProducerInterceptor. Apache Software Foundation. Retrieved June 1, 2020, from kafka.apache.org/27/javadoc/org/apache/kafka/clients/producer/ProducerInterceptor.html

142. Documentation. (n.d.). Apache Software Foundation. Retrieved May 1, 2020, from kafka.apache.org/documentation/#monitoring

143. Running Kafka in Production. Confluent Documentation. (n.d.). Retrieved January 10, 2020, from docs.confluent.io/platform/current/kafka/deployment.html

144. Post Kafka Deployment. Confluent Documentation. (n.d.). Retrieved October 11, 2020, from docs.confluent.io/platform/current/kafka/post-deployment.html#tuning-virtual-memory

145. Security Overview. Confluent Documentation. (n.d.). Retrieved June 23, 2019, from docs.confluent.io/platform/current/security/general-overview.html

146. Encryption and Authentication with SSL. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_ssl.html

147. Security Tutorial. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/security_tutorial.html

148. Security Tutorial. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/security_tutorial.html

149. Security Tutorial. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/security_tutorial.html

150. See github.com/confluentinc/confluent-platform-security-tools/blob/master/kafka-generate-ssl.sh

151. Encryption and Authentication with SSL. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_ssl.html

152. Encryption and Authentication with SSL. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_ssl.html

153. Encryption and Authentication with SSL. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_ssl.html

154. Documentation. (n.d.). Apache Software Foundation. Retrieved June 1, 2020, from kafka.apache.org/24/documentation.html#security_rolling_upgrade

155. Brennen, V. Alex. "An Overview of a Kerberos Infrastructure." Kerberos Infrastructure HOWTO. tldp.org/HOWTO/Kerberos-Infrastructure-HOWTO/overview.html

156. Configuring GSSAP. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_sasl/authentication_sasl_gssapi.html

157. Configuring GSSAP. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_sasl/authentication_sasl_gssapi.html

158. Configuring GSSAP. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authentication_sasl/authentication_sasl_gssapi.html

159. Security Tutorial. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/security_tutorial.html

160. Authorization using ACLs. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authorization.html

161. Authorization using ACLs. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authorization.html

162. Authorization using ACLs. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authorization.html

163. Authorization using Role-Based Access. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/rbac/index.html

164. ZooKeeper Security. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/zk-security.html

165. Authorization using ACLs. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/kafka/authorization.html

166. ZooKeeper Security. Confluent Documentation. (n.d.). Retrieved June 27, 2021, from docs.confluent.io/5.2.1/security/zk-security.html

167. ZooKeeper Security. Confluent Documentation. (n.d.). Retrieved June 10, 2020, from docs.confluent.io/platform/current/security/zk-security.html

168. Documentation. (n.d.). Apache Software Foundation. Retrieved June 1, 2020, from kafka.apache.org/documentation/#brokerconfigs_quota.producer.default

169. Documentation. (n.d.). Apache Software Foundation. Retrieved June 19, 2020, from kafka.apache.org/documentation/#design_quotasbandwidth

170. Documentation. (n.d.). Apache Software Foundation. Retrieved June 15, 2020, from kafka.apache.org/documentation/#quotas

171. Documentation. (n.d.). Apache Software Foundation. Retrieved June 15, 2020, from kafka.apache.org/documentation/#quotas

172. Documentation. (n.d.). Apache Software Foundation. Retrieved May 25, 2019, from kafka.apache.org/documentation/#design_quotasconfig

173. Documentation. (n.d.). Apache Software Foundation. Retrieved June 19, 2020, from kafka.apache.org/documentation/#design_quotascpu

174. KIP-124 - Request rate quotas. (2017, March 30). Wiki for Apache Kafka - Apache Software Foundation. Retrieved June 1, 2020, from cwiki.apache.org/confluence/display/KAFKA/KIP-124+-+Request+rate+quotas

175. Fowler, Martin. Maturity Model. (2014, August 26). Retrieved June 15, 2021, from martinfowler.com/bliki/MaturityModel.html

176. Fowler, Martin. Richardson Maturity Model. (2010, March 18). Retrieved June 15, 2021, from martinfowler.com/articles/richardsonMaturityModel.html

177. The website text is noted as licensed under the Creative Community License (creativecommons.org/licenses/by-sa/2.0/legalcode)

178. Hedderly, Lyndon. (2018). Five Stages to Streaming Platform Adoption. White paper. Retrieved January 15, 2020 from Confluent, Inc.: www.confluent.io/resources/5-stages-streaming-platform-adoption/

179. Schema Registry Overview. Confluent Documentation. (n.d.). Retrieved July 15, 2020, from docs.confluent.io/platform/current/schema-registry/index.html

180. Running Schema Registry in Production. Confluent Documentation. (n.d.). Retrieved April 25, 2019, docs.confluent.io/platform/current/schema-registry/installation/deployment.html#schema-registry-prod

181. Installing and Configuring Schema Registry. Confluent Documentation. (n.d.). Retrieved July 15, 2020, from docs.confluent.io/platform/current/schema-registry/installation/index.html

182. Using Schema Registry. Confluent Documentation. (n.d.). Retrieved June 5, 2020, from docs.confluent.io/4.1.1/schema-registry/docs/using.html#starting-sr

183. Schema Registry Overview. Confluent Documentation. (n.d.). Retrieved July 15, 2020, from docs.confluent.io/platform/current/schema-registry/index.html

184. Schema Registry API Reference. Confluent Documentation. (n.d.). Retrieved July 15, 2020, from docs.confluent.io/platform/current/schema-registry/develop/api.html#schemaregistry-api

185. Formats, Serializers, and Deserializers. Confluent Documentation. (n.d.). Retrieved April 25, 2019, docs.confluent.io/platform/current/schema-registry/serdes-develop/index.html

186. On-Premises Schema Registry Tutorial. Confluent Documentation. (n.d.). Retrieved April 25, 2019, docs.confluent.io/platform/current/schema-registry/schema_registry_onprem_tutorial.html

187. Schema Evolution and Compatibility. Confluent Platform. Retrieved June 1, 2020, from docs.confluent.io/current/schema-registry/avro.html#compatibility-types

188. Schema Validation on Confluent Server. Confluent Documentation. (n.d.). Retrieved July 15, 2020, from docs.confluent.io/platform/current/schema-registry/schema-validation.html

189. Schema Registry Maven Plugin. Confluent Documentation. (n.d.). Retrieved July 16, 2020, from docs.confluent.io/platform/current/schema-registry/develop/maven-plugin.html

190. Documentation. (n.d.). Apache Software Foundation. Retrieved May 30, 2021, from kafka.apache.org/documentation/streams/

191. Streams Concepts. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/concepts.html

192. Streams Concepts. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/concepts.html

193. Streams Concepts. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/concepts.html

194. Streams Concepts. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/concepts.html

195. Streams Architecture. Confluent Documentation. (n.d.). Retrieved June 17, 2020, from docs.confluent.io/platform/current/streams/architecture.html

196. Streams: Core Concepts. Apache Software Foundation. Retrieved June 25, 2021 from, kafka.apache.org/26/documentation/streams/core-concepts

197. Queries. Confluent Documentation. (n.d.). Retrieved May 5, 2021, from docs.ksqldb.io/en/latest/concepts/queries/

198. Queries. Confluent Documentation. (n.d.). Retrieved May 5, 2021, from docs.ksqldb.io/en/latest/concepts/queries/

199. Installing ksqlDB. Confluent Documentation. (n.d.). Retrieved June 20, 2020, from docs.confluent.io/platform/current/ksqldb/installing.html

200. confluent local services ksql-server start. Confluent Documentation. (n.d.). Retrieved June 1, 2021, from docs.confluent.io/confluent-cli/current/command-reference/local/services/ksql-server/confluent_local_services_ksql-

201. Galasyn, Jim. How to Run Confluent on Windows in Minutes. (2021, March 26). Confluent Blog. Retrieved June 11, 2021, from www.confluent.io/blog/set-up-and-run-kafka-on-windows-and-wsl-2/

202. docker.com

203. Galasyn, Jim. How to Run Confluent on Windows in Minutes. (2021, March 26). Confluent Blog. Retrieved June 11, 2021, from www.confluent.io/blog/set-up-and-run-kafka-on-windows-and-wsl-2/

204. Kafka Clients. Confluent Documentation. (n.d.). Retrieved June 15, 2020, from docs.confluent.io/current/clients/index.html

205. confluent-kafka-python. (n.d.). Confluent Inc. Github. Retrieved June 12, 2020, from github.com/confluentinc/confluent-kafka-python

206. MockProducer (kafka 2.7.0 API). (n.d.). Apache Software Foundation. Retrieved May 30, 2021, from kafka.apache.org/27/javadoc/org/apache/kafka/clients/producer/MockProducer.html

207. MockConsumer (kafka 2.7.0 API). (n.d.). Apache Software Foundation. Retrieved May 30, 2021, kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/MockConsumer.html