Kafka IN ACTION

Dylan D. Scott

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kafka in Action
Version 6

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 Author Online 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

In this chapter, we will:

- Introduce why you would want to use Kafka
- Address common myths in relation to Hadoop and message systems
- Understand Real World Use Cases where Kafka helps power messaging, website activity tracking, log aggregation, stream processing, and IoT data processing

From a certain point onward there is no longer any turning back. That is the point tha must be reached.

--

Franz Kafka

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. While Kafka, who is quoted above, was addressing something far different than IT infrastructures, the wording applies to the situation we find ourselves in today with the emergence of his namesake Apache Kafka. Kafka is changing the standard for data platforms. It is leading the way to move from Batch and ETL workflows to the near real-time data feeds. 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 inundated 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 state of the system.

Kafka also is starting to make a push into microservice design. Kafka is touching a lot

of the newest and most practical trends in today's IT fields as well as making its way into many users daily work. As a de-facto technology in 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 capabilites: Provide the ability to publish/subscribe to records like a message queue, store records with fault-tolerance, and process streams as they occur.

For readers who are not as familiar with queues or message brokers in their daily work, it might be helpful to discuss the general purpose and flow of a publish/subscribe system. As a generalization, a core piece of Kafka can be thought of providing the IT equivalent of a receiver that sits in a home entertainment system.

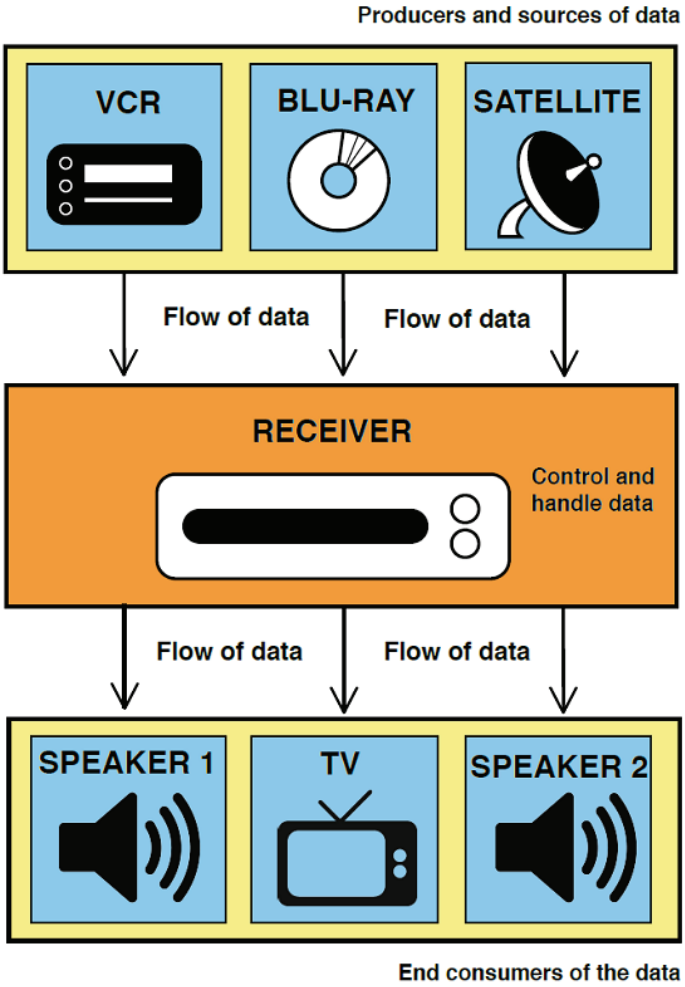

Figure 1.1 Receiver overview

As shown above, digital satellite, cable, and blu-ray players can all connect to this central receiver. Simplifying of course, you can think of those individual pieces as

constantly 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; ie. the receiver sends the video on to your television and audio to a decoder as well as on to speakers.

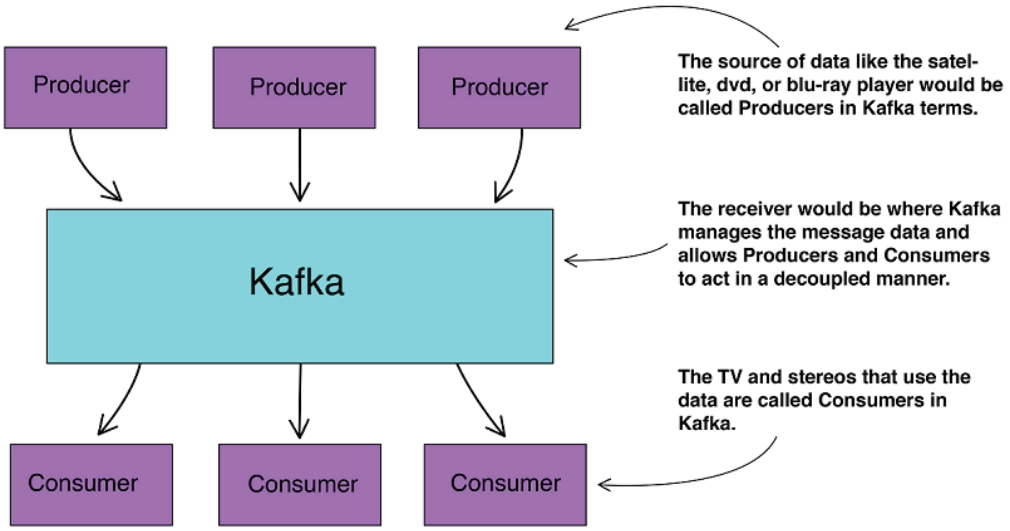

Figure 1.2 Kafka flow overview

So what does this have to do with Kafka exactly? Kafka has the concepts of sending data from various sources of information and introducing them into Kafka called Producers. In addition, Kafka allows multiple output channels that are enabled by Consumers. The receiver can be thought of as the message brokers themselves that can handle a real-time stream of messages. Decoding isn't the right term for Kafka, but we will dig into those details in later chapters.

Kafka, as does other message brokers, acts, in reductionist terms, as a middle man to data coming into the system (from producers) and out of the system (consumers). By allowing this separation between the producer and end-user of the message, loose coupling can be achieved. 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:

- At least once semantics
- At most once semantics
- Exactly once semantics

Let's dig into what those messaging options mean.

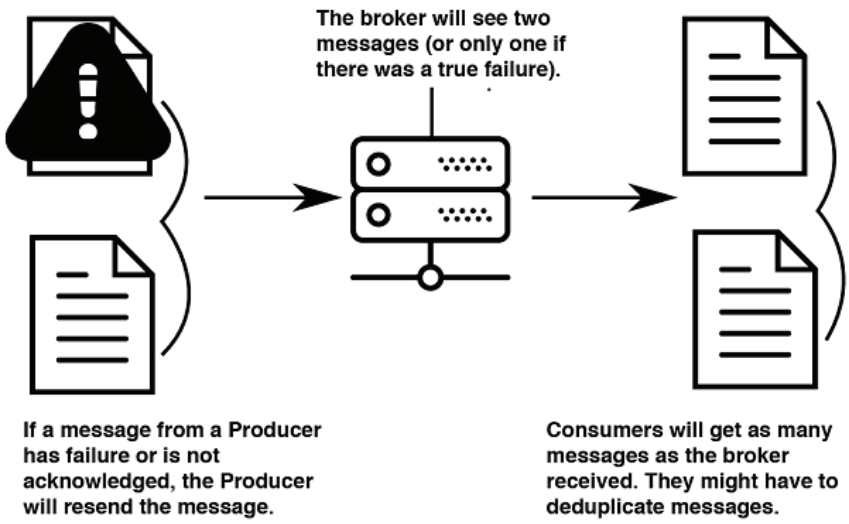

Figure 1.3 At Least Once message flow semantics

Kafka's default guarantee is at least once semantics. This means that Kafka can be configured to allow for 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 in order 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 methods for delivery.

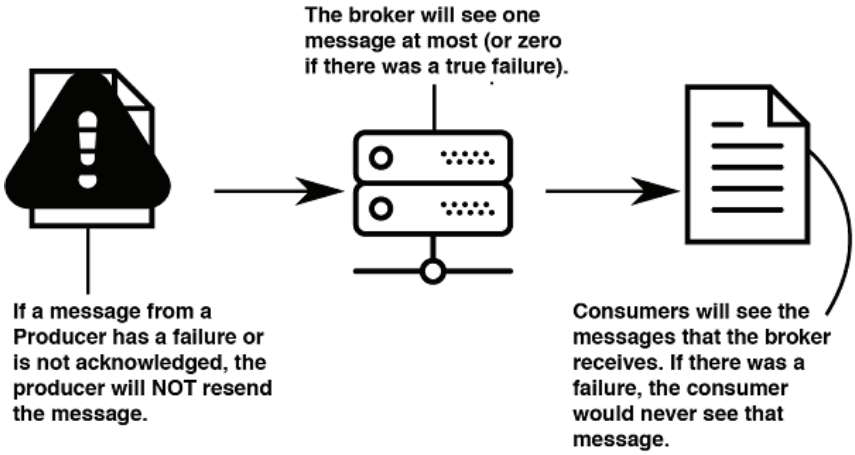

Figure 1.4 At Most Once message flow semantics

Moving on, at most once semantics are when a producer of messages might send a message once and never retires. 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 fine with missing a few page few events out of the millions they do process per day. Keeping the system performing and not waiting on acknowledgements might outweigh any gain from lost data.

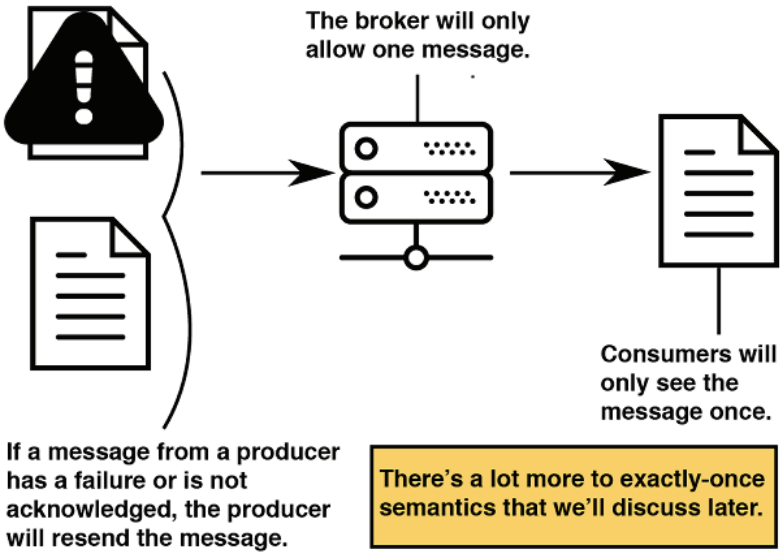

Figure 1.5 Exactly Once message flow semantics

Exactly once semantics (EOS) are one of the newest additions to the Kafka feature set and generated a lot of discussion with its release. In the context of a Kafka system, if a producer sends a message more than once, it would still be delivered once to the end consumer. EOS has touch points at all layers of Kafka, 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 be able to pick up where they left off and not drop messages.

1.2 Why the need for Kafka?

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

Some companies, including Netflix and LinkedIn, have already reached the level of processing over 1 trillion messages per day with their deployments. Millions of messages per second are possible in production settings; all with a tool that was not at its 1.0 version release (which finally occurred in October of 2017). Their enormous data processing needs are handled with their production usage of Kafka. These examples show the potentially 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 Why 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. [1] A shift in traditional data process thinking is needed and various shared experiences or past pain

points can help developers see why Kafka could be an appealing step forward in their data architectures.

Footnote 1 www.confluent.io/blog/2017-apache-kafka-survey-streaming-data-on-the-rise/

One of the various on-ramps for newer developers to Kafka is a way to apply things they know to help them with the unknown. For Java developers used to Spring concepts and dependency injection (DI) Spring 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 pretty common shared milestone, most programmers have experienced the pain 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 the large amount of mocks you are having to create. Kafka, when applied thoughtfully, can help in this situation.

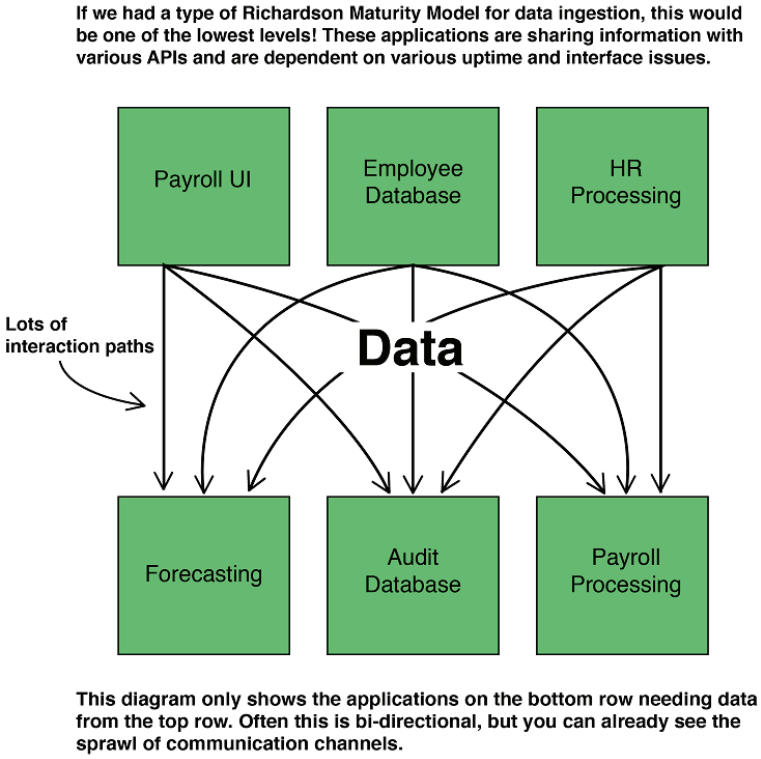

Figure 1.6 Before Kafka message flow

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 burn down charts for forecasting work. So do you tie the two applications together? What if the payroll system was down? Should that really 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.

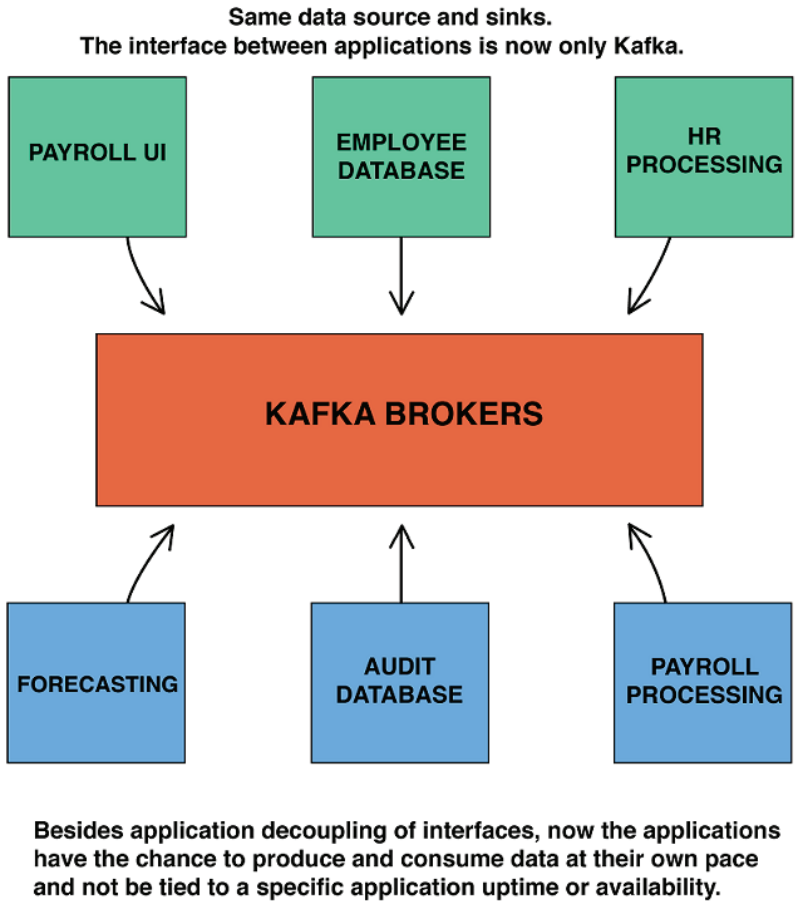

Figure 1.7 After Kafka message flow

With the above view of putting Kafka in the middle of the flow, your interface to data becomes Kafka instead of various APIs and databases.

Some will say that there are pretty simple solutions. What about using ETL to at least load the data into its own 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. When we dig into Consumers in Chapter 5, we will see how Kafka uses topics under the hood to manage commit offsets. And in release 0.11, exactly once semantics for Kafka also uses internal topics that are leveraged by the Transaction Coordination. The ability to have many consumers of the data use the same message is better than a choose your own adventure game, as all outcomes are possible.

Another developer question might be: Why not learn Kafka Streams, 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 really believe that Kafka itself is worth learning. There is a difference in knowing that Kafka is a channel option for 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 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.

From a purely technical viewpoint, there are interesting 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 (www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels -PROD will shorten). Taking a peek into the core.src.main.scala.kafka.utils.timer.Timer.scala class and you can see 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 being able to dig into the source code and also have documentation and examples just by searching the internet. This alone can be worth avoiding the Not Invented Here syndrome.

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 really does. As conferences have taken to making the case for attendees to help convince their boss about attending conferences, it might be nice to do the same for explaining the value found in this product as well. Also, it really 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 the most important features of Kafka is the ability to take volumes of data and make it available for use by various business units. 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, 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 there is real value focus 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 key driver for some whose data requires on-site oversight. And the Cloud is also an option as well. It can grow horizontally and not depend 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 startup and other disrupters in their industry are able to 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 are first starting 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 bonus 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 own products. Spark Streaming and Flume are examples of tools that use Kafka. The dependency on ZooKeeper is also a tool that is often found in Hadoop clusters and might tie Kafka further to this myth.

One other key myth that often appears is that Kafka requires the Hadoop Filesystem - HDFS. Once we start to dig into how Kafka works, we see the speed of Kafka is achieved with techniques that would likely be slowed down with a Node Manager in the middle of the process. Also, the replications that are usually a part of HDFS would be even more replications if your topics ever used more than one replica. The ideas of replication also lead to another logical group of confusion, 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).

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 or MQSeries to Kafka often have asterisks (or fine print) attached. Some tools over time have gained or will gain new features just as Kafka has added 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 interesting 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 that has already seen that message can choose to read that (and other messages) again. With some

architecture models such as Lambda, programmer mistakes are expected just as much as hardware failure. Imagine consuming millions of messages you forgot to use a specific field from the original message. In some queues, that message would have been removed or have been sent to a duplicate or replay location. However, Kafka provides a way for Consumers to seek to specific points and can read messages (with a few constraints) again by just seeking to 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 exact topic. With the concept of consumers being part of a Consumer Group, membership in a group determines which consumers get which messages and what work has been done in that group of consumers. Consumer groups act independently of the other groups and allow for multiple applications to consume messages at their own pace with as many consumers as they require. So the processing can happen in a couple of 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 powerful use cases that have made Kafka one of the default options developers turn to for getting work done.

1.4 Real World Use Cases

I am large, I contain multitudes.

-- Walt Whitman

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 really has many specific uses in which it can 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.

1.4.1 Messaging

Many 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 the Extensible Messaging and Presence Protocol (XMPP), the Advanced Message Queuing Protocol (AMQP), or Java Message Service (JMS) API in favor of a custom TCP binary protocol.

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

Having a durable channel for sending messages is also why Kafka is used. Often times, memory storage of data 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. In fact, 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. Kafka enables robust applications to be built and helps handle the expected failures that distributed applications are bound to run into at some point.

1.4.2 Website Activity Tracking

Part of Kafka's origin story is serving as a way to gather information about users on Linkedin's website. Kafka's ability to handle large volumes of data was a key requirement since a website with many visitors had the need to handle the possibility of multiple events per page view. The flood of data had to be dealt with fast and able to scale without impacting the end-user's experience on the site. Did a user click on a link or visit the new marketing page? Those frequent user events can be fed into Kafka and made quickly available to various applications to get an overall view of the paths consumers take through a shopping site (for instance). Imagine the ability to do a real-time feed of data to show customers recommended products on the next page click as one example of the importance of speed in a e-commerce flow.

1.4.3 Log Aggregation

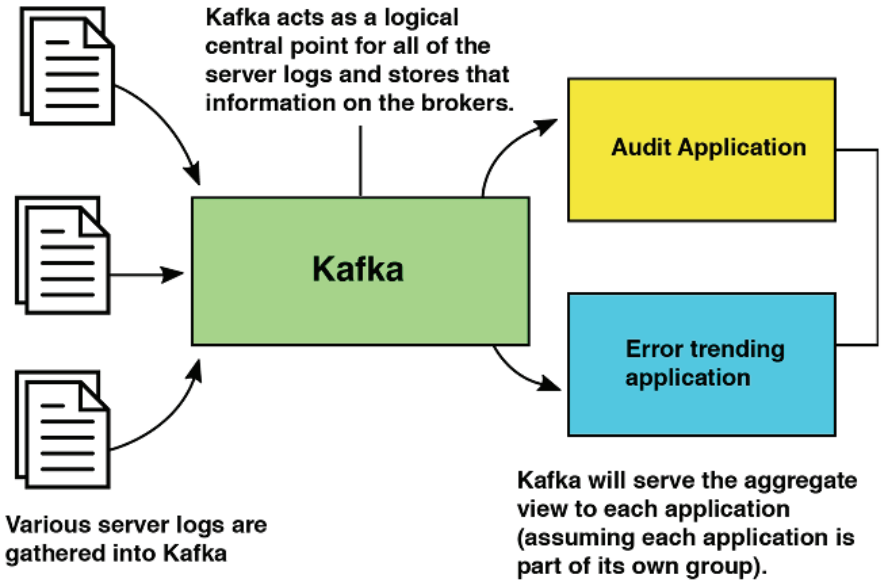

Figure 1.8 Kafka Log Aggregation

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 logical single 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 as well as failure detection trending. Kafka is also used in various logging tools (or as an input option) such as Graylog (www.graylog.org).

How do all of these log files 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 in Chapter 4 and more on the log format of messages in Chapter 6 and 7.

1.4.4 Stream Processing

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 were introduced. We will cover what that really means in practice later on once we get a more solid foundation. However, for users who are running end-to-end workloads inside Kafka with the Streams API, message delivery now is more of what most users would find as ideal. Imagine banks using Kafka to debit or credit your account. With at least once semantics, you would have to ensure that a debit was not completed twice (due to duplication of messages) through various consumer side logic. Streams makes 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 was only processed once from the beginning to the end of the transaction.

1.4.5 Internet of Things

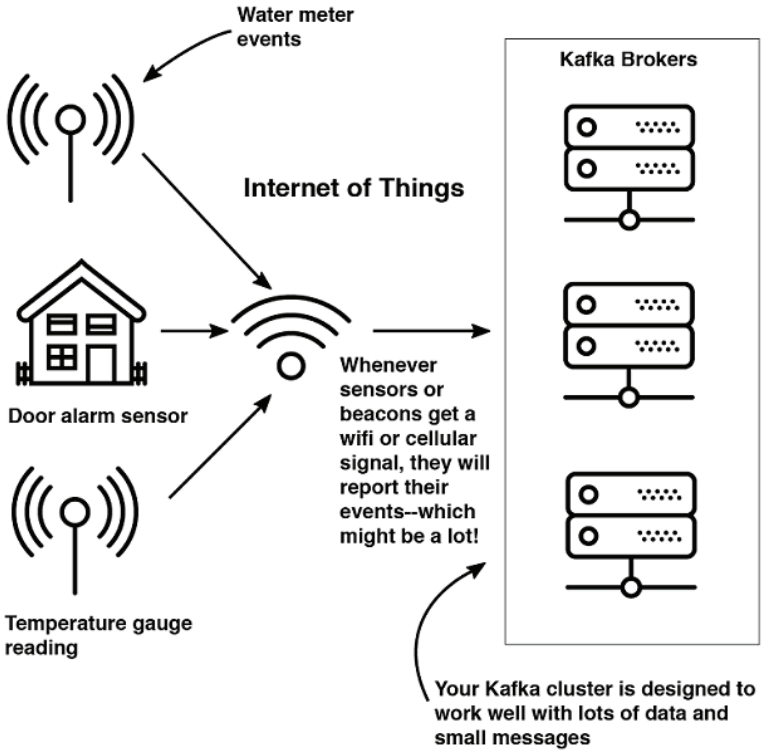

Figure 1.9 IoT

The number of internet-connected devices seems like the Internet of Things will have no where to go but up. With all of those devices sending messages, sometimes in bursts whenever they get a wifi or cellular connection, something needs to be able to handle that data effectively. As you may have gathered, massive quantities of data is one of the key areas where Kafka really 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 important to be able to see how even more practical applications are possible.

1.4.6 When Kafka might not be the right fit

It is important to note that while Kafka has shown some really interesting use cases, it is not always the best tool for the job at hand. Let's take a quick look at 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? If you don't need an on-demand view, quick answer, or even the ability to reprocess data, then you might not need Kafka running throughout the entire year for that

task alone. Especially, if that amount of data is manageble to process at once as a batch (as always, your mileage may vary: different users have different thresholds on what is large batch).

If your main access pattern for data is 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 off of.

Similarly, if you need 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 would have to be taken to make sure that only 1 producer request thread is the max at one time and that there is only 1 partition in the topic. There are various workarounds, but if you have huge amounts of data that depends 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 into mind is that large messages are an interesting challenge. The default message size is about 1 MB. With larger messages, you start to see memory pressure increase. In other words, the less page messages you can store in page cache is one main 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 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 some of the creators of Kafka and is actively influencing 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 open source nature of the tool 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 good reference if further details are needed rather than list them all in these chapters.

1.6 Summary

This chapter was about getting familiar with Kafka and its use cases at a very high level. In the next chapter, we will look at more details in which we use specific terms and start to get to know Apache Kafka in a more tangible and hands-on way.

In this chapter we:

- Discussed the streaming and fault-tolerance of Kafka that makes it a modern distributed solution
- Discovered that Kafka is not exclusive to Hadoop or HDFS. While it uses some of the same fault-tolerance and distributed systems techniques, it can stand alone as its own cluster.
- Talked about special features that make Kafka standout from other message brokers. This includes its replayable messages and multiple consumers features.
- Displayed specific use cases where Kafka has been used in production. These uses include messaging, website activity tracking, log aggregation, stream processing, and IoT data processing.

2 Getting to know Kafka

This chapters covers:

- Sending and receiving your first Kafka message from the command line.
- Reviewing the high-level architecture of Kafka.
- Understanding client options and how your applications can communicate to the system.
- Creating messages with the Java client.

Now that we have a very high-level view of why we would try out Kafka, this is the point at which we want to start and get more concrete with our terms and understanding, and look at the components that make up the whole of this system.

This chapter will also involve us setting up Kafka for the rest of our examples. While a distributed system, Kafka can be installed and run on a single machine. This will give us a starting point to dive into our hands-on use. As is often the case, the real questions start to appear 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. If you do not have Kafka set up already, please refer to Appendix B to help you get started so you can get Kafka running on your computer. Let's get started with Kafka and then spend a little more time digging into the details of Kafka's architecture.

2.1 Kafka's Hello World: Sending and retrieving our first message

Let's start using Kafka! Hopefully by now you have updated the configuration and started the 3 brokers we will be using in our examples and confirmed that they are still running (check out Appendix B if you have not set it up yet). With our broker setup ready, one major question now becomes: What do we really need to send when we talk to Kafka? For our first example, we are going to be creating a topic and sending our first message to Kafka from the command line. Messages are how your data is represented in Kafka. You will also see references to messages as records as those terms are often used interchangeably. Each message consists of a Key and a Value. A key is not required and we are just going to be sending a value in our first example. Figure 2.1 shows the general format of messages that users will be dealing with directly.

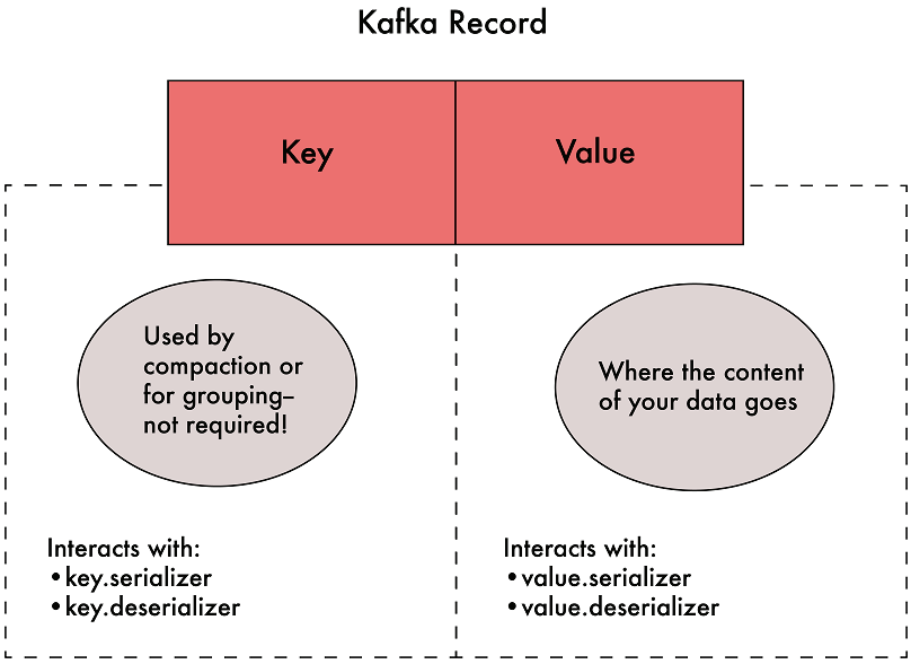

Figure 2.1 Kafka messages are made up a key and value

One thing that you should note is that Kafka was built with the command line in mind. There really is not a GUI that we will be using, so that is why it is important for us to have a way to interact with the operating system's command line interface. The following commands will be entered in a text-based prompt. Whether you use vi, emacs, nano, or whatever: just 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 help your usage of this product.

TIP **Shell Helper**

If you are a command line user and want a shortcut to auto-complete some of the commands (and help with the arguments available): Check out a Kafka auto-complete project at github.com/DylanLWScott/kafka_tools_completion.

In order to send our first message, we will need a place to send it to. We will create a topic by running the kafka-topics.sh command in a shell window with the `--create` option. This script is provided in your install of Kafka and delegates to Scala classes. Please note that any 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`.

Listing 2.1 Creating the helloworld Topic

```
bin/kafka-topics.sh --zookeeper localhost:2181 \
    --create --topic helloworld --partitions 3 --replication-factor 3
```

You should see the ouput: `Created topic "helloworld".`

In this example, we are going to use the name 'helloworld' for our topic. You can use your own name of course, consisting of upper and lower case characters, numbers, period, underscore, and dash (-). My personal preference is to follow general naming conventions that you would follow when using Unix/Linux. You will probably avoid a lot of frustrating errors and warnings if you do not have spaces or various special characters that do not always play nice with what you really want to do.

A couple of the other options are not as clear about their meaning at first glance. The `partitions` option is used to tell how many parts you want the topic to be split into. For example, since we have three brokers, using three partitions will give us a partition per broker. For our test workloads, we might not need this many. However, doing more than one at this stage will let us take a peek at how the system works and can spread data across partitions. The `replication-factor` also is set to three in this example. In essence, it says that for each partition, we are attempting to have three replicas of each one and will be a key part of your design to improve reliability and fault-tolerance. The `zookeeper` option is also pointed to our local instance of zookeeper which should have been running before you started your brokers. For our work right now, the most important goal is to get a picture of the layout as we dig into the details of how to make a best estimate at the numbers you will need in your other use cases when we get into broker details.

At this point, you can also look at any existing topics you have created and make sure that your new one is in 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.

Listing 2.2 Verify the Topic

```
bin/kafka-topics.sh --zookeeper localhost:2181 --list
```

To get a feel for how our fresh topic looks, let's run another command that will give us a little more insight into our cluster. You can see that when we describe the topic, we get more details from the broker. You really can see that it is not like a traditional single topic in other systems. The numbers you see under by the leader, replicas, and isr fields are the `broker.id` that we set in our configuration files. Briefly looking at the output, for example, we can see that our broker id of 0 is the leader for partition 0. Partition 0 also has replicas that exist on brokers 1 and 2. The last column "isr" stands for in-sync replicas. Having a partition copy that is out of date or behind the leader is an issue that we will cover later but is an important point to remember that replicas in a distributed system can be something that we will want to keep an eye out for.

Listing 2.3 View topic layout

```
bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic helloworld      (1)

Topic:helloworld    PartitionCount:3   ReplicationFactor:3  Configs:               (2)
    Topic: helloworld   Partition: 0   (3) Leader: 0   (4) Replicas: 0,1,2   (5) Isr: 0,1,2   (6)
    Topic: helloworld   Partition: 1      Leader: 1      Replicas: 1,2,0      Isr: 1,2,0
    Topic: helloworld   Partition: 2      Leader: 2      Replicas: 2,0,1      Isr: 2,0,1
```

(1) The describe option lets us look at the details of the topic we pass in

(2) The shows a quick summary of the total count of partitions and replicas that this topic has

(3) Each partition in tht topic is shown. This is for partition labeled 0

(4) This shows the leader for the specific partition (0 in this case). Leader 0 means the message broker with id 0

(5) Replicas tells us that the copies of this partition reside on the message brokers with those ids

(6) Insync replicas (ISR) shows the message brokers that are caught up with the leader and are not lagging behind

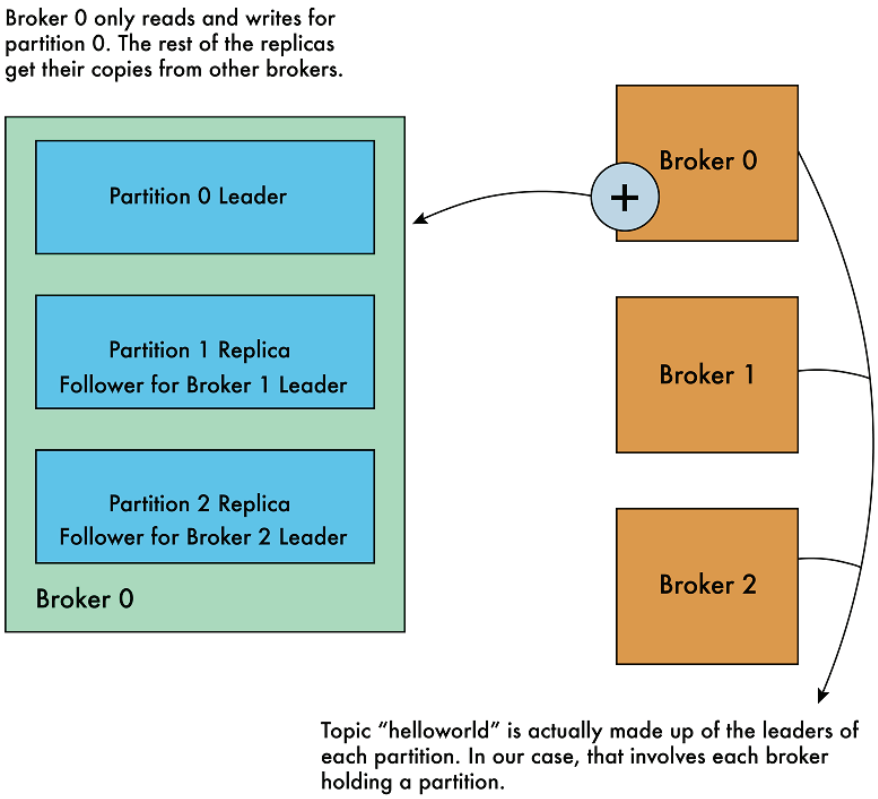

Figure 2.2 View of one broker

Now that we have our topic created, we can start sending real messages! Those who might have worked with Kafka before might ask why we took the above step to create the topic before we sent a message. There is configuration to enable or disable auto-creation of topics. However, I prefer to control the creation of topics as a specific action and do not want any new topics to randomly show up if I mistype a topic name once or twice or be recreated due to producer retires.

To send a message, we are going to start-up a terminal tab or window to run a producer that will run as a console application and take user-input. 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: helloworld.

Listing 2.4 Kafka Producer Command

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic helloworld
```

Now, we are going to start-up a new terminal tab or window to run a consumer that will also run as a console application. The below command will start a program that will take over that shell as well. On this end, you will hopefully see the message you had written in the producer console! Note to make sure that you are using the same `topic` parameter for both commands if you do not see anything.

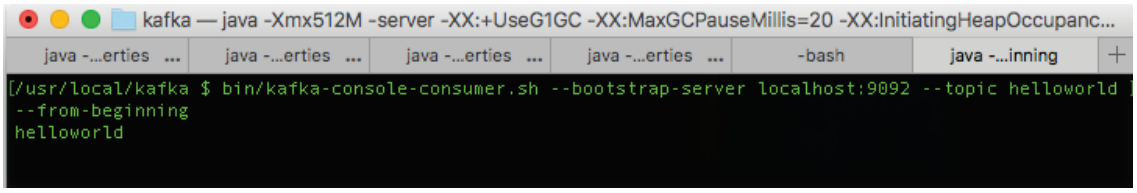

Figure 2.3 Example Consumer Output

Listing 2.5 Kafka Consumer Command

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

As you send more messages and confirm the delivery to the consumer application, feel free to terminate the process and leave off the `--from-beginning` option when you restart it. Did you notice that you didn't see every message you have ever sent before? 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.

One thing that some readers might have noticed from the exercises above is that the producer and consumer console scripts used a parameter called `bootstrap-server`. Contacting a message broker provides enough information to get the job done for these commands that send and retrieve messages. However, if we look at the kafka-topics command, we see that a `zookeeper` parameter is needed. The reliance on zookeeper has been lessened with later client versions, but ZooKeeper still provides value that we will touch on a little later in this chapter.

Now that we have seen a simple example in action, we have a little more context in order to discuss the parts we utilized above.

2.2 A quick tour of Kafka

Table 2.1 The major components and their roles within the Kafka architecture

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

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

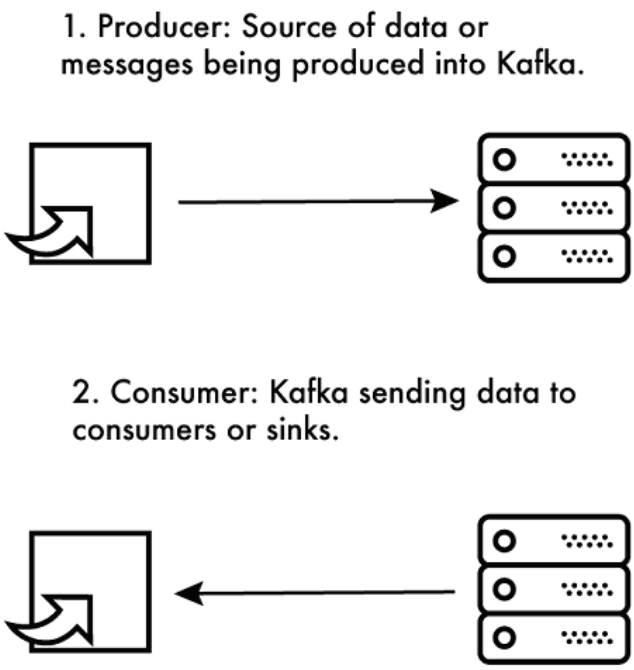

Figure 2.4 Producers vs Consumer

A Producer is the way to send 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 them in their own implementation code. Some of the entry ways might include using a separate tool such as Flume or even other Kafka APIs such as Connect and Streams. `WorkerSourceTask`, inside the Kafka Connect source code, is one example where a producer is used, but inside the API that the end-user uses. A producer is also a way to send messages inside Kafka itself. For example, if you are reading data from a specific topic and wanted to send it to a different topic, you 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 used in `WorkerSourceTask` that was a Java class mentioned earlier. Not all of the source code is listed for the method `sendRecords()`, but what is left is the logic dealing with sending a message with the standard KafkaProducer. It is not important to understand each part of the following example, but just try to get familiar with the usage of the producer.

Listing 2.6 WorkerSourceTask sending messages

```
private boolean sendRecords() {
...
final ProducerRecord<byte[], byte[]> producerRecord = new ProducerRecord<>(record.topic(),
    record.kafkaPartition(),                                                    ❶

...
```

```
try {
  final String topic = producerRecord.topic();

  producer.send(producerRecord,                    (2)
                                                   (3)
    new Callback() {
        @Override
        public void onCompletion(RecordMetadata recordMetadata, Exception e) {
          if (e != null) {
             log.error("{} failed to send record to {}: {}", this, topic, e);
             log.debug("{} Failed record: {}", this, preTransformRecord);
          } else {
            log.trace("{} Wrote record successfully: topic {} partition {} offset {}",
            this,
           recordMetadata.topic(), recordMetadata.partition(),
           recordMetadata.offset());
           commitTaskRecord(preTransformRecord);
          }
          recordSent(producerRecord);
          counter.completeRecord();
         }
    });
    lastSendFailed = false;
                                                   (4)
} catch (RetriableException e) {
  log.warn("{} Failed to send {}, backing off before retrying:", this, producerRecord, e);
  toSend = toSend.subList(processed, toSend.size());
  lastSendFailed = true;
  counter.retryRemaining();
  return false;
                                                   (4)
} catch (KafkaException e) {
 throw new ConnectException("Unrecoverable exception trying to send", e);
}
processed++;
}
toSend = null;
return true;
}
```

(1) The ProducerRecord will be what holds each message being sent into Kafka

(2) producer.send is what starts the actual call to send to our brokers

(3) A callback can be used for asynchronous sending of messages

(4) These are not thrown by the producer API but are good to know that Kafka often deals with errors and attempts to retry if possible

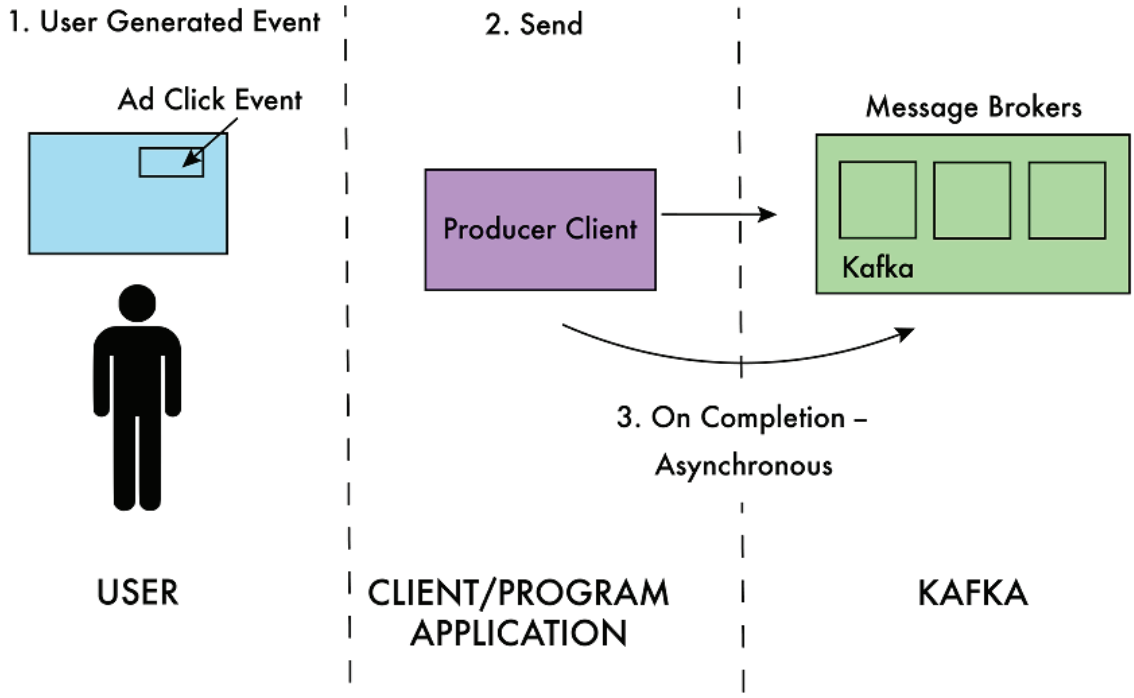

Figure 2.5 Producer Example

Figure 2.5 shows a user interaction that could start the process of data into a producer flow.

In contrast, a Consumer is the way to retrieve messages from Kafka. In the same vein as producers, if you are talking about getting data out of Kafka, you are looking at consumers being involved directly or somewhere in the process. `WorkerSinkTask` is one similarly named class inside Kafka Connect that shows the same usage of a consumer in Connect as a parallel with the similar producer example above. Consuming applications will 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.

Listing 2.7 WorkerSinkTask consuming messages

```
    protected void initializeAndStart() {
        String topicsStr = taskConfig.get(SinkTask.TOPICS_CONFIG);
        if (topicsStr == null || topicsStr.isEmpty())
            throw new ConnectException("Sink tasks require a list of topics.");
        String[] topics = topicsStr.split(",");

        consumer.subscribe(Arrays.asList(topics), new HandleRebalance());   (1)
...
    }

    protected void poll(long timeoutMs) {
...
        ConsumerRecords<byte[], byte[]> msgs = pollConsumer(timeoutMs);   (2)
        assert messageBatch.isEmpty() || msgs.isEmpty();
        log.trace("{} Polling returned {} messages", this, msgs.count());

        convertMessages(msgs);
        deliverMessages();
    }
```

1. The consumer needs to subscribe to the topics that it cares about
2. Messages are returned from a poll of data

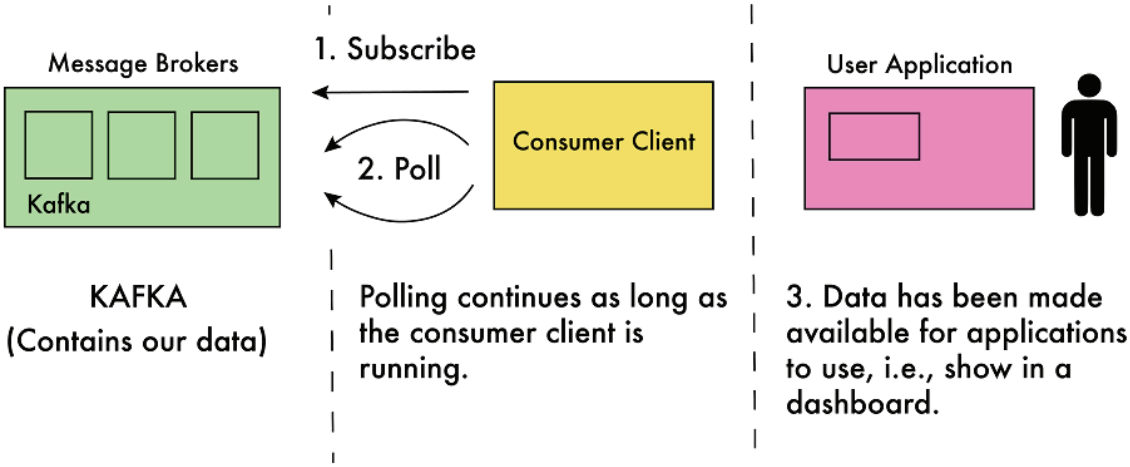

Figure 2.6 Consumer Example

These examples above can show two parts of a concrete 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 clicks and events generated by users would be the data going into the Kafka ecosystem. The consumers of the data would be the ad agency's itself that would be able to use its own applications to make sense of that data. Putting data into Kafka and out of Kafka with code like the above (or even with Connect itself) allows users to be able to work with the data that can impact their business requirements and goals.

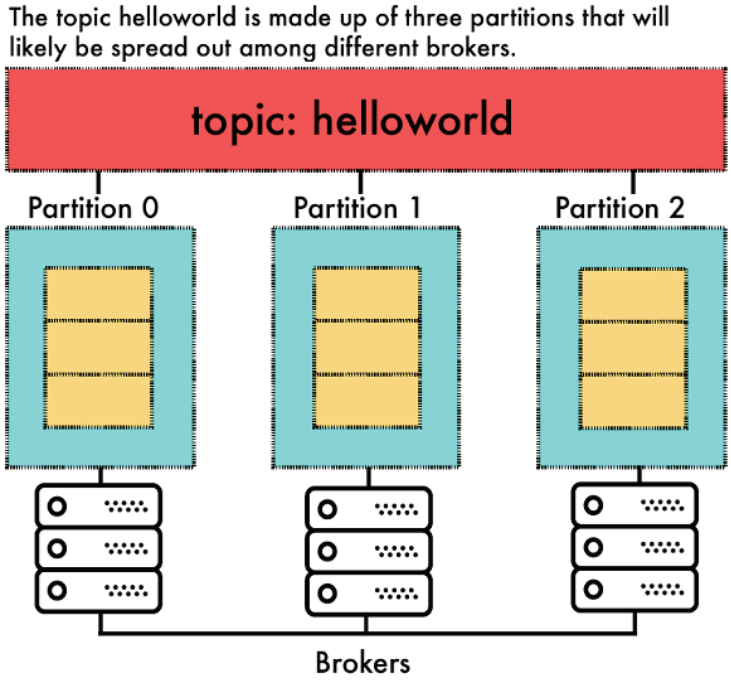

Figure 2.7 Segment files make up partitions. Partitions make up Topics.

Topics are where most users will start to think about the logic of what messages should go where. Topics are actually made out 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 only exists on one broker and will not be split between brokers. Figure 2.7 shows how each partition will exist on a single message broker and will not be divided smaller than that unit. Think back to our first example of the helloworld topic. If you are looking at reliability and want 3 copies of the data, the topic itself is not one entity (or single file) that is copied; rather it is the various partitions that are replicated 3 times each. As a side note, the partition is even further broken up into segment files (the actual) files written on the disk drive. We will cover the details of these files and their location when we talk about brokers in later chapters.

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 3 partitions and a replication factor of 3, each and every partitions will have a leader elected. That leader will be one of the copies of the partition and the others 2 (in this case) will be followers that update their information from their partition leader. Producers and consumers will only read or write (in happy path scenarios) from the leader of each partition it is assigned. But how does your producer or consumer know which partition is the leader? In the event of distributed computing and random failures, that answer is often influenced with help from ZooKeeper.

2.2.1 The what and why of ZooKeeper

One of the oldest complaints about the Kafka ecosystem might be that it uses ZooKeeper. Apache ZooKeeper (see the homepage at: zookeeper.apache.org/) is a distributed store which is used to provide configuration and synchronization services in a high available way. In more recent versions of Kafka, work was done in order for the client consumers to not store information about how far it had consumed messages (called offsets) into 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 however. While Kafka provides fault-tolerance and resilience, something is needed in order to provide the coordination needed and ZooKeeper enables that piece of the overall system. We will not cover the internals of ZooKeeper in detail, but will touch on how it is used by Kafka throughout the following chapters.

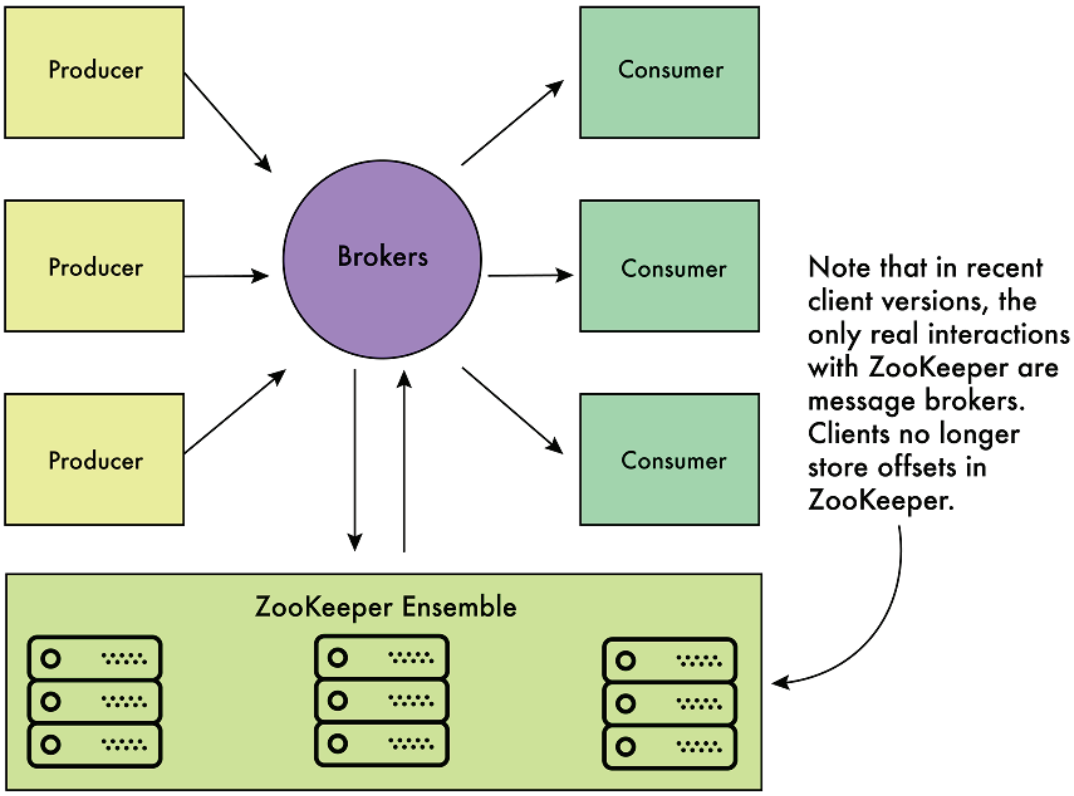

Figure 2.8 ZooKeeper Interaction

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 agreement. Agreeing on who the 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 true time if all of them are showing different times. 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) as well, but we will be running just one server in our local setup. Figure 2.8 shows the ZooKeeper clusters and how the interaction with Kafka 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 code-wise is `ZkUtils.scala`.

Knowing the fundamentals of the above concepts will really help us be able to add up these building blocks to make a practical application with Kafka. Also, you should be able to start seeing how existing systems that leverage Kafka are likely interacting to

complete real use cases.

2.2.2 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 allows this to be possible?

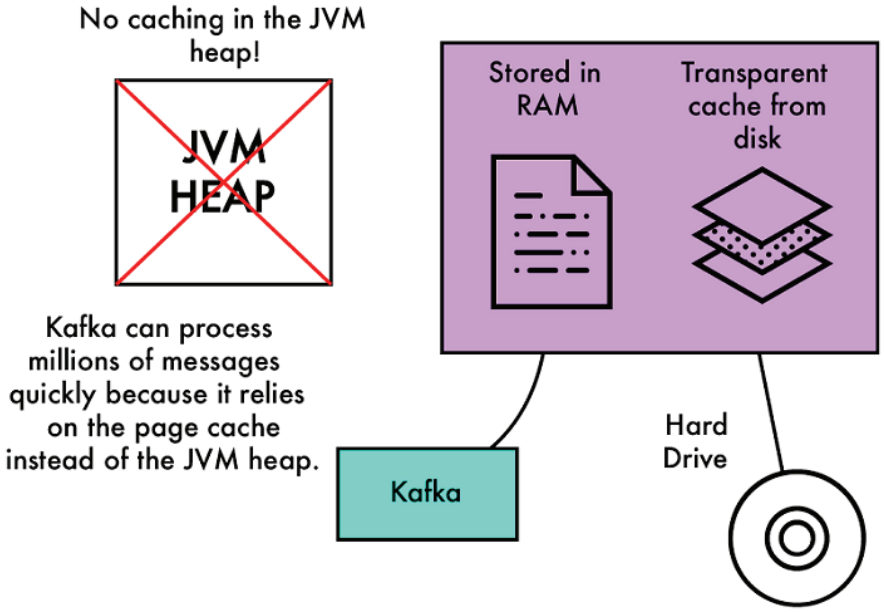

Figure 2.9 OS Pagecache

One of the keys for Kafka is its usage of the pagecache of the operating system. By avoiding caching in the JVM heap, the brokers can help avoid some of the issues that large heaps can hold, ie. 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 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 work fall into the pagecache.

As mentioned before, Kafka uses its own protocol (cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuide-PROD will shorten). 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.

2.2.3 The Commit Log

One of the core concepts to help you master the foundation of Kafka is to understand the commit log. The concept is simple but powerful when you get into 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.

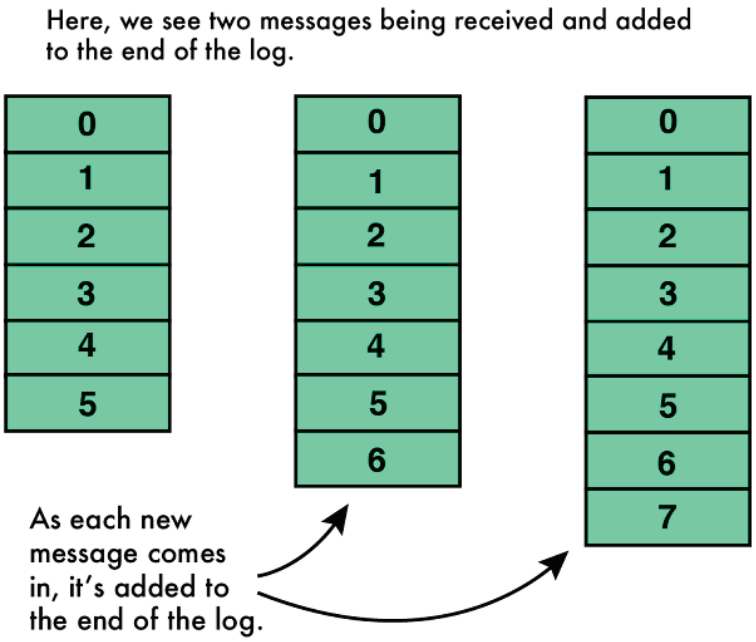

Figure 2.10 Commit Log

The figure above shows how simple the concept of a commit log can be. While there are more mechanics that actually take place, like what happens when a log file needs to come back from a broker failure, this concept really is a key 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 added to the end. In most traditional systems, linear read and writes usually perform better than random operations that would require spinning 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 data in the Kafka commit logs hits a configurable size or time retention period, the data is often moved into a permanent store like S3 or HDFS. However, it is really a matter of how much disk space you need as well as your processing workflow. The New York Times has a single partition that holds less than 100GB.[2] 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, but overall, logs are either retained by age or size configuration properties for those who wish to have some control over their Kafka stored data.

Footnote 2 www.confluent.io/blog/publishing-apache-kafka-new-york-times/

2.3 Various source code packages and what they do

Kafka is often mentioned in various titles or APIs. There are certain sub-packages that are often mentioned as stand-alone products. We are going to look at a some of those to see what options we have out there. The below packages are really APIs that are found in the same source code repository as Kafka core except for KSQL.

2.3.1 Kafka Stream Package

Kafka Streams has grabbed a lot of attention lately. This API is found in the core Kafka folder `streams` of the Kafka project and is mostly written in Java. One of the sweet spots for streams is that no separate processing cluster is needed. In fact, it is meant to really be a lightweight library to use in your application. For some of the smaller use cases, this means that you aren't required to have something like Apache YARN (Yet Another Resource Negotiator) in order to run your workloads. However, there are still very powerful features including local state with fault-tolerance, one-at-a-time message processing, and exactly once support. The more that you move throughout this book, you will be able to understand the foundations of how the streams API makes use of the existing core of Kafka to do some interesting and powerful work. This API was really made to make sure that creating streaming applications was as easy as possible and even provides 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. Microservice 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 microservice system. While the top part of the figure relies on each applicaiton talking directly to each other at mutliple 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.

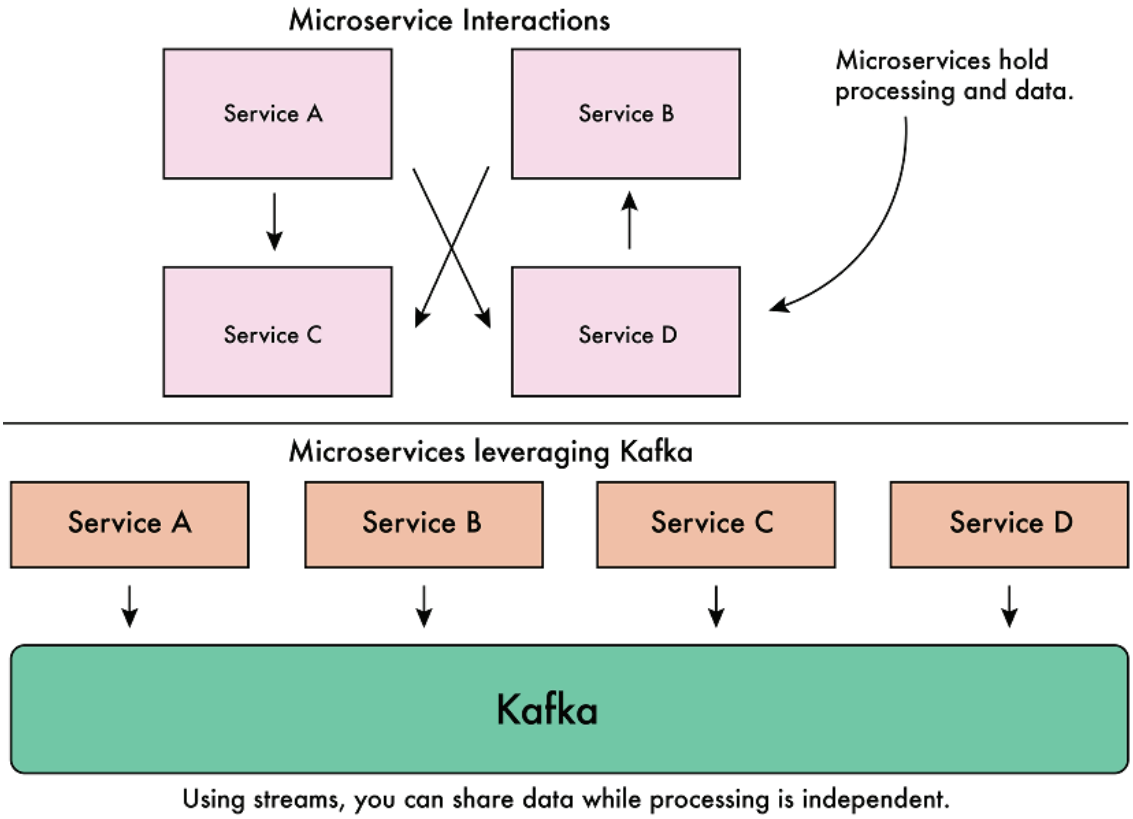

Figure 2.11 Microservice design

2.3.2 Connect Package

Kafka Connect is found in the core Kafka folder `connect` and is also mostly written in Java. This framework was created to make integration with other systems easier. In many ways, it can be thought to help replace other tools such as Camus, Apache Gobblin, and Apache Flume. If one is familiar with Flume, some of the terms used will likely seem familiar. Source connectors are used in order to import data from a source into Kafka. For example, if you wanted to move data from MySql tables to topics inside of Kafka, you would be using a connect source in order to produce those messages into Kafka. On the other hand, sink connectors are used to export data out of Kafka into a different system. Continuing the above example, if you wanted those topics to maintain data for longer term, you would use a sink connector in order to consumer those messages from the topic and place them somewhere else like HDFS or S3. As a note, using a direct comparison to Flume features are not the intention or sole goals of Connect. Kafka does not have an agent per node setup and is designed to integrate well with stream processing frameworks as well as copying data only. Connect is great for making quick and simple data pipeline that tie together a common systems.

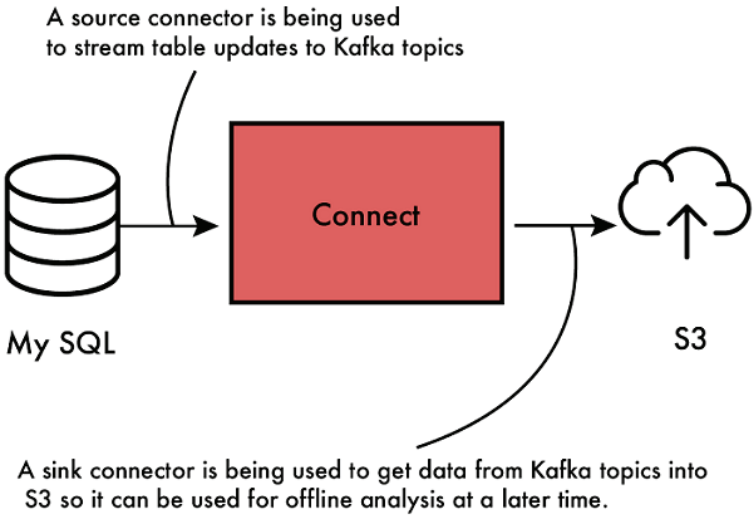

Figure 2.12 Connect Use Case

2.3.3 AdminClient Package

With version 0.11.0, Kafka introduced the AdminClient API. Before this API, scripts and other programs that wanted to perform certain administrative actions would either have to run shell scripts (which Kafka provides) or internal classes that were often used by those shell scripts. It is actually part of the kafka-clients.jar which is hosted in a different source repo than the other APIs. However, this interface is a great tool that will come in handy the more involved you become with the administration of Kafka. Also, this brings familiar usage with configuration being handled the same way it is for Producers and Consumers. The source code can be found in the following packages: `org/apache/kafka/clients/admin`.

2.3.4 KSQL

In late 2017, a developer preview was released by Confluent of a new SQL engine for Kafka. This allows for 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 important differences. Most queries that relational database users are familiar with involve on-demand or one-time queries that involve lookups. The mindset shift to a continuous query over a data stream is an important shift and new viewpoint for developers. As with the stream API, KSQL is really making the bar lower for entry into being able to leverage the power of continuous flows of data. While the interface for data engineers will be familiar with a 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 used point-in-time SELECT statements.

2.4 What sort of clients can I use for my own language of choice?

Due to the popularity of Kafka, the choice of which language to interact with Kafka usually isn't a problem. For the exercises and examples in this book, we will use the Java clients that are maintained as part of the Kafka project itself. There are many other languages (cwiki.apache.org/confluence/display/KAFKA/Clients -PROD will shorten) such as Python, golang, Ruby, and .NET, just to name a few, that are created by various companies and other open-souce contributors.

As with any code you plan to use in a production setting, you should make sure that you look into the quality of any independent clients and if they support the full feature set of the Kafka version protocol you want to use. If you truly want to peek under the hood of the Kafka protocol, 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 own client or even help you learn a new language. For example, for Erlang fans, there are a couple of clients that use bit syntax and pattern matching which can help you tangentially on other parts of your application development.

Since using a client is the most likely way you will interact with Kafka in your own applications, let's take a look at using the Java client. We will do the same process of producing and consuming a message as we did when using the command line earlier. You can run this code in a Java main method to produce one message.

Listing 2.8 Java Client Producer

```
Properties props = new Properties();                                          (1)
props.put("bootstrap.servers", "localhost:9092,localhost:9093");              (2)

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");   (3)
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);              (4)

ProducerRecord producerRecord = new ProducerRecord<String, String>
("helloworld", null, "hello world again!");                                   (5)

producer.send(producerRecord);                                                (6)

producer.close();                                                             (7)
```

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

(2) This property can take a list of message brokers.

(3) The key and value of the message have to be told what format they will be serializing.

(4) This creates a producer instance. Producers are thread-safe!

(5) This is what represents our message.

(6) Sending the record to the message broker!

❼ Cleaning up before we shut it all down.

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 very important config item and its purpose may not be clear at first glance. This list really is a list of your message brokers. A best practice is to include more than one server in order to make sure 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 brokers and will 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 be able to serialize the data as it moves into Kafka. Keys and values do not have to use the same serializer.

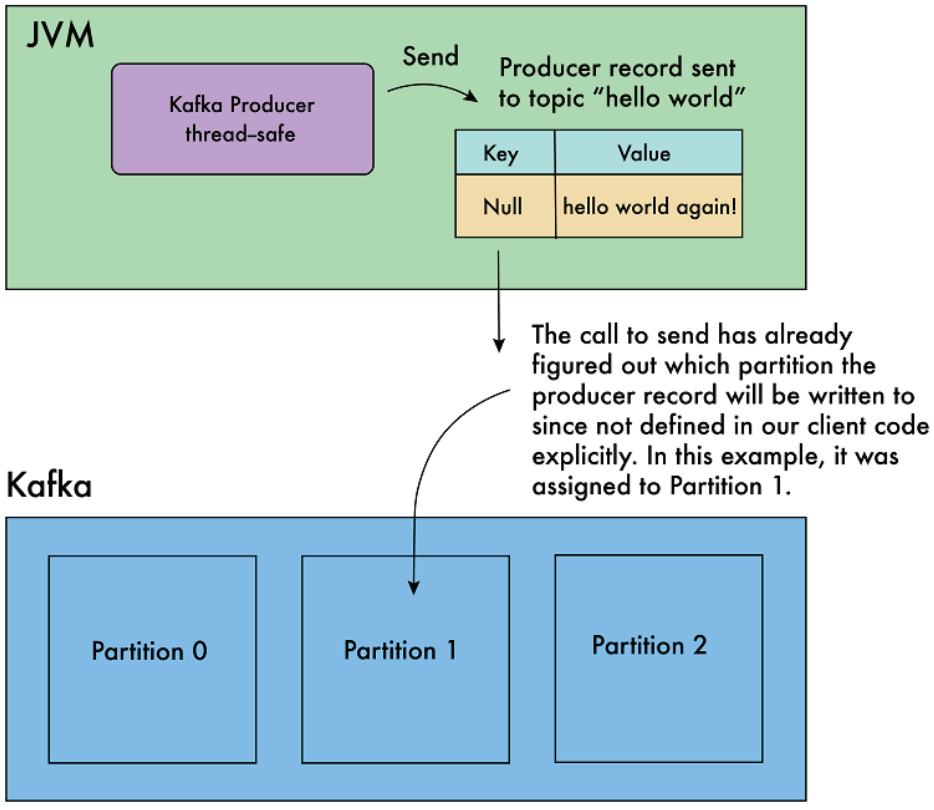

Figure 2.13 Producer Flow

Figure 2.13 helps display the flow of what is happening when a producer goes to send 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 our actual input that we wish to send. In our examples, "helloworld" is the name of the topic we wish to send the messages to. The next fields are the message key followed by the message value. We will get into keys more in Chapter 4, but for now it is enough to know that it can indeed be a null value.

Null values mean something different and we will cover that later as well. The message we send as the last argument is something different from our first message we sent with our first 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, as well as 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 so it can block until previously sent requests complete and shut down gracefully.

Before you try to run these Java client examples, make sure that you have the following in your pom.xml if you are using Maven like we will be in this book.

Listing 2.9 Java Client POM entry

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

Now that we have created a new message, let's use our Java client to create a consumer that is able to see that message. You can run the code inside a Java main method as well, and make sure you terminate the program after you done reading messages!

Listing 2.10 Java Client Consumer

```
Properties props = new Properties();                                    (1)
props.put("bootstrap.servers", "localhost:9092,localhost:9093");
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");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("helloworld"));                        (2)

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);       (3)
    for (ConsumerRecord<String, String> record : records)               (4)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(),
        record.value());
}
```

(1) Properties are set the same way as producers.

(2) The consumer needs to tell Kafka what topics it is interested in.

(3) An infinite loop! But 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 the fact that you really do have an infinite loop. It seems weird to do that on purpose, but you really 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 and you will need to make sure you take that into account as you scale past one consumer in later sections. Also, whereas we told the producer where to send the message, we now have the consumer subscribe to the topics it wants to retrieve messages from. The 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 your 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 you retrieve the above message and are done. As a note, these examples are relying on many configuration properties that are defaulted and we will have a chance to dig into more as we study each in later chapters.

2.5 Terminology of Kafka

As we start to look at how to use Kafka more and more, the most 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 own work use cases, we will be presented with the following terms and can hopefully use the descriptions below as a lens for which to view our processing mindset.

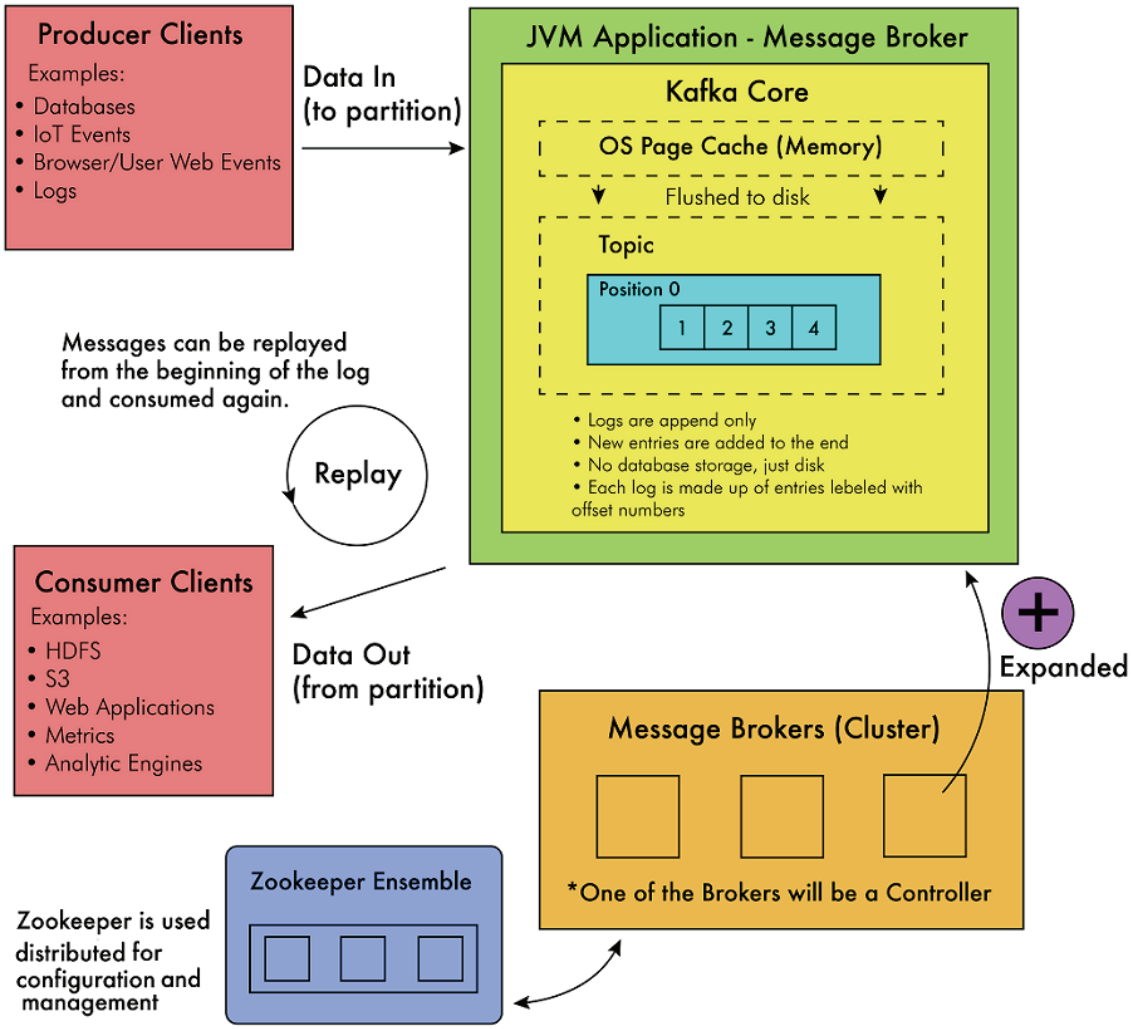

Figure 2.14 Kafka Context Overview

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 in order 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, then Consumers will help users utilize that data in their other applications and use cases.

2.5.1 What is a streaming process

Stream processing seems to have various definitions throughout various projects. The core principle of streaming data is that there is data that will keep arriving and will not end (unbounded). Also, your code process should be processing this data all of 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 not a part of this workflow. If you think of a river, the same principles apply. Sometimes there is a huge amount to transit and sometimes not that much, but it is constantly flowing between destinations.

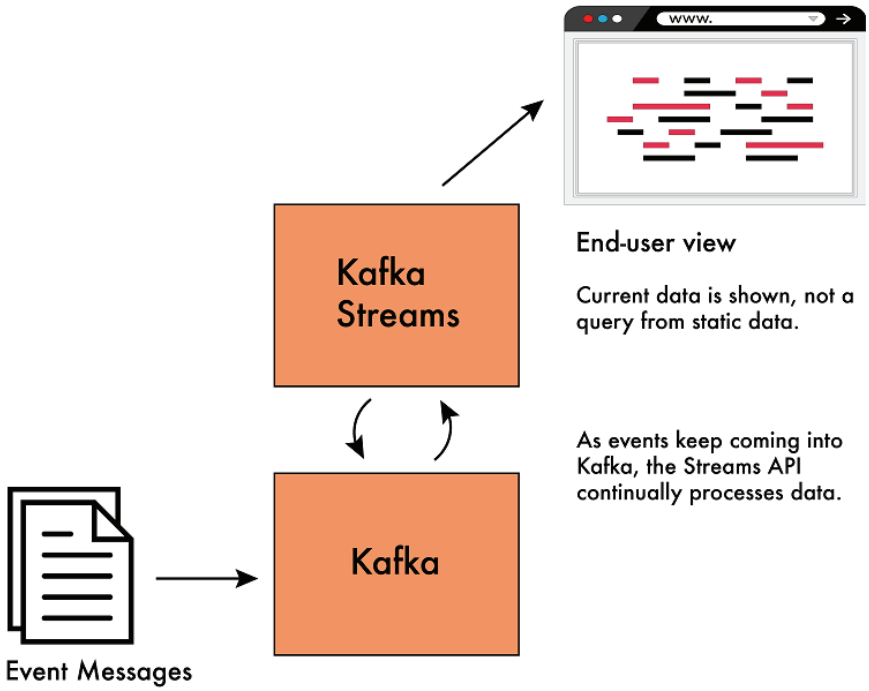

Figure 2.15 Stream Process

2.5.2 What exactly once means in our context

One of the most exciting and maybe most discussed feature was when Kafka debuted its exactly once semantics. Some in 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 your everyday usage of Kafka. Some important things to note: The easiest way to maintain exactly once is to stay within the walls (and topics) of Kafka. Having a closed system that can be completed as a transaction is why using the Streams API is one of the easiest cases. Various Connect connectors also support exactly once and are great examples for how to bring data out of Kafka since it won't always be the final endpoint for all data in each scenario.

2.6 Summary

This chapter was about learning more details about Kafka and its context in which we use specific terms. We also started to get to know Apache Kafka in a more tangible and hands-on way.

In this chapter we learned:

- How to produce our first Kafka message and consume it!
- How Kafka uses a commit log as its foundation and the features that are involved with that structure.
- What context to use as we discuss Kafka and distributed systems.
- How to produce a message using the Java client!

3 Designing a Kafka project

This chapters covers:

- Discuss how Kafka Connect can help you start a data streaming journey.
- Designing a real-world Kafka project.
- Which data format to use for our project.
- Existing issues (like data silos) that impact usage of data.
- When data transformation should take place.

In our previous chapter, we started to see how you can work with Kafka from the command line and from a Java client. Now, we are going to expand on those first concepts and look at designing various solutions with Kafka. We will discuss some questions to look at while you start to design a strategy for our example project we introduce below. 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 solve some real-world use case scenarios while having produced a design to start our further exploration of Kafka in the rest of this book. Let's start on our new 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 5 years old is probably considered legacy these days), that is not a luxury we always have in reality. One benefit of dealing with existing architectures, however, is that it gives us a list of pain points that we want to address. The contrast will also help us highlight the shift in thinking about data in our work.

To start focusing our brainstorming on the applications we could apply with this new hammer we just heard about, we are going to be working on a project for a company that

is ready to make a shift from their current way of doing data.

3.1.1 Taking over an existing data architecture

Let's look at some background to give us context. Our new consulting company has just gotten a contract to help re-architect a manufacturing plant. As with most projects, intentions were good and modern technology was built throughout the assembly line. Sensors were placed throughout the line as well and continuously provide events about the health and status of the equipment they are monitoring. So many events in fact, that the most of the messages are ignored by the current system. In an effort to make use of the data they already are already producing, we have been asked to help them unlock the potential in that data for their various applications to utilize. In addition, our current data context includes traditional relational database systems that are large and clustered. With so many sensors and an existing database, how could we create our new Kafka-based architecture without impacting the manufacturing itself?

3.1.2 Kafka Connect

One of best ways to start your journey is probably not a big-bang approach: all your data does not have to move into Kafka at once. If you 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. In fact, 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 famailiar with Connect.

3.1.3 Connect Features

The purpose of Connect is to help move data in or out of Kafka without having to deal with writing our own producers and clients. Connect is a framework that is already part of Kafka that really can make it simple to use pieces that have been already been built to start your streaming journey. These pieces are called connectors and they have been developed to work in a reliable way with other datasources. If you remember from Chapter 2, some of the producer and consumer Java client code in fact were used as examples that 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 could take a normal logfile and move that data into a Kafka topic. The easiest option to run and test Connect on your local machine is to run it in standalone mode.

In your folder where you installed Kafka, you should be able to locate the following files: `connect-standalone.properties` and `connect-file-source.properties`. 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 had used in making your own Java clients. Having knowledge of the underlying

producers and consumer clients help you understand how Connect is using that same configuration to complete its work by listing items such as `bootstrap.servers` and `key.converter`.

Also of interest is the fact that we are using the connect-file-source properties file. Since we are taking data out of one datasource and into Kafka, we will treat data as being sourced from that file. Using the file included with your Kafka installion, let's look inside at the contents of that file.

Listing 3.1 Starting Connect for a file source: connect-file-source.properties

```
name=local-file-source
connector.class=FileStreamSource    (1)
tasks.max=1                         (2)
file=test.txt                       (3)
topic=connect-test                  (4)
```

(1) This is the pre-built class that we are leveraging to do the work of interacting with our file source.

(2) For standalone mode, 1 is a valid value to test our setup. Scaling can come later if we like what we can do in standalone mode!

(3) This is our file that will be monitored for changes. Any messages you write to that file should be copied into Kafka.

(4) The topic property is the topic name of where this data will end up.

The value of the topic property is pretty important to note and we will be using it later to verify that messages are actually being pulled from a file into that specific `connect-test` topic. The file name of test.txt is also important to note. That is the file that will be monitored for changes for new messages.

To start Connect, in your terminal 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 config/
connect-file-source.properties
```

As a side note, if you were running ZooKeeper and Kafka locally, make sure that you have your own Kafka brokers still running as part of this exercise in case you had shut them down after previous chapters! Moving over to another terminal tab or window, we now should be able to add some text to the file `test.txt`. Use your favorite text editor or even something like: `echo -e "test 1\ntest 2" > test.txt` to add a couple of lines to the file and save.

We can quickly verify that Connect is doing its job by looking into the topic that was created with the console consumer command.

Listing 3.3 Confirming file messages made it to Kafka

```
bin\kafka-console-consumer
--bootstrap-server localhost:9092
--from-beginning --topic connect-test --new-consumer
```

Before we move on to looking at another connector type, I want to quickly touch on the sink connector to show how we can do the reverse quickly for a file as well: ie. move data from a Kafka topic to a file. Since the destination (or sink) for this data will be to a different file, we are interesting in looking at the file `connect-file-sink.properties` and the properties being set.

Listing 3.4 Starting Connect for a file source

```
name=local-file-sink
connector.class=FileStreamSink    (1)
tasks.max=1                       (2)
file=test.sink.txt                (3)
topics=connect-test               (4)
```

(1) This is the pre-built class that we are leveraging to do the work of interacting with our file source.

(2) For standalone mode, 1 is a valid value to test our setup. Scaling can come later if we like what we can do in standalone mode!

(3) This is our file that will be the destination for any messages that make it to our Kafka topic.

(4) The topic property is the topic name of where this data will come from that populates our file.

If you still have the Connect instance running in a terminal, close that terminal or `Ctrl-C` and running the updated command to start it with the file-source and file-sink property files.

Listing 3.5 Starting Connect for a file source and sink

```
bin\connect-standalone  config/kafka/connect-standalone.properties config/
connect-file-source.properties config/connect-file-sink.properties
```

To confirm that Connect is using our new sink, open or `cat` the file we used in our configuation for the sink file, `test.sink.txt`, to verify that you see the messages that we had created in the first place on our topic making it from the topic to the filesystem.

3.1.4 When to use Connect vs Kafka Clients

Since we have introduced both client APIs and Connect to get data into and out of Kafka, a common question is when should I use one over the other.

One of the simpliest questions that one should ask is if you can modify the

application code of the systems from which you need data interaction. If you have a database table that you want to interact with, do you have a hook into the flow of updates that go to your tables? Or would you not be familiar enough to even attempt to interace with a third-party provider? Connect allows those that have the in-depth knowledge the ability to create custom connectors and share them with others in order to help those of us that may not be the experts in those systems. For example, if you want to update an index in Elasticsearch from an existing Kafka topic, but have never used Elasticsearch, using a pre-built sink connector takes the burden off of learning that new interface. If you have used a connector before, it should be relatively simple to integrate a different connector since Connect has standarized the interaction with other systems.

To start using Connect in our manufacturing example, we will look at using a pre-built 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 setup a database for our local examples. For ease of use, I am going to be using SQLite but the process will be similar for any database that will have a JDBC driver available. 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 README.md in the source code of Chapter 3 to get install instructions.

To create a database, you just run the following from your command lien: `sqlite3 kafkatest.db`. In this database, we will run the following to create the `invoices` table and insert some test data.

Listing 3.6 Creating the Employees Table

```
CREATE TABLE invoices(                                  (1)
                                                        (2)
   id INT PRIMARY KEY     NOT NULL,
   title           TEXT    NOT NULL,
   details        CHAR(50),
   billedamt         REAL,
   modified    DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL
);

INSERT INTO invoices (id,title,details,billedamt)
VALUES (1, 'book', 'Franz Kafka', 500.00 );             (3)
```

(1) This create a table called 'invoices' that we will use in our examples.

2. Having an id that increments will help Connect know which entires to capture.
3. This is an example to insert test data into our new table.

By starting Connect with the pre-built `quickstart-sqlite.properties` file, we can see how additional inserts and updates to the rows will cause messages into Kafka. Refer to the source code for Chapter 3 to see more detailed setup instructions for pulling the jdbc connector since it is not part of the Apache Kafka distribution like the file connector was.

Listing 3.7 Starting Connect for a database table source

```
bin/connect-standalone etc/schema-registry/connect-avro-standalone.properties etc/
kafka-connect-jdbc/source-quickstart-sqlite.properties (1)
```

1. Note that the schema-registry and sqlite files are from the Confluent install. Pleaes see the source code for Chapter 3 to setup these files if not already.

Connect also has a REST API and of course provides options to build your own connector. While the power of Connect is great for moving our exisitng database table to Kafka, our sensors are going to be another story.

3.2 Sensor Event Design

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

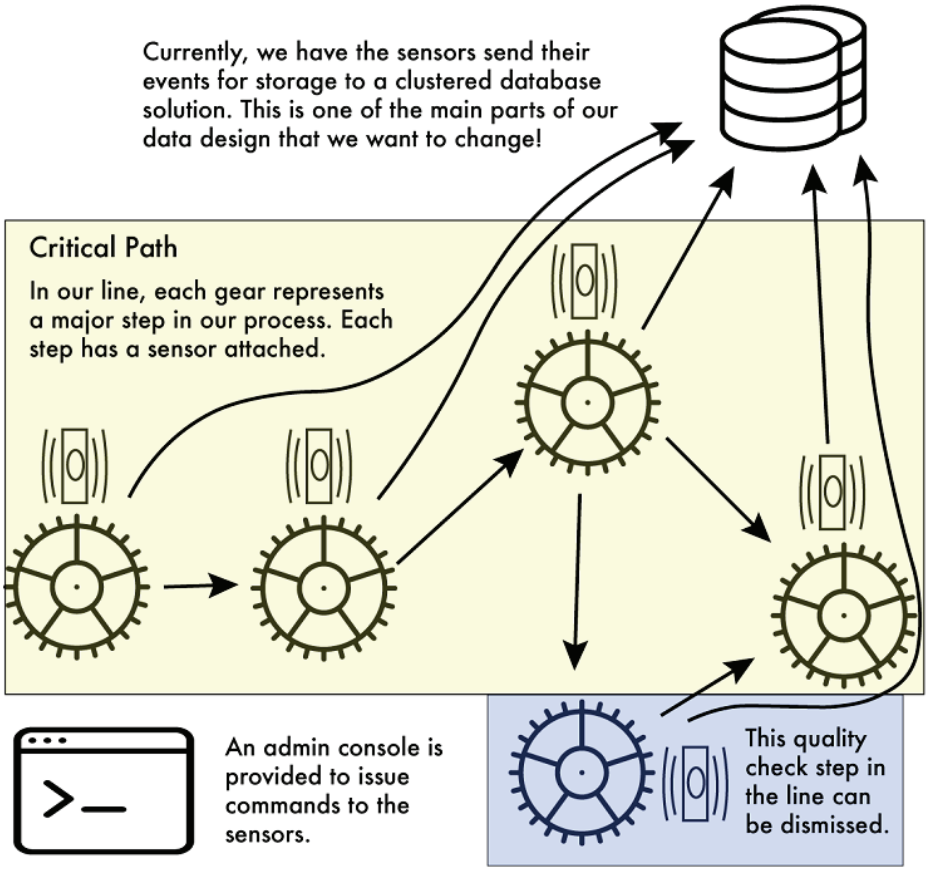

Figure 3.1 Factory Setup

Our diagram shows that we have a critical path of steps that need to be completed. There is an extra step that is an extra quality check stage that can be skipped if down for maintenance or failure to avoid processing delays. Sensors are attached to all of the steps of the line and send messages to the clustered database machines that exist in the current system. There is also an administration console that is used to update and run commands against that sensors that are built into the system already.

3.2.1 Existing issues

DEALING WITH DATA SILOS

Let's start with thinking about one of the issues that has been mentioned throughout most of our previous use cases. The need for data to not only exist, but also to be available to users is a hard problem. In most of my past work 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 was one you could process easily? Or what if they did not even 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 came in, you do not have to worry about the application API exposing it in their specific formats or even custom transformations that they applied. And what if the application providing the API parsed

the original data wrong? To untangle that mess might take a while if you have to recreate your own data off of changes to the data source you were using.

One of the nicest perks about a distributed system like Kafka is that failure is an expected condition that is planned for and handled. However, besides system blips, we still have the human element in coding applications. If an application had a defect or logic issue that destroyed your data, what would be your path to correct it? With Kafka, it can be as simple as starting to consume from a topic from the beginning as we looked at with the console consumer flag `--from-beginning` in Chapter 2. The retention of that data makes it available for use again and again. The ability to reprocess data for corrections is powerful. If storage of the original event was not available, it might be really hard to retro fix the existing data.

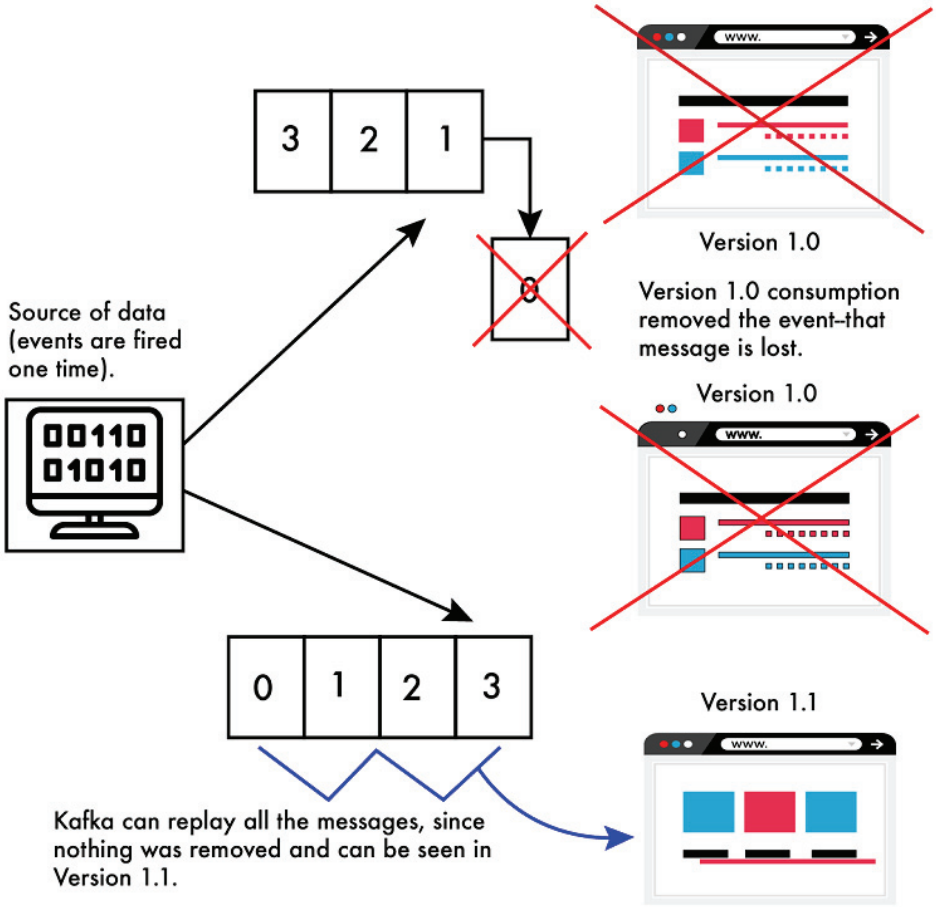

Figure 3.2 Developer Coding Mistake

Since events are only produced once from the sensor source for that specific instance, the message broker can play a key 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 the top of the figure 3.2, it is gone from the system. If a defect in an application's logic was found after the fact, there would need to be analysis on if the data could be altered from what was left over 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 it has already consumed with the new application version logic. The messages can be replayed and our new application code that fixed our logic mistake from verison 1.0 could process all the events again. The chance to process our events again can open-up the chance for our applications to get something right a second time without much hassle.

The replay of data can also show you how a value changed over time. Like a write ahead log (WAL), you can tell what a value used to be, the changes that happened over time, and what is it currently. If you followed the events from the beginning to the end, you would be able to 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 probably 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. Transformation is really something that can wait for a different stage in the process. Whether data is coming from a database or a log event, my preference is to get the data into Kafka first. The data will be available in the purest form that you can achieve. Each step before is a change that data could have been altered or injected with various formatting and programming logic errors.

Of course there will always be exceptions to the rules. One great reason to transform the data is if it is a key part of your records to have keys 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 always a great place to get inside of the Kafka ecosystem and have the ability to replay data if any of those failures occur. Otherwise, you are at the mercy of the tools where you were extracting data to attempt to gather that data again.

3.2.2 Why Kafka is a correct fit

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

One thing that has been made clear from our clients is that the current database is getting expensive to scale larger in a vertical way. By vertical scaling, we mean that improvements in power like CPU, RAM, and disk can be added to an existing machine. To scale dynamically, we would be looking at adding more servers to our environment. With the ability to horizontal scale out our cluster, we hope to get more overall benefit 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 be able to handle production loads. The commodity label gets thrown around a lot to discuss servers in some scaled distributed systems, but modest servers, as noted, will probably be what you would rely on in reality.

The other item that probably jumped out at you is the fact that we have events being produced continuously. It hopefully sounds 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 stop point. Our system should be ready to handle messages constantly.

Another interesting point to note is that our messages are usually under 10 kB. The small sizes of the messages will be under the default 1 MB size which would require some configuration changes. The smaller the message size and the more memory we can offer to pagecache should put us in good shape to keep our performance healthy.

One of our security minded developers noticed that there wasn't 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 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 are going to use a recent version (at this time of this writing 1.0.0) for our examples, some developers might not have control over the current brokers 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 Kafka Version History

Kafka Version	Feature
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

One thing to note since we are focused on clients in this chapter is the feature: improved client compatibility. Broker versions 0.10.0.0 and 0.10.1.0 can be run with newer client versions. The true importance is that if you want to try new versions of the client, you should be able to upgrade your client first. The brokers can remain at their version until you decide you want to upgrade them after the clients, if at all.

Now that we have decided to give Kafka a try, it might be time to frame how we want our data to live. The questions below are intended to make us think about how we really want to process our data. These preferences will impact various parts of our implementation, but our main focus is on just figuring out the what. We will cover the how in later chapters. This list is not meant to be exhaustive, but is a good starting point to tackling the 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 if your social account RSS feed misses 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? In other words, one example could be if we are going to be taking in address changes. In that case, we might want to associate the various address changes with the customer who is doing the ordering. Grouping events upfront might also help applications avoid coordinating messages from multiple consumers while reading from the topic.

Do you need data delivered in an exact order? What if a message gets delivered in an order other than when it actually occurred? For example, if you get a deposit before a withdaw from an ATM comes in but it is received by your bank in the other order. That overdraft fee and customer service impact are probably good enough reasons to say that the ordering is indeed important in that case. Or course, not everything will need this order. If you are just looking at page count numbers, the order is not as important as making sure that you can get a total at the end (a combination).

Do you only want the last value of a specific value? Or is history of that value important? Do you really care about the history of how your data values evolved over time? 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 when 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 end of our processing.

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

3.2.4 User data requirements

Our new architecture will want 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, we want to make sure that they can process the events that were sent at a later time 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 parts of our line could lead to delays or stoppages.

Besides the information above, we also want to make sure we maintain a history of the health statuses of the sensors. This data will be used in order to determine if we can trend and predict failures from sensor data before actual events lead to line and work stoppages.

In addition, we want to keep an audit log of any users that pushed any updates or queries directly against the sensors. For compliance reasons, we want to make sure we 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 look 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 the 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. Order does not really matter inside of our topics, each message will have a timestamp in the data itself. Our main concern is that all the data is there to process.

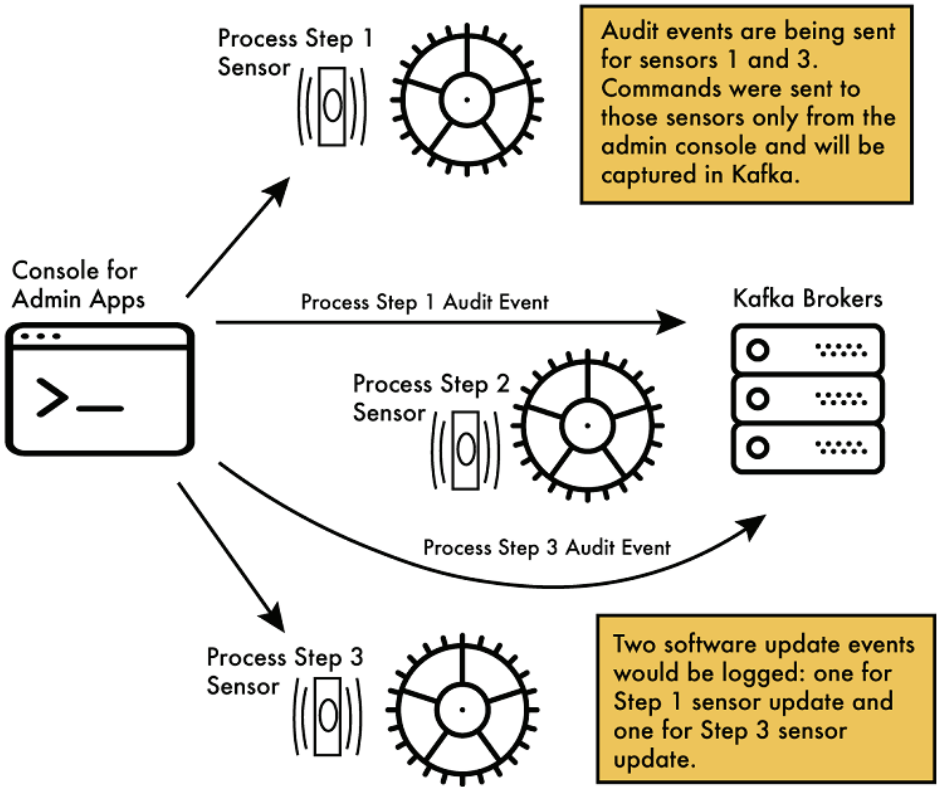

Figure 3.3 Audit Use Case

Table 3.2 Audit Checklist

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

To make things a little more clear from our discussion alone, I have created a rough checklist of things that I would be thinking of in regards to data for each requirement. This at a glance view will help us when we go to figure out the configuration options we want to use for our producer clients.

The health statuses requirement for each process of the assembly line for trending might be helpful to group by a key. Since we have not really addressed the term 'key' in

depth, it is imporant to think of it as a way to group events that are related. We will likely use the stage name of the assembly line 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 that have been produced to spot these trends over time per stage. By using the same key for each sensor, we should be able to easily consume these events. Since health 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 equiqment making the line work, not the sensor itself. If a sensor sends a 'Needs Maintenance' message every couple of days, that is the type of information we want to have in order to spot trends in equiqment failing.

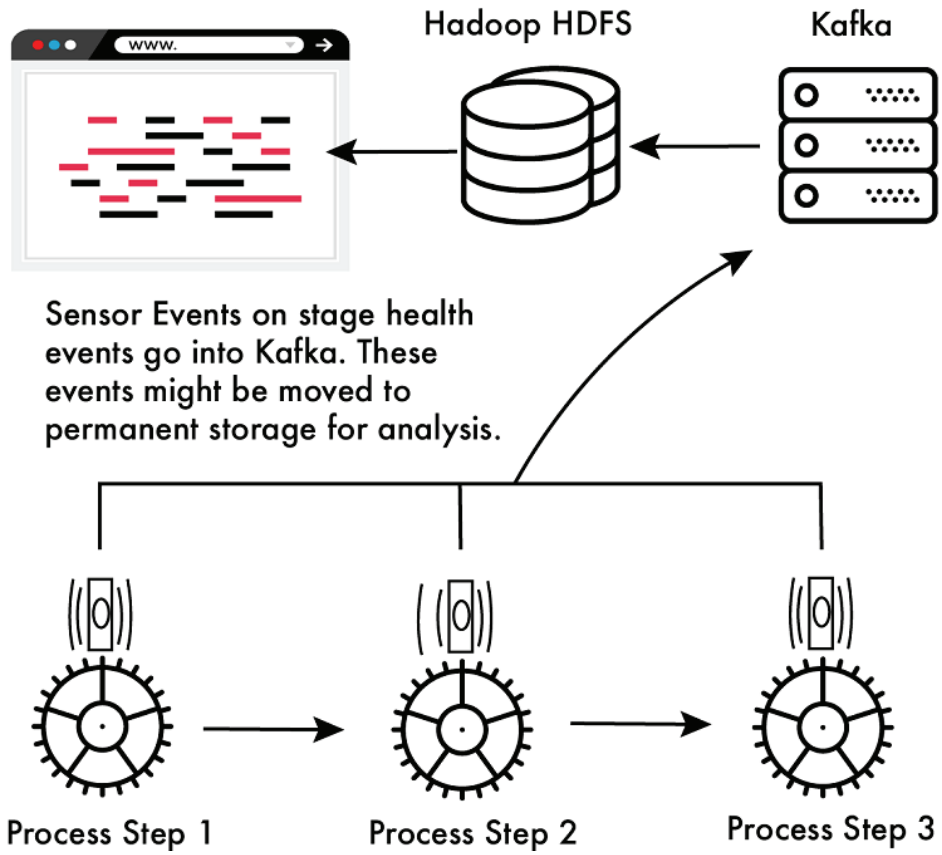

Figure 3.4 Health Trending Use Case

Table 3.3 Health 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 proces step as well. However, we do not care about past states of the sensor, but rather the current status. In other words, the last status is all we care about and need for our requirement. 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 probably use to thinking of). Internally, Kafka will add the new event that it recieves to the end of its logfile like any other message it receives. After all the log is immutable and can only appended to the end. So how does Kafka make what appears to be an update happen? We will dig into this process called log compaction in the next chapter. Another difference we are going to 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).

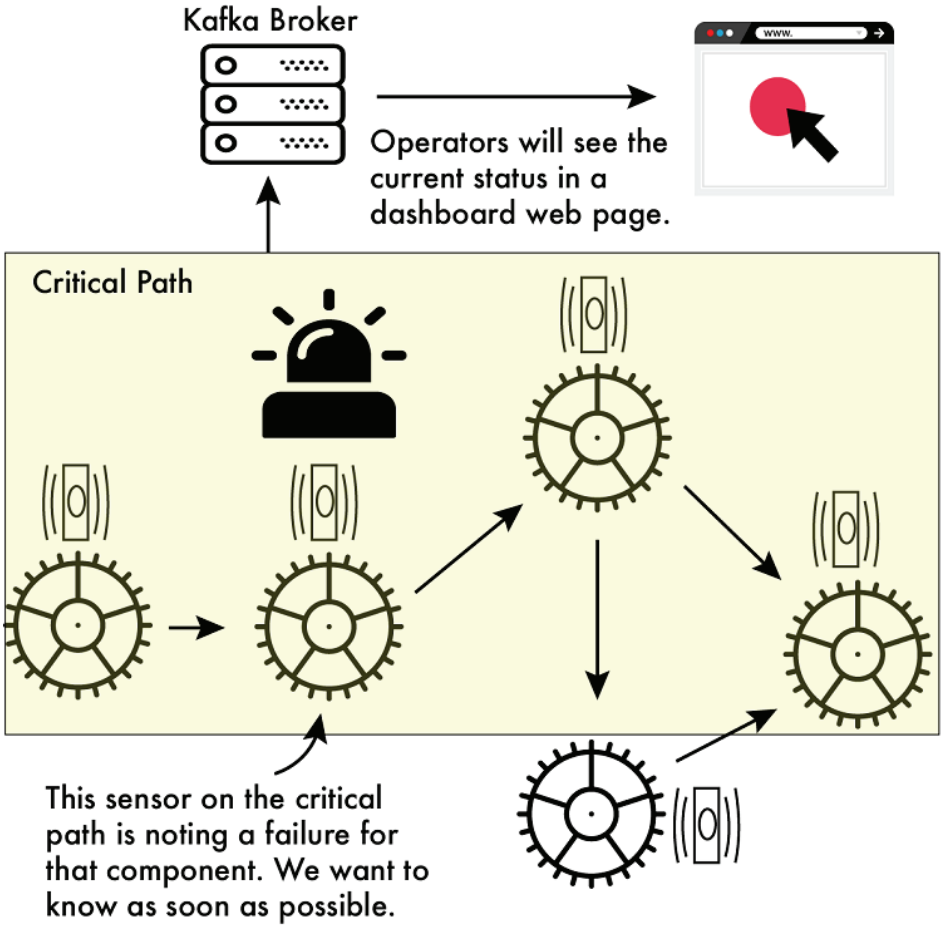

Figure 3.5 Alert Use Case

Table 3.4 Health Alert Checklist

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

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
- health data
- alert data

For those of you already jumping ahead, let's keep in mind that we might be able to use our health data as a starting point for our alerts topic: ie. you can use one topic as the starting point to populate another topic. However, to start off in our design, we will write each event type from the sensors to their own 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 health trend events will end up on a health topic, and our alert events on a alert topic. This 1-to-1 mapping will make it easier to focus on the requirement at hand.

3.3 Data Format

One of the easiest things to skip but truly important topics to cover in your design involves figuring out what format you want all of your data to be in. XML, JSON or comma-separated values (CSV) are probably pretty common formats that help define some sort of structure to our data. However, even with clear syntax format, there can be information missing in your data. What is the meaning of the first column or third one? What is the data type of the field in column 2? The knowledge about how to parse your data or analyze is hidden in applicatons over and over again that pull that data form its storage location. Schemas are a way to provide some of this needed information in a way that your code can handle.

If you look at the Kafka documentation, you may have noticed references to another serialization system called Apache Avro that is able to provide schema support as well as storage of schemas in the avro file itself. Let's start to look at what purpose and why this format is common to see in use in Kafka.

3.3.1 Why Schemas

One of the major gains of using Kafka is that the producers and consumers are not tied directly to each other. Further, Kafka does not really care about the content of the data or do any validation by default. Kafka understands how to quickly move bytes and lets you have the freedom to put whatever data you need into the system. However, there likely still exists 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, you can give a way to define in your application about what is intended. The definition doesn't have to be posted in a README for others in your organization to determine data types or try to reverse-engineer from data dumps.

Listing 3.8 Avro schema example

```
{
    "type" : "record",                    ❶
    "name" : "contactDetails",
    "namespace" : "avro.example",
    "fields" : [{"name" : "fullname",
                 "type" : "string",
```

```
                 "default" : "NONE"},

                {"name" : "age",          (2)
                 "type" : "int",          (3)
                 "default" : -1},         (4)

                 {"name" : "cell_number",
                  "type" : "string",
                  "default" : "NONE"},

                  {"name" : "address",
                   "type" : {
                         "type" : "record",
                         "name" : "home_address",
                         "fields" : [
                            {"name" : "street",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "city",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "state_prov",
                             "type" : "string",
                             "default" : "NONE"},

                            {"name" : "zip",
                             "type" : "string",
                             "default" : "NONE"}
                          ]},
                          "default" : {}
                }
    ]
}
```

(1) The schema is defined as json

(2) Usually a mapping to a field name

(3) The schema can tell you that the age will be an 'int' in this case

(4) You can ever define default values if needed!

By looking at the example of an Avro schema, you can see that questions such as do we parse the zipcode as a number or a string (in this case string) is easily answered by a developer looking at the schema. Applications could also automatically use this information as well to generate data objects for this data and help avoid parsing data type errors.

If you are coming from a relational database background, you might say how is this different than using a table that has a schema defined? After all a schema is usually associated to one topic just like a schema is applied to one database table. However, as we will discuss below, schemas can be used by tools like Avro in order to handle data that evolves over time. Most of us have dealt with alter statements or tools like Liquibase to work around these changes. With schemas, we are starting out with the knowledge that our data is probably going to change upfront. Do you really need a schema when you are first starting out with your data designs? One of the main concerns is if the scale of your system 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 Why 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. In addition, 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. An interesting point is that Avro data is serialized with its schema. If the schema changes, the old and new schema are present when processing data, so differences can be handled.

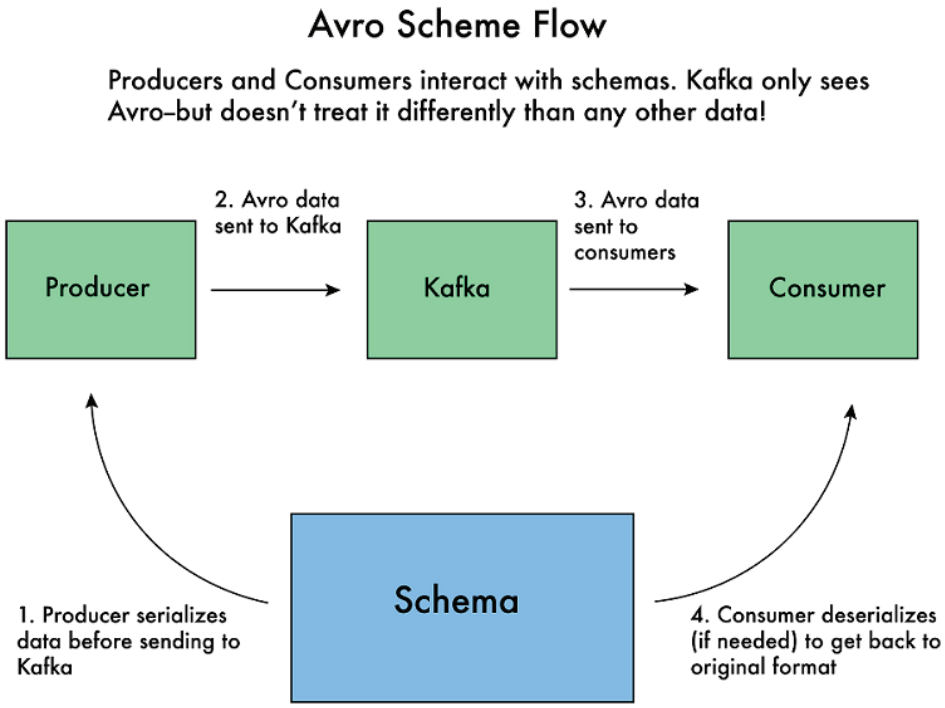

Figure 3.6 Avro Flow

Another benefit for looking at Avro is the popularity of its usage. I first saw its usage on various Hadoop efforts and can be used in many other applications as well. Bindings exist for many programming languages and should not be hard to find in general. For those that have past bad experiences and prefer to avoid generated code, it can be used in a dynamic manner.

Let's get started with how we are going to use Avro by adding it to our pom.xml. If you are not used to the pom.xml or Maven, you can find this file at the root of your project.

Listing 3.9 Add Avro to pom.xml

```
<dependency>
  <groupId>org.apache.avro</groupId>          (1)
  <artifactId>avro</artifactId>
  <version>1.8.2</version>
</dependency>
```

<1>This entry will be added to 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 an standalone java titles `avro-tools` if you do not want to use a Maven plugin. Avro does not require that code be generated.

Listing 3.10 Adding Avro Maven plugin to pom.xml

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>                                   (1)
  <version>1.8.2</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>                                          (2)
      <goals>
        <goal>schema</goal>                                                    (3)
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>   (4)
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>        (5)
      </configuration>
    </execution>
  </executions>
</plugin>
```

(1) The name of the artifact we need listed in our pom.xml as a plugin.

(2) The Maven phase configuration.

(3) The Maven goal configuration.

(4) The directory that will be scanned for avro files.

(5) The location where the generated Java code will be placed.

Let's start by defining our schema by thinking about the data types we are going to be using. For our scenario above about the alert status, let's try to define what fields and types we want to deal with. To start you just create a new file with your text editor of choice. I named my file the following: alert.avsc.

Listing 3.11 Alert Schema: alert.avsc

```
{"namespace": "com.kakfainaction",
 "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"}
 ]
}
```

1. Namespace will be the generated package you want the source code to be created in.
2. Alert will be the name of the Java class you will interact with.
3. The fields we want in our data alone with their data type and documentation notes.

One thing to note is that "doc" is not a required part of the definition. However, I do believe there is value in adding specific details that will help future producer or consumer developers understand what your data means in context. The hope is to stop others from inferring the meaning of your data and to 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 note not only tells us the data type, but also that epoch is used with the unit being seconds.

Now that we have the schema defined, let's run the maven build in order to see what we are working with. `mvn generate-sources` can be used to generate the sources in our project. If you used the same configuration above, you should see a couple of generated classes: `com.kafkainaction.Alert.java` and `com.kafkainaction.alert_status.java` that 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 Producers and Consumers in order to use this schema we have created.

While you are welcome to create your own serializer for Avro, we already have an excellent example that is provided by Confluent. Access to those existing classes is accomplished by adding the `kafka-avro-serializer` dependency in your build.

Listing 3.12 Adding Kafka serializer to pom.xml

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>3.3.1</version>
</dependency>
```

1

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

NOTE **Confluent Avro Maven Repository**

If you are using Maven to follow along: Make sure that you add the following to your pom

```
<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 last chapter `HelloWorldProducer` and slightly modify the class in order to use Avro.

Listing 3.13 Producer using Avro Serialization

```
...
props.put("key.serializer",
    "org.apache.kafka.common.serialization.LongSerializer");
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroSerializer");                    (1)
props.put("schema.registry.url", "http://localhost:8081");                (2)

Producer<Long, Alert> producer = new KafkaProducer<Long, Alert>(props);
Alert alert = new Alert();                                                (3)
alert.setSensorId(12345L);
alert.setTime(Calendar.getInstance().getTimeInMillis());
alert.setStatus(alert_status.Critical);
System.out.println(alert.toString());

ProducerRecord<Long, Alert> producerRecord =
new ProducerRecord<Long, Alert>("avrotest", alert.getSensorId(),
                alert);                                                   (4)

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

1. Now we are sending the Alert as a value and using the KafkaAvroSerializer.
2. We will cover the Schema Registry in Chapter 11! This property points to our url of the registry which has a versioned history of schemas and will help in regards to the evolution of schemas.
3. Creating a alert to show using the generated Java classes from our schema creation.
4. Notice the type of the generic is now Alert instead of a string from earlier examples.

The key differences are pretty minor in that our types change for the Producer and ProducerRecord as well as the configuration settings for the `value.serializer`. This example also makes use of the Schema Registry. We will cover more details of this feature in Chapter 11.

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

3.4 Summary

In this chapter we learned:

- How Kafka Connect can be leveraged with existing connectors to write to and from a file.
- How to compile a checklist of how features such as message lose and grouping need to be considered in producer design.
- How to leverage Avro schemas defintions to generate Java code.
- How we can leverage serializers like Avro in order to handle data changes in the future and use them in our own custom Kafka producer client.

4 Producers: Sourcing Data

This chapters covers:

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

In the previous chapter, we started to look at some of the requirements that an organization could have in regards to their data. Some of those discussions that were made have practical impacts in how we need to send our data into Kafka. To enter into the world of Kafka, the portal gate is through the producer. Please take note that when we are looking at how things work below in detail, I will try to lean towards the defaults. Since the producer can be in different programming languages (different client implementations), the details provided will be specific to the Java client that we used in our previous examples. While some clients (C/C++, Python, and, Go) will likely have similar configuration, it is not guaranteed. Defects and features can also be unique per client.

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

4.1 Introducing the Producer

There are not many ways around it (none recommended anyway): the producer will be the way to push data into the Kafka system.

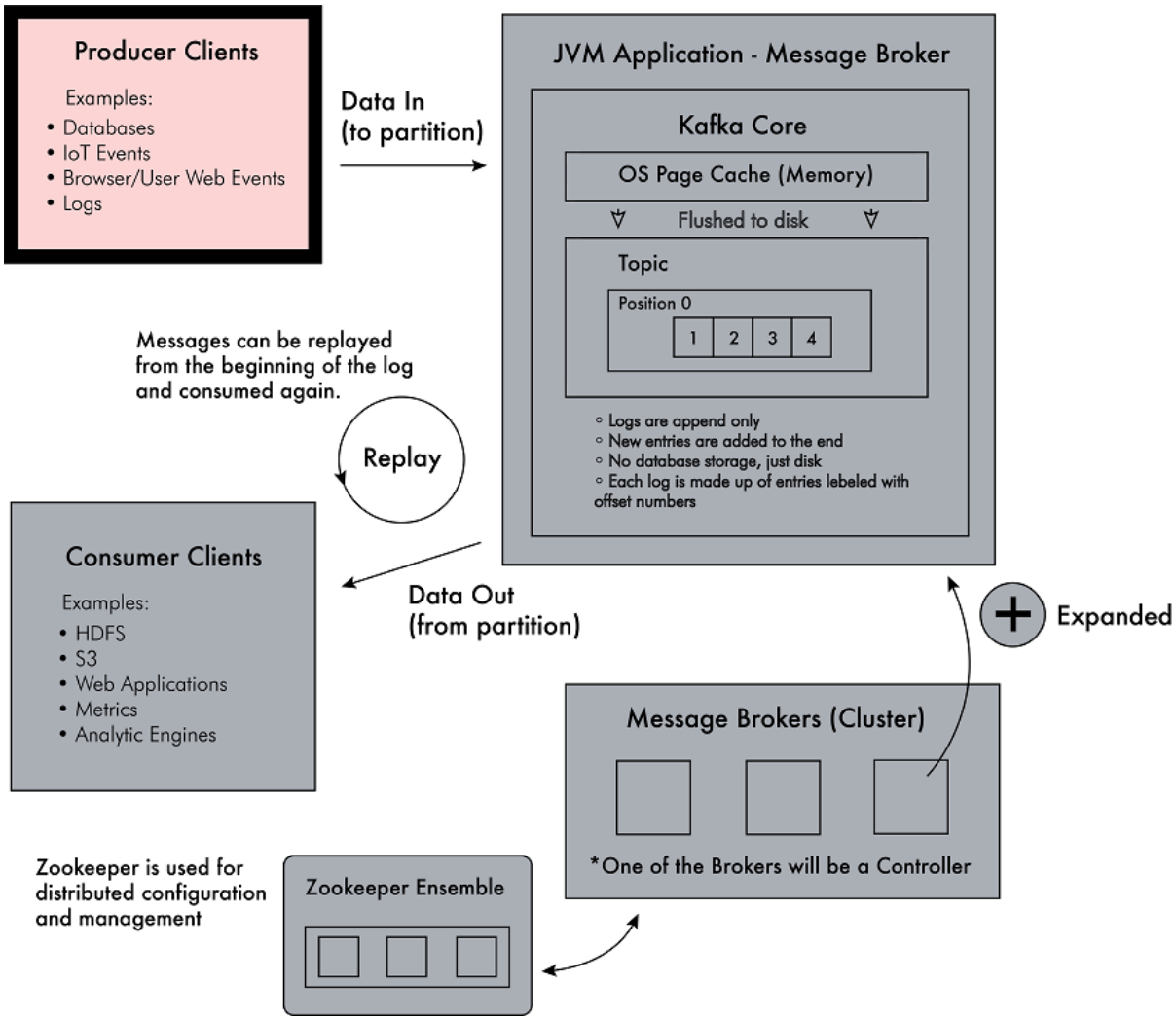

Figure 4.1 Kafka Producers

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, we are focusing on how data gets in Kafka. This data could be anything from database records, IoT events, events, or logs. To make it more concrete, let's imagine a practical example that you might have written for one of your projects already. Let's say that you have 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 your support staff watches that inbox to figure out 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 accessable than in a email

inbox. If we instead send this message into a Kafka topic, we could produce more robust responses rather than just reactive responses to customers.

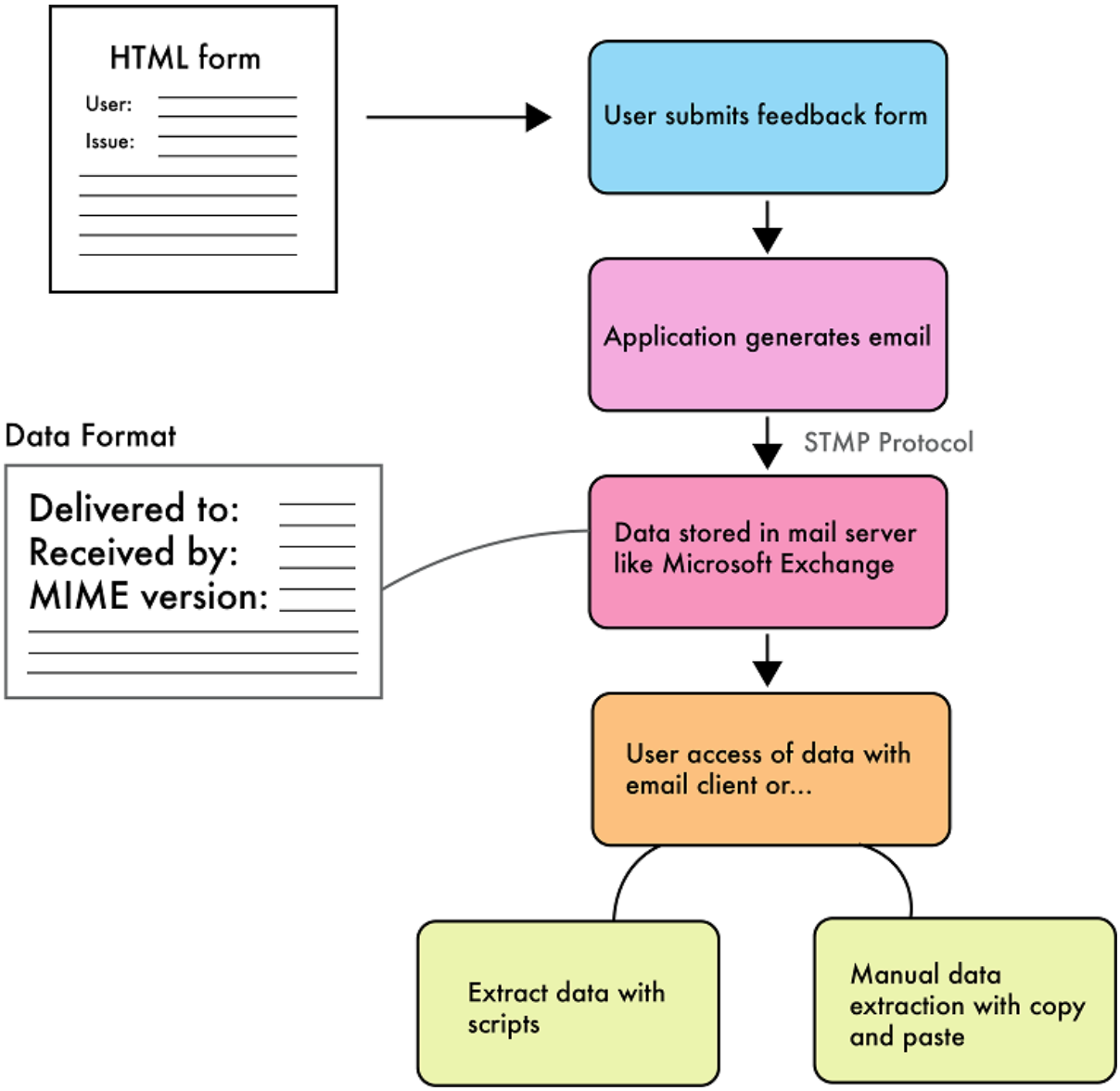

Figure 4.2 Sending data in email

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. Traditional email will use the SMTP protocol, and you 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 easily, but how else can you pull data out of that system for other uses rather than just reading email? Copy and paste are a common manual step of course as well as email parsing scripts. Parsing script would include using a tool or programming language and maybe even prebuilt 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. You should be able to enjoy writing the data in whatever format you choose.

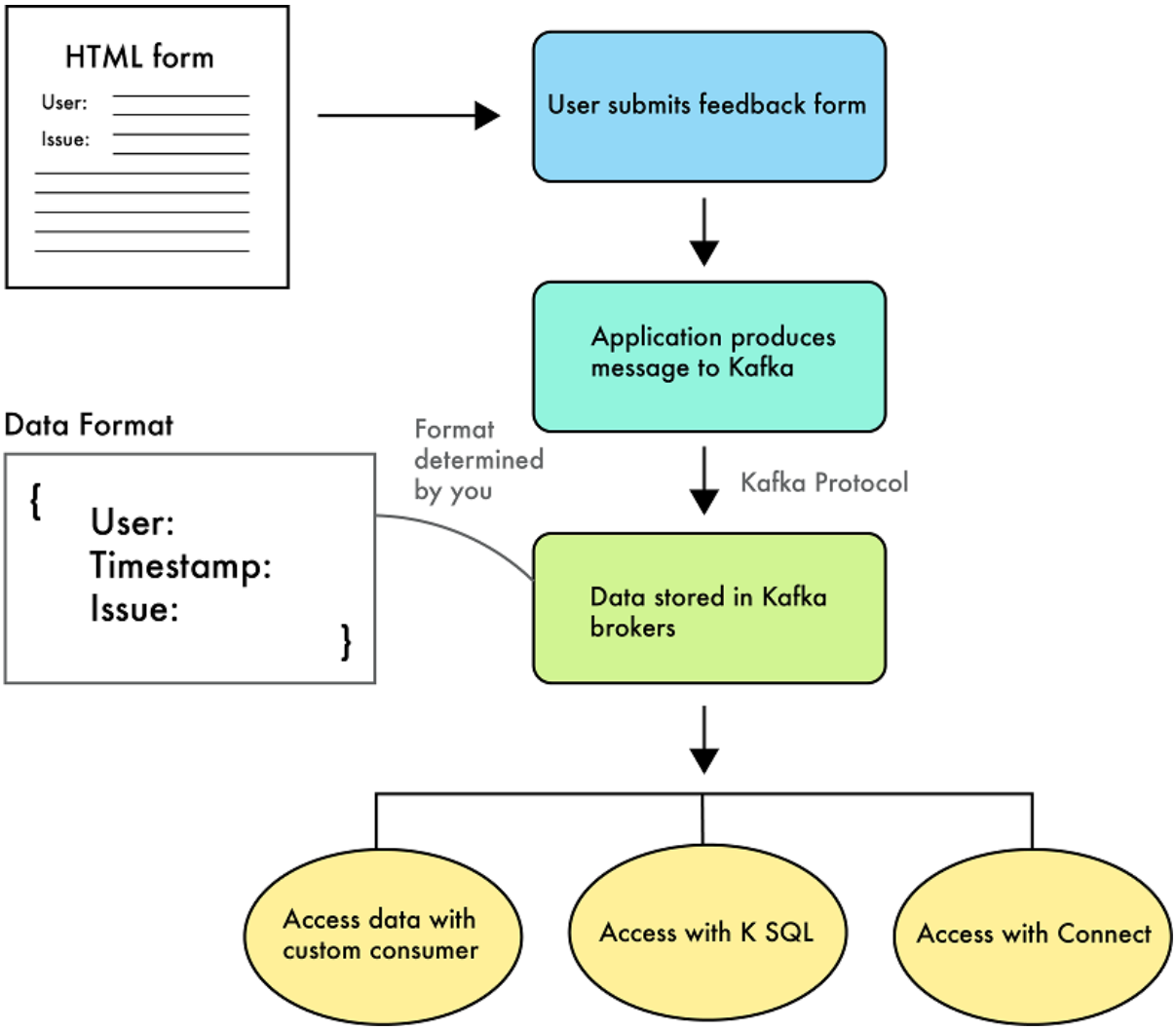

Figure 4.3 Sending data to Kafka

One other usage pattern that comes to mind is that idea that customer issues or website outages might be a temporary alert that is addressed and then deleted after replying to a customer. However, what if we wanted to start to turn this customer input into more uses than one. What if we were able to look for trends in outages that were reported by customers? Is there a trend that we can find? Does the site always slow to a crawl after sale coupon codes go out in mass marketing emails? Or what about finding the features that your users are missing from your site? Does 40% of your 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 you did have retention needs, it would be controlled by the teams

running your email infastructure vs a configuration setting you 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. You can extract the information that is important for you in whatever format you need and it can be used in many different ways. Your consuming applications can use schemes to know how to work with the data and will not be tied to a protocol format. You can retain and reprocess these messages for new use-cases since you control the retention on those events.

Now that we have seen an exmaple of why we might use a Producer, let's quickly look at how the wirte path works 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.

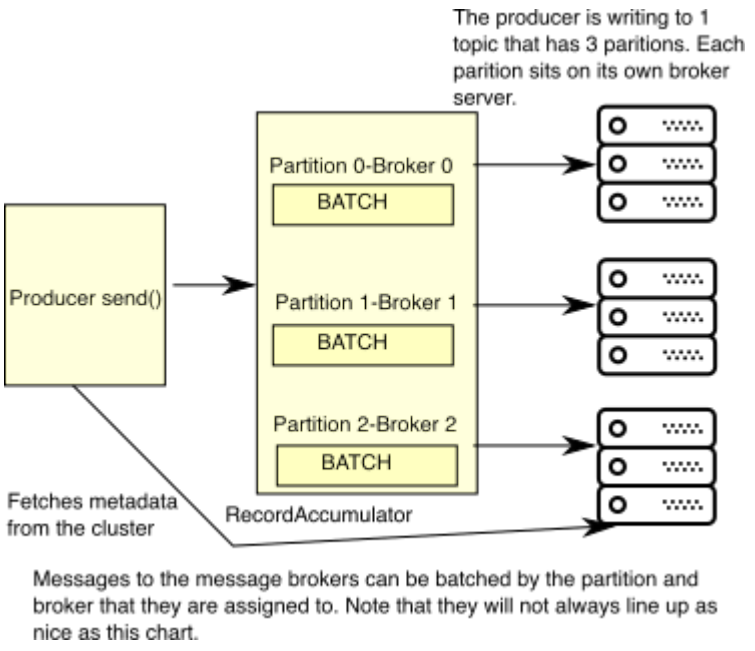

Figure 4.4 Producer Write Path

Looking at the figure shown, calling `send()` in our code puts some interesting machinery into motion. The `Sender` class provides the plumbing to work with the `RecordAccumulator` in order to send requests to the Kafka brokers from a producer. The senders job includes fetching metadata about the cluster itself. Since producers only write to the leader of the partition they are assigned, metadata helps the producer find the information about which actual broker is written to since the producer record might only have been given a topic name without any other details. This is nice in the fact that the end-user of the client does not have to make a seperate call to figure out the information.

The end-user needs to at least have a running broker to connect to and the client library can figure out the rest.

Following along with the write path flow, we can see that the record accumulators job is to 'accumulate' the messages into batches. These batches are grouped by the combination of the broker and partition that the messages are being sent to. 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 and every message was sent one at a time, a slow down likely could be seen in regards to how fast your messages are being processed. Another option to think about for your data is it you want to compress messages. If compression for the messages is used as well, the full batch is compressed. This likely would allow for a higher compression ratio than one message at a time.

It is important to note that if one message in the batch fails, then the entire batch fails for that partition. One question that comes up is if you really want to retry that batch and send those messages again. Since this distributed systems is used to account for transient errors, like a network blip, the logic is built-in for retires already. However, if ordering of the messages is important, then besides setting the retries to a non-zero number, you will also need to set the `max.in.flight.requests.per.connection` to less than or equal to 5. Setting `acks` to 'all' will provide the best situation for making sure your producer's messages arrive in the order you intend. These settings can also be set with the one configuration property `enable.idempotence`.

One thing you do not have to worry about is one producer getting in the way of another producer's data. Data will not be overwritten, but handled by the log itself and appended on the brokers log.

One thing that we need to cover next is how do we enable compression and the values like `max.in.flight.requests.per.connection`? Let's take a look at some of those details about how the producer uses configuration to enable our different requirements.

4.2 Important Configuration

One of the things that was interesting when I started working with sending data into Kafka was the ease of configuration. If you have worked with other queue or messaging systems, it seems like the setup can include everything from providing remote and local queues lists, manager hostnames, starting connections, connection factories, and sessions. While far from being setup free, the producer will work from configuration on its own to retrieve some of the information it needs (such as a list of all of your message brokers). Using the value from the property `bootstrap.servers`, the producer you create will talk to the broker with which it will write to without going through a router or proxy for each request after it established its connection and fetches metadata about the partition it needs.

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 you could choose to set. One way to deal with all of the producer config key names is to use the constants provided in `ProducerConfig`.

4.2.1 Producer Configuration

Table 4.1 Important Producer Configuration

Key	Purpose
acks	Number of acknowledgments producer requires before a success is considered
bootstrap.servers	List of Kafka message brokers to connect to on 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 type (if any) of how messags are compressed

With so many options, the Kafka documents have a helpful feature in order for you to know which ones might have the most impact. Look for the *IMPORTANCE* label of High in the documentation listed at kafka.apache.org/documentation/#producerconfigs (-PROD needs to shorten). Let's go through some of the configuration options listed that we have not discussed before and what they mean in more context.

4.2.2 Configuring the Broker list

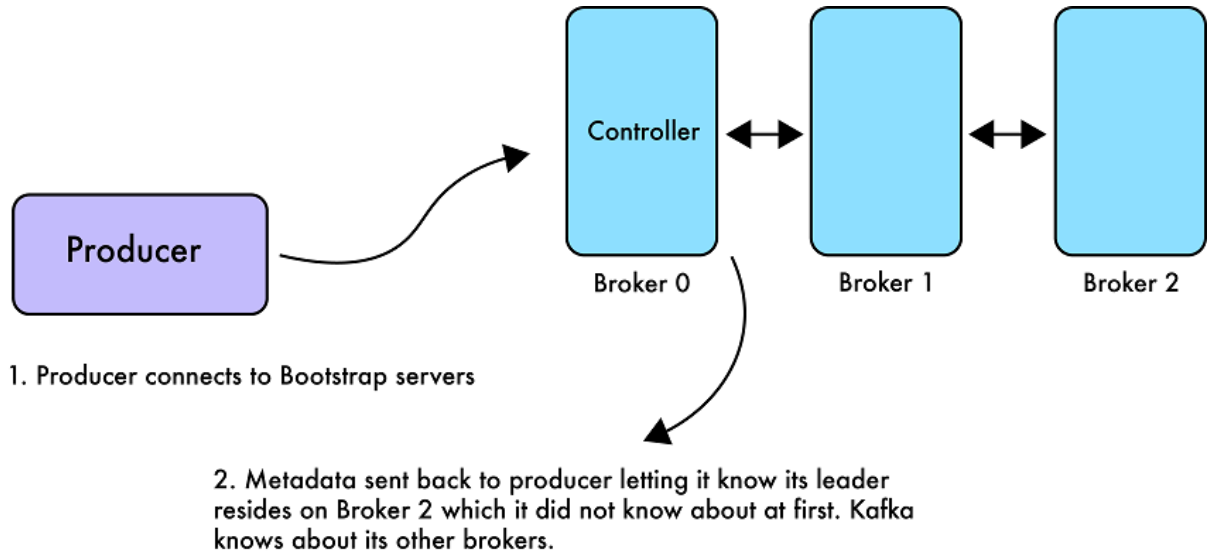

Figure 4.5 Bootstrap Servers

It is pretty clear that we have to tell the producer which topic we need to send messages to. However, how does it know where that topic actually resides? As you'll recall from earlier, you know that topics are made up of partitions. However, we do not have to know the assignment of the partitions when we are sending messages. You will notice that one of the required configuration options is `bootstrap.servers`. This property can take a list of message brokers. It is a good idea to send a list rather than one in case one of the servers is down or in maintenance. This configuration is key to helping the producer find a broker to talk to. Once the producer is connected to the cluster, it can then obtain metadata that it needs in order to find out those details we did not have to provide above. Producer clients can also overcome a failure of the partition they are reading from since they can use the information about the cluster that they know how to find out by requesting metadata.

4.2.3 How to go Fast (or Safer)

While asynchronous message patterns in a system are one reason that many use queue-type systems, asynchronous sending of messages is also a feature provided by the producer for its communication with the message brokers. You can wait in your code for the result directly, but it's nice to know the option is available with callbacks that can be provided to handle responses and/or failure conditions.

One of the configuration properties that will apply to our scenarios the most will be the `acks` key which stands for acknowledgements. This controls how many acknowledgments the producer needs to receive from the leader's followers before it

returns a complete request. Figure 4.6 shows how a message with ack=0 will behave. Setting this value to 0 will probably get you the lowest latency, but at a cost of durability. Guarantees are not made if the any broker actually received the message and retries are not attempted. For example, let's say that you have a web tracking platform that is following the clicks on a page and sends these events into Kafka. In this siutation, it might not be a big deal to lose a link press event or hover event. If it is lost, there is no real business impact.

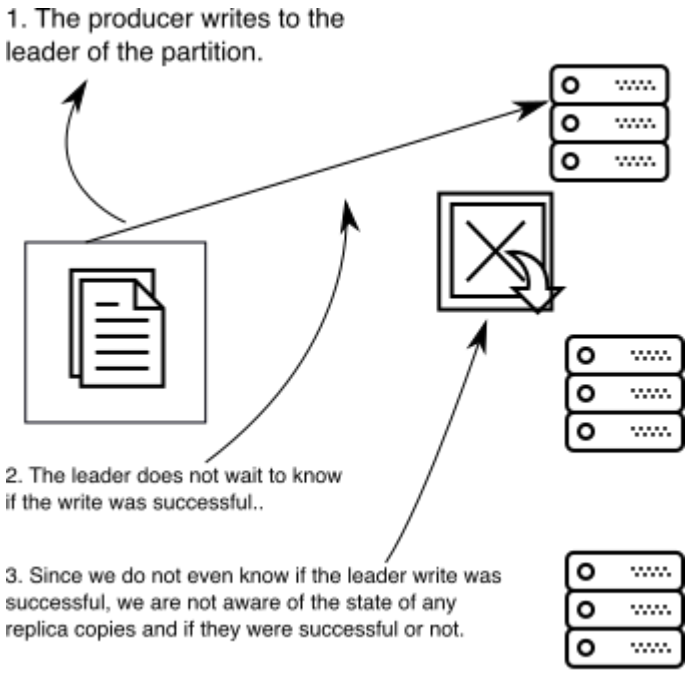

Figure 4.6 Property acks=0

Figure 4.7 shows the impact of setting the acks to 1 and asking for an acknowledgement. An acknowledgement would involve the receiver of the message sending confirmation back to the leader broker for that partition. The leader would wait for that acknowledgement, which is something to watch for if you are considering the various options for this property value. Setting the value to 1 will at least mean that the leader has received the message. However, the followers might not have copied the message before a failure brings down the leader. If that error timing occurs, then the message could appear that it had never made it to the cluster.

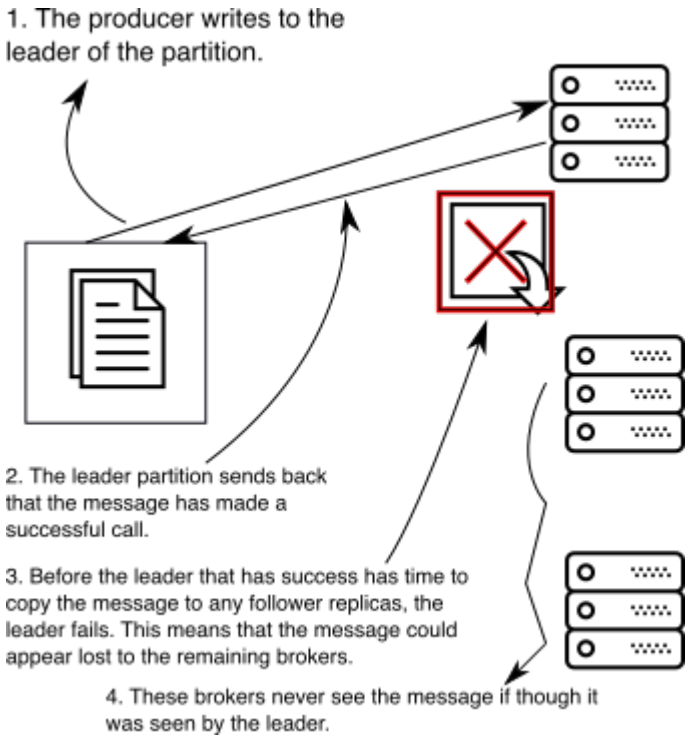

Figure 4.7 Property acks=1

Values all or -1 are the strongest available option. Figure 4.8 shows how this value mean that the leader will wait on all of the in-sync replicas (ISRs) to acknowledge they have gotten the message. While this might be the best for durability, it is easy to see that it might not be the quickest due to the dependencies it has on other brokers. In some cases, you will likely pay the price for guarantees if you really do not want to lose data. If you have a lot of brokers in your cluster, make sure you are aware of how many brokers the leader would have to wait on. The broker that takes the longest to reply will be your fastest time before the producer has a success message.

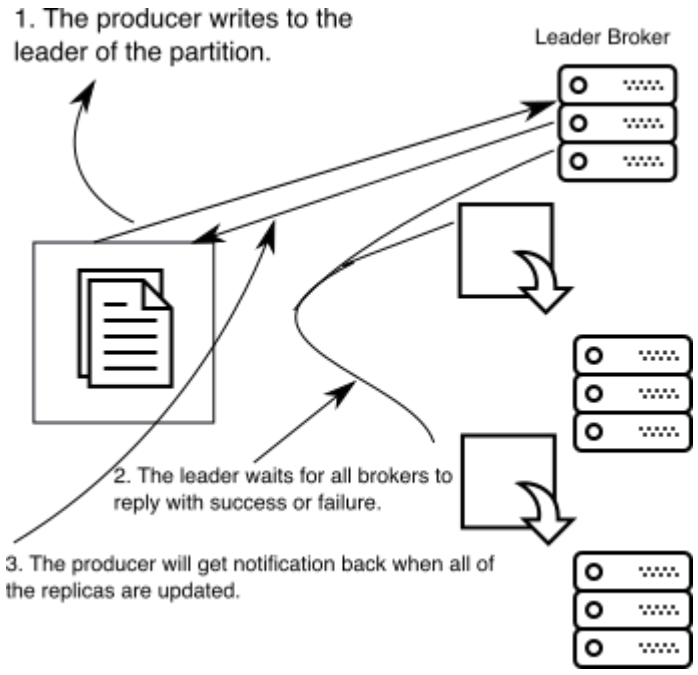

Figure 4.8 Property acks=all

> **NOTE** **What Committed Means**
>
> Your message will not be seen by consumers until it is considered to be committed. However, this status is not related to the `acks` setting given in our producer configuration. We will discuss more about when the messages become available for consumption later which involves the data being received by all the replicas in its in-sync replica list.

4.2.4 Timestamps

Recent versions of the producer record will have a timestamp. A user can either pass the time into the constructor as a Java type `long` or the current timestamp will be used. The actual time that is used by the system can depend on this time or can be a broker timestamp that occurs when the message is logged. A topic configuration setting of `CreateTime` will respect and use this time set by the client. As always, timestamps can be tricky. One such example is that you might get records with a timestamp that is actually earlier than a timestamp before it, i.e. in cases where you maybe had a failure and resend. The data will be ordered in the log by offsets and not by timestamp. While reading this time stamped data is a consumer concern, it is also a producer concern as well. It is good to think about if you want to allow retries or many inflight requests at a time. If a retry happens and other requests succeeded on their first call, your messages might occur after those 'later' messages. Figure 4.9 is an example of when a message can get out of order. Even though message 1 was sent first, it might not make it into the ordered log if retries are enabled.

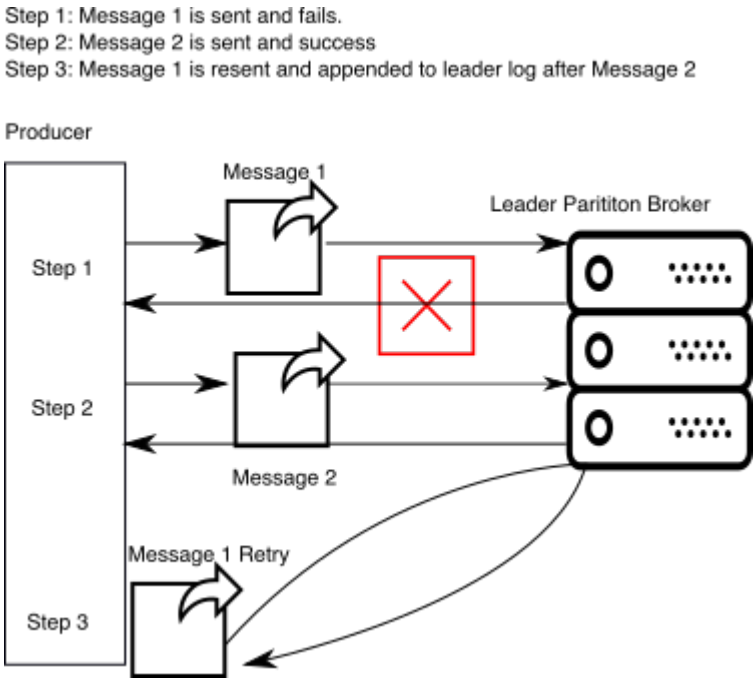

Figure 4.9 Retry Impact on Order

As a reminder, if you are using a Kafka version before 0.10, you will not have timestamp information to rely on as it was not a feature included as part of earlier releases. You can still include timestamp, but it would be part of the value of your message itself that you would need to include.

4.2.5 Adding compression to our messages

One of the topics that I touched on briefly above when talking about message batches was compression. Snappy, gzip, lz4, and none are all options that can be used and are set with the configuration key `compression.type`. If the size of data being sent is a concern, like the need to mirror data across various data centers, compression might be a feature you 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. However, do not forget that CPU and memory are needed for this work and that tradeoff might not make sense in every use case.

Compression is done at the batch level. Small messages being compressed do not necessarily make sense in all cases. In addition, if you have so low traffic or events that you do not have that many messages, you might not gain any more from compression. The size and rate of your data is something that you should try to work with to decide what is the best use for your own situation.

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, we reached for the class: `io.confluent.kafka.serializers.KafkaAvroSerializer`. But what if you have a specific format that you need to produce? This often happens when you are trying to use a custom object and want to use it as it exists today. For our context, serialization is used to translate our data into a format that can be stored, transmitted, and then retrieved to achieve a clone of our original data. 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, we reached for the class: `io.confluent.kafka.serializers.KafkaAvroSerializer`. But what if you have a specific format that you need to produce? This often happens when you are trying to use a custom object and want to use it as it exists today. For our context, serialization is used to translate our data into a format that can be stored, transmitted, and then retrieved to achieve a clone of our original data.

Listing 4.1 Alert Class

```
    private int alertId;
    private String stageId;
    private String alertLevel;
    private 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 messages of the alerts.

Listing 4.1 shows code that we have used to create a bean named Alert in order to hold the information we want to send. Those familiar with Java will notice that it is nothing more than a POJO (plain old java object) with nothing more than getters and setters in this instance.

Listing 4.2 Custom Serializer

```
public class AlertKeySerde implements Serializer<Alert>,
Deserializer<Alert> {                                          (1)

    public byte[] serialize(String topic, Alert value) {       (2)
        if (value == null) {
            return null;
        }

        try {
            return value.getStageId().getBytes("UTF8");        (3)
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }

    }


    public void close() {
        // nothing needed
    }

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing needed
    }

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

}
```

(1) We implement the Serializer interface to be a valid extension point.

(2) The topic as well as the Alert object itself is sent into our method.

(3) Our end goal is to convert objects to bytes. <4>The rest of the interface methods do not need any logic at this point.

It is interesting to note that you can serialize key and values with different serializers on the same message. So be careful about your intended serializers and your configuration values for both. In Listing 4.2, we are planning on just using this custom class as the key serializer class at the moment and leaving the value serializer as a `StringSerializer`. The code implements the `Serializer` interface and pulls out just the field `stageId` to use as our key for our message.

NOTE **What is Serde?**

If you see or hear the term `Serde`, it simply means that the serializer and deserializer are both handled by the same implementation of that interface. However, it is still common that you will also see each interface defined separately. Just watch when you use `StringSerializer` vs `StringDeserializer`, the difference can be hard to spot!

Another thing you should 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.

4.2.7 Creating custom partition code

The client also has the ability to control what partition it writes to. This power can be one way to load balance the data over the partitions. This might come into the picture if you have specific keys that you might want to avoid all being hashed to the same partition due to the volume you expect. For example, if two customers produce 75% of your traffic, it might be a good idea to split those two customers into different partitions in order to avoid filling your storage space on specific partitions only.

The default for a null key is a round-robin strategy. If a key is present, then the key is used to create a hash to help assign the partition. However, sometimes we have some specific ways we want our data to be partitioned. One way to take control over this is to write your own custom partitioner class.

One example we can think about is alert levels from our service. Some sensors information might be more important than others, ie., they might be on the critical path of our work line that could cause downtime if not addressed. Let's say we have 3 levels of alerts: informational, warning, and critical. We could create a partitioner that would place the different levels in different partitions. Your consumers could always make sure to read the critical alerts before processing the others. If your consumers were keeping up with the messages being logged, it probably wouldn't be a huge concern. However, this example shows that you could change the partition assignment with a custom class.

Listing 4.3 Custom Partitioner

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;

import java.util.*;
```

```
public class AlertLevelPartitioner implements Partitioner {      (1)

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

        final List<PartitionInfo> partitionInfoList =
                cluster.availablePartitionsForTopic(topic);
        final int partitionSize = partitionInfoList.size();
        final int criticalPartition = partitionSize - 1;
        final int partitionCount = partitionSize- 1;

        final String key = ((String) objectKey);


        if (key.contains("CRITICAL")) {                           (2)
            return criticalPartition;
        } else {
            return Math.abs(key.hashCode()) % partitionCount;
        }

    }

    public void close() {
        // nothing needed
    }

    public void configure(Map<String, ?> configs) {
        // nothing needed
    }

}
```

(1) We need to implement the Partitioner interface.

(2) We are casting the key to a String in order to check the value.

Listing 4.3 shows an example of 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 as the value of the key to make sure that any `CRITICAL` events make it to a specific place. In addition to the class itself being created, Listing 4.4 shows how the configuration key `partitioner.class` will need to be set in order for your producer to use this custom class you created. The configuration that powers Kafka is used to leverage your new class.

Listing 4.4 Custom Partitioner Class Config

```
Properties props = new Properties();
...
props.put("partitioner.class",                                    (1)
"com.kafkainaction.partitioner.AlertLevelPartitioner");
```

(1)

We need to set our producer config to use our custom partitioner

4.2.8 Producer Interceptor

One of the options when using the producer is creating producer interceptors. They were introduced in KIP-42 whose main design was to help support measurement and monitoring. It is true that you can modify the message and since the interceptor will run before a partition gets assigned, it is important to note any key changes. The client might not know what to expect if the key changes the partition assignment they intended. In comparison to Kafka Streams workflows, the usage of these interceptors might not be your first choice. Kafka also does report various internal metrics using JMX that can be used if monitoring is a main concern.

If you do create an interceptor you wish to use, remember to set the producer config `interceptor.classes`. At the current time, there are no default interceptors that run in the life cycle.

4.3 Generating data for our requirements

Let's try to use this information we gathered about how producers work in order to see if we can start working on our own solutions.

Let's start with our audit checklist. As noted in Chapter 3, we wanted to make sure that we did not lose any messages and did not need to correlate any events across the individual events. One of the first things that we should look at is how to make sure we can keep from losing any messages. Listing 4.5 shows how we would start our producer config. We are trying to make sure that we are being safe by making sure that `acks=all` is set, for example.

Listing 4.5 Audit Producer Config

```
Properties props = new Properties();                                    (1)
props.put("bootstrap.servers", "localhost:9092,localhost:9093");
props.put("acks", "all");                                               (2)
props.put("retries", "3");                                              (3)
props.put("max.in.flight.requests.per.connection", "5");
```

(1) We start with creating properties for our configuration as before.

(2) We are using acks=all in order to get the strongest guarantee we can.

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

Did you 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 change that has a major 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 even using a key for these messages. There is an important part that we do want to change. That means waiting for the response to complete in a synchronous way and is shown in Listing 4.6. The `get` method is how we are wating for the result to come back before moving on in the code.

Listing 4.6 Synchronous wait

```
RecordMetadata result = producer.send(producerRecord).get();          (1)
System.out.printf("offset = %d, topic = %s,
timestamp = %Tc %n", result.offset(), result.topic(),
result.timestamp());
```

(1) 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 is one way to can make sure our code is handling each record as it comes back.

Listing 4.7 Health Trending Producer

```
Properties props = new Properties();
...
props.put("key.serializer",
"com.kafkainaction.serde.AlertKeySerde");                              (1)
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

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

Alert alert = new Alert(0, "Stage 0", "CRITICAL", "Stage 0 stopped");

ProducerRecord<Alert, String> producerRecord = new ProducerRecord<Alert,
   String>("healthtrend", alert, alert.getAlertMessage());             (2)

producer.send(producerRecord);
```

(1) We need to let our producer client know how to serialize our custom Alert object into a key

(2) Instead of null for the 2nd parameter, we put the actual object we wish to use help populate the key.

As a reminder, we were interested in trying to take our sensor health status and track their

health over time. Since we care about the information about one sensor (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 sensor 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. 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. 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 outages. We do want to group by the sensor id as well in this case as a key. One reason is so we can tell if that sensor was failed or recovered (any state change) by looking at only the last event for that sensor id. We do not care about the history of the status checks, only what is 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 so those that process the data would have access to the critical alerts specifically and could go after other alerts when those were handled.

Listing 4.8 Alert Producer

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092,localhost:9093");
props.put("key.serializer",
"com.kafkainaction.serde.AlertKeySerde");                    (1)

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("partitioner.class",
"sf.kafkainaction.AlertLevelPartitioner");                   (2)

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

Alert alert = new Alert(1, "Stage 1", "CRITICAL", "Stage 1 stopped");

ProducerRecord<Alert, String> producerRecord = new ProducerRecord<Alert, String>("alert",
   alert, alert.getAlertMessage()); //


producer.send(producerRecord, new AlertCallback());         (3)
```

(1) We are reusing our custom Alert key serializer

(2) We are using the property partitioner.class to set our custom partitioner class we created above.

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

One addition you will see to 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 frequency of events, it might be a good idea to make sure that we are not seeing 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 will log a message only if an error occurred.

Listing 4.9 Alert Callback

```
public class AlertCallback implements Callback {                                    (1)

        public void onCompletion(RecordMetadata metadata, Exception exception) {    (2)
            if(exception != null){
                System.out.printf("Error sending message: "+ "offset = %d, topic = %s,
                timestamp = %Tc %n", metadata.offset(), metadata.topic(), metadata.timestamp());
            }
        }
}
```

(1) We need to implement the Kafka interface Callback

(2) The completion can have a 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 features. When Kafka is used as a 'sink', Flume places data into Kafka. In the examples, we will be looking at Flume version 1.8 located at github.com/apache/flume/tree/flume-1.8 if you also 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.

Listing 4.10 Flume Sink Configuration

```
a1.sinks.k1.kafka.topic = mytopic
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 own code, we declared the configuration inside our code as properties. However, this is showing an example of an application that choose to externalize the configuration values and is something we could use on our own projects. Now, let's look at the KafkaSink source code. You can find a copy at (

raw.githubusercontent.com/apache/flume/flume-1.8/flume-ng-sinks/flume-ng-kafka-sink/src/m - PROD needs to shorten). Please note that I am only showing specific lines from the file to highlight the familiar producer concepts. Refer to the complete file for proper syntax, imports, and the complete class as the code below will not execute without error. I have left method names so make it easier to locate later and removed source code comments to not conflict with my notes.

Listing 4.11 Flume Kafka Sink Configuration and send()

```
public class KafkaSink extends AbstractSink implements Configurable {
...
  public Status process() throws EventDeliveryException {
   ...
          if (partitionId != null) {
            record = new ProducerRecord<String, byte[]>(eventTopic, partitionId, eventKey,
                serializeEvent(event, useAvroEventFormat));
          } else {
            record = new ProducerRecord<String, byte[]>(eventTopic, eventKey,
                serializeEvent(event, useAvroEventFormat));
          }                                                                          (1)

          kafkaFutures.add(producer.send(record, new SinkCallback(startTime)));     (2)
   ...
      }
   ...
      producer.flush();
  }
  public synchronized void start() {
    producer = new KafkaProducer<String,byte[]>(kafkaProps);                        (3)
    ...
  }
...
  private void setProducerProps(Context context, String bootStrapServers) {
    kafkaProps.clear();
    kafkaProps.put(ProducerConfig.ACKS_CONFIG, DEFAULT_ACKS);                       (4)
    kafkaProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, DEFAULT_KEY_SERIALIZER);
    kafkaProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, DEFAULT_VALUE_SERIAIZER);
    kafkaProps.putAll(context.getSubProperties(KAFKA_PRODUCER_PREFIX));
    kafkaProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootStrapServers);
  }

}
```

(1) Logic to choose which ProducerRecord constructor to create the message.

(2) Our familiar producer.send with a callback.

(3) Creating our producer with properties.

(4) Setting our configuration.

While there is more code in `KafkaSink` that I have omitted, the core Kafka producer pieces might be starting to look familiar in Listing 4.11. Setting configuraiton and the

producer send() method should all look like 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 that will have.

Going further in that same real-world class example, it might be helpful to see how our own `AlertCallback.java` compares to the Kafka Sink callback in Listing 4.12. The Flume example uses the `RecordMetadata` object to learn more information about successful calls. This information can help us learn more about where the producer message was actually written to including the partition and offset within that specifc partition. Their logging takes place on exceptions as well as on success.

Listing 4.12 Flume Kafka Sink Callback

```
class SinkCallback implements Callback {                    (1)
...
  public void onCompletion(RecordMetadata metadata, Exception exception) {
    if (exception != null) {
      logger.debug("Error sending message to Kafka {} ", exception.getMessage());
    }
    if (logger.isDebugEnabled()) {
      long eventElapsedTime = System.currentTimeMillis() - startTime;
      if (metadata != null) {
        logger.debug("Acked message partition:{} ofset:{}", metadata.partition(),
                metadata.offset());
      }
      logger.debug("Elapsed time for send: {}", eventElapsedTime);
    }
  }
}
```

(1) Callback logic to log errors as well as success.

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 really 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.

We crossed a major hurdle by starting to get data into Kafka. Now that we are in the 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.

4.3.1 Client and Broker Versions

One of the things to take note of is that Kafka and clients 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, the broker will handle this upgrade in the message version. However, because you can does not mean that you should in all cases. Some of the things that make Kafka fast can be impacted by a choice 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 tradeoff 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, you can take a peek at KIP-35 and KIP-97. Just be aware of the cost of the version changes and watch for any impacts to your applications when versions change.

Now that we have some ideas about the 'how' we get data into Kafa, we can start to work on learning more about making that data useful to other applications by getting that data back out. Consumer clients will be a key part of this discovery, and like producers, there are various configuration-driven behaviors that we can use to help us satisfy our various requirements for consumption.

4.4 Summary

In this chapter we learned:

- How we can control where our data gets written at a partition level.
- To create our own serializers for our own Java objects.
- About using our new knowledge of producers and their configuration to met our requirements.

5 Consumers: Unlocking Data

This chapters covers:

- Exploring the consumer read path and how it works
- Learning about offsets and their use
- Examining various configuration options for consumer clients

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 can exist outside of brokers, they have the same language options available as we do the 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. As with producers, the actual consumer processes can run on separate machines. Consumers are not required to run on a specific server, as long as they can connect to the message brokers, they are good to read messages.

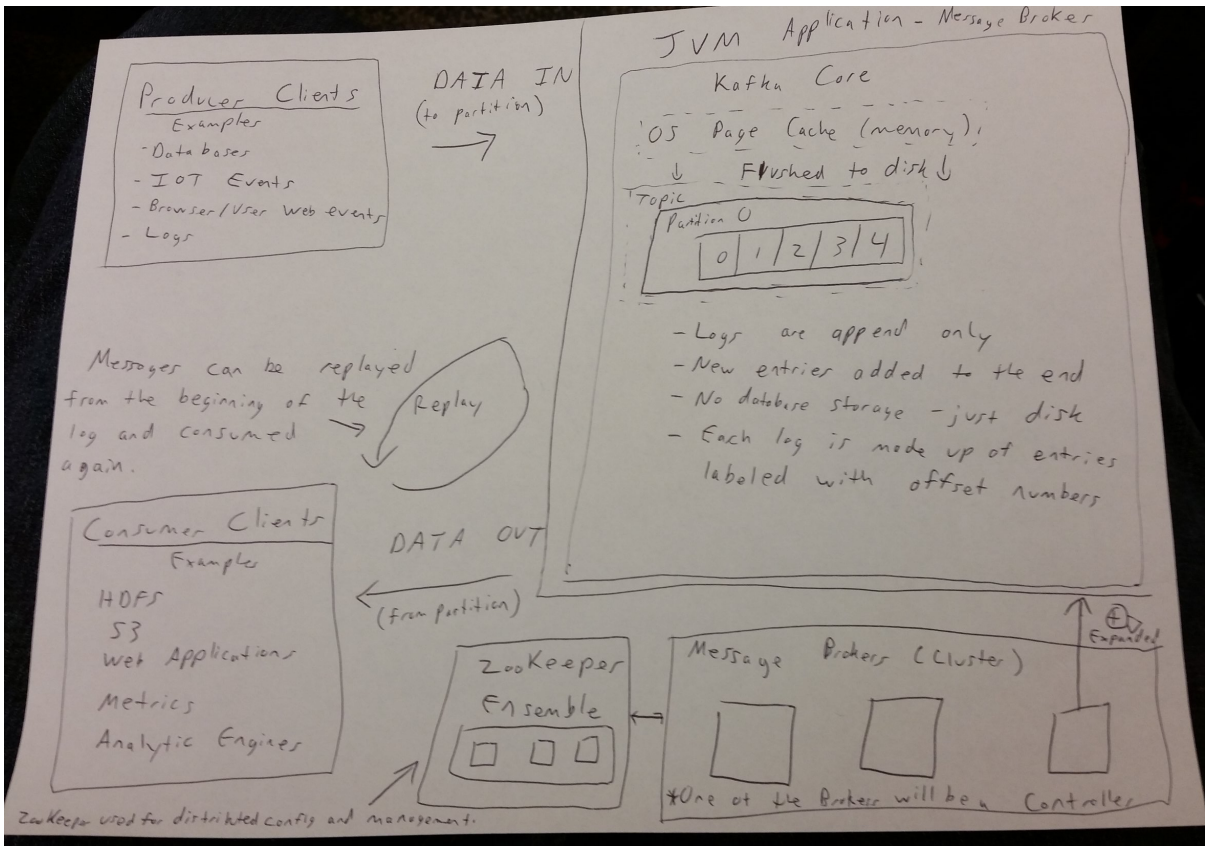

Figure 5.1 Kafka Consumers

Why is it important to know that the consumer is subscribing to topics and not being pushed to instead? The power of processing control really shifts to the consumer in this situation. 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 consumers resources to be exhausted and fail. With the consumer being in the driver seat, if a failure occurred and the applications comes 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 used to handling this constant data flow or backpressure, you need to know that you are not a listener to the brokers, you are the one polling 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 time 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.

As a reminder from our first introduction to consumers, it is important to note that the Kafka consumer is not thread-safe. Your application that uses the consumer will need to

make sure you take steps to make sure you are working with synchronized code in mind. In other words, you can get a ConcurrentModificationException if you have many threads going at once. In most of the examples in this book, the usage will be one consumer per thread and will try to avoid these situations from cropping up.

Let's take a look at reading from a topic with one consumer like we did in Chapter 2. For this instance, we have an application that is using click events on the web site to project future click rates with a new promotion. Let's say that we have a specific formula that we use to uses the time a user spends on the page as well as the number of interactions they had on the page. Imagine we run the consumer and process all of the messages on the topic and are happy with our application of the formula.

Listing 5.1 Web Click Consumer

```
public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092,localhost:9093");
    props.put("group.id", "helloconsumer");                                    (1)
    props.put("key.deserializer",                                              (2)
    "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",                                            (2)
    "org.apache.kafka.common.serialization.StringDeserializer");

    KafkaConsumer<String, String> consumer =
    new KafkaConsumer<String, String>(props);                                  (3)

    consumer.subscribe(Arrays.asList("webclicks"));                            (4)

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);          (5)
        for (ConsumerRecord<String, String> record : records)
            System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),
                record.key(), record.value());
            System.out.printf("value = %s%n",
            record.value() * 1.543);
    }

    // consumer.close(); //unreachable code at the moment
}
```

(1) Note that we are using a group.id in this instance. We will see how those help group multiple consumers into reading all of the messages of a topic.

(2) We are using deserializers for the key and values which is different if than having serializers for producers.

(3) We use our properties to create an instance of a consumer

(4) We are choosing this topics we want to subscribe to. In this case we are only interested in the webclicks topic.

(5) We poll in an infinite loop for records that will come from the webclicks topic. The

100 used (in milliseconds) is the amount of time we are willing to wait if data is not available. This number is an example value that can be changed depending on your data processing.

After reading all of the data, we find out that our modeling formula wasn't correct! So what do you do now? Attempt to un-caluclate the data we have from our end-results (assuming the correction would be harder than in this example)?

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 data. Developer mistakes, application logic mistakes, and even dependent application failures have a chance to be corrected with our topics not removing the data once read one time.

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

5.2 Important Configuration

Important Consumer Configuration You will notice a couple of properties that are similar to the ones that were important for the producer clients as well. We always need to know the the brokers we can attempt to connect to on 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.

Table 5.1 Consumer Configuration

Key	Purpose
bootstrap.servers	List of Kafka message brokers to connect to 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	A id used for being able to identify a client
fetch.min.bytes	How many bytes your consumer should wait for until a timeout happens
heartbeat.interval.ms	The time between when the consumer pings the group coordinator
max.partition.fetch.bytes	This is the max amount of data the server will return. This is a per-partition value
session.timeout.ms	How long until a consumer not contacting a broker will be removed from the group as a failure

Differences in the configuration needed revolves around the fact that we are more focused on how we can let the brokers know we are actively reading data as well as how much data we want to consume. Making sure that our session timeout value is greater than the heartbeat interval is important in order to make sure that your consumer is letting other consumers know we are likely still processing our messages. These settings will help as when we talk more about consumer groups later in this chapter.

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 are really use an index position in the log that the consumer sends to the broker to let it know what messages it wants to consume from where. If you think back to our console consumer example, we used the flag `--from-beginning`. This really set the consumer configuration `auto.offset.reset` to `earliest`. With that configuration, you should be able to see all of the messages on that topic for the partitions you are connected to-even if they were sent before you started up the console consumer.

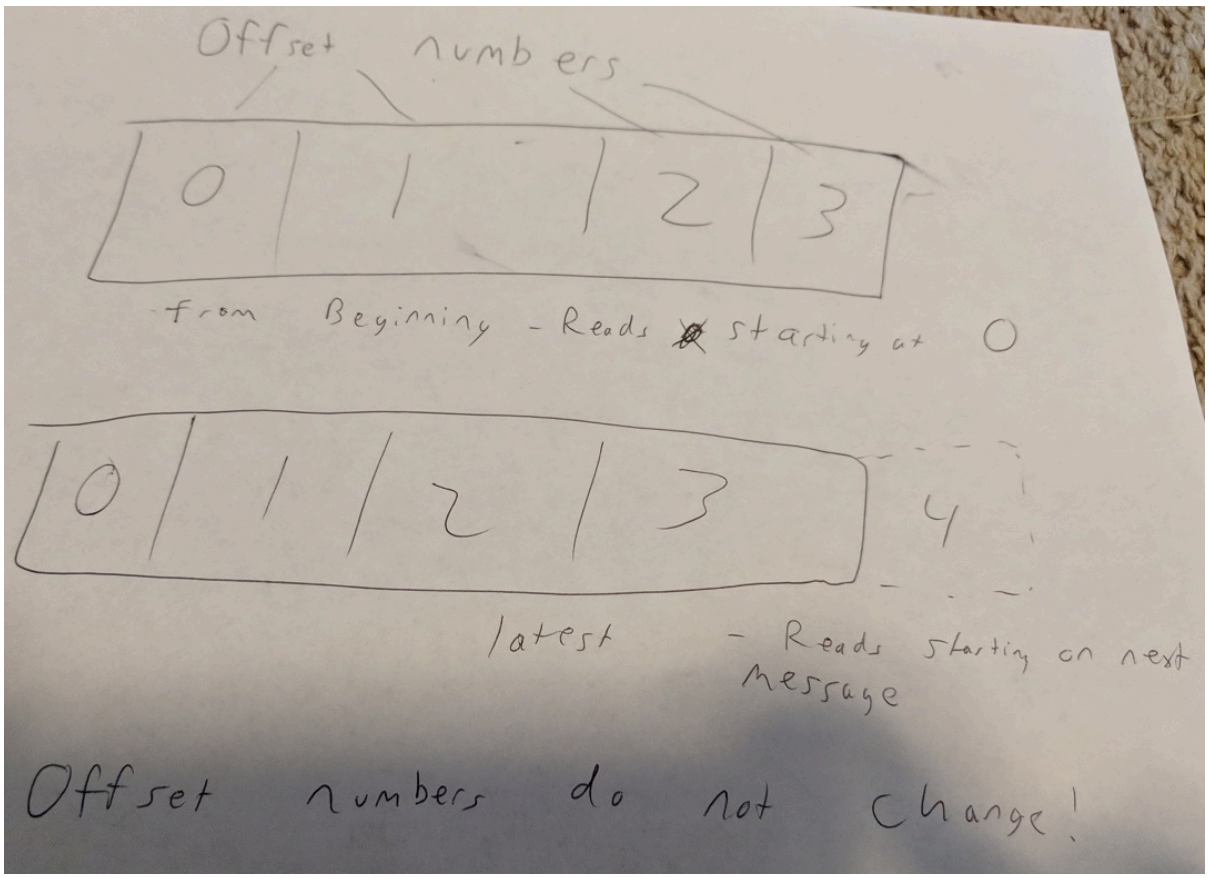

Figure 5.2 Kafka Offsets

If you didn't add that option, the default is `latest`. In those cases, you didn't see any messages from the producer unless you sent them after you had started the consumer. It is saying disregard processing the messages that I have not seen, I only want to process what comes in after my consumer joins. You can think of it as an infinite array that has 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.

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 bounds. 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 seen as an issue that likely won't occur.

For a message written to a topic, what would the coordinates be to find the message? Using a topic, we would want to find the partition it was written to. Not just any partition, the lead partition. Then you would find the index-based offset.

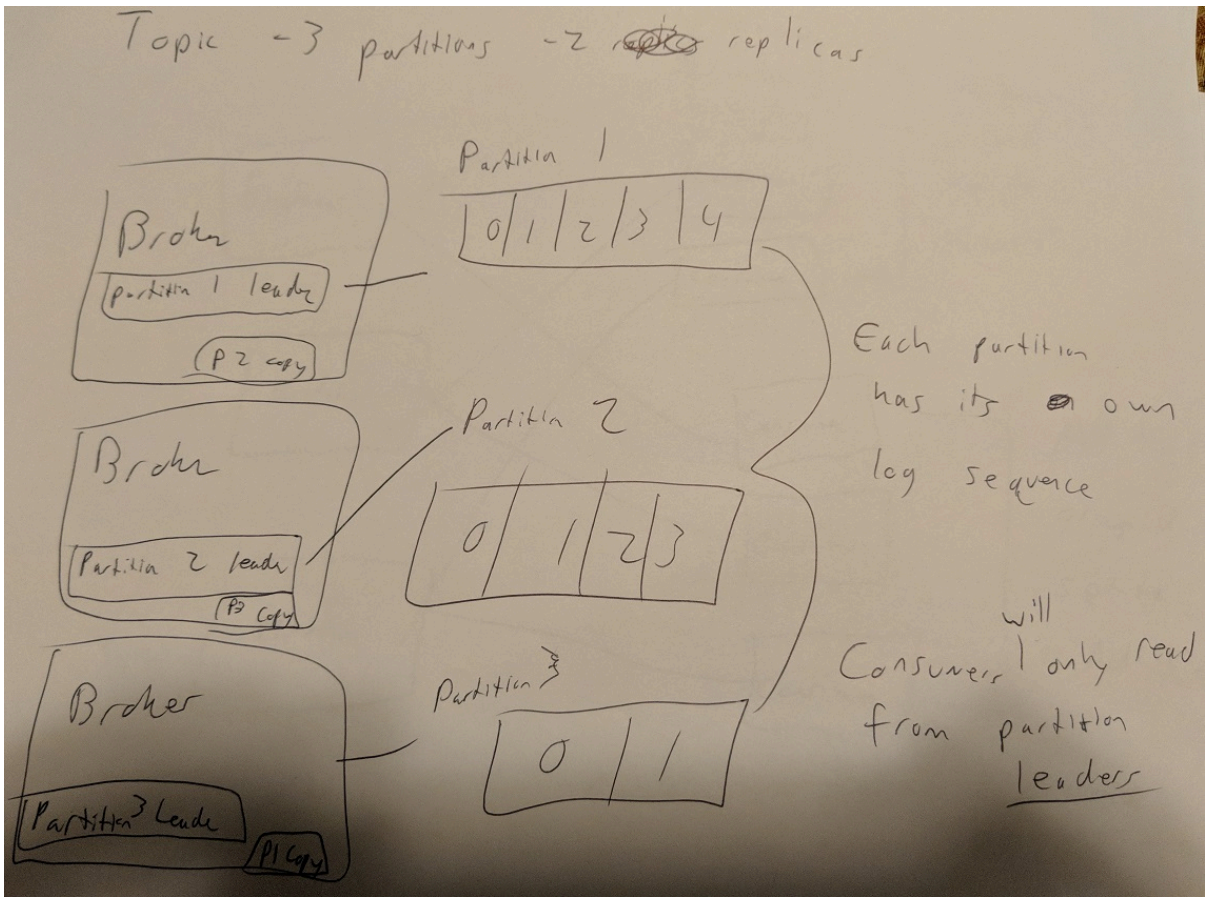

Figure 5.3 Partition Leaders

As Figure 5.3 shows, consumers only read from the consumer's leader partition. Replicas are used in the case of failure 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 taking about within a topic.

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 actually read from the leader 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 consumer group, one of the brokers takes on the role of a group coordinator. The client will talk to this coordinator in order to get an assignment of which details it needs to consume.

The number of partitions also come into play when talking about consumption. If there are more consumers in a group then the number of partitions, then some consumers will be idle. 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.

Since the number of partitions determines the amount of parallel consumer 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. Systems often involved tradeoffs, and this is why you will need to choose what 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 have in-sync replicas and is done before a message can even be available for a message 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 a partition to a consumer, the more partitions a consumer consumes will likely have more memory needs overall.

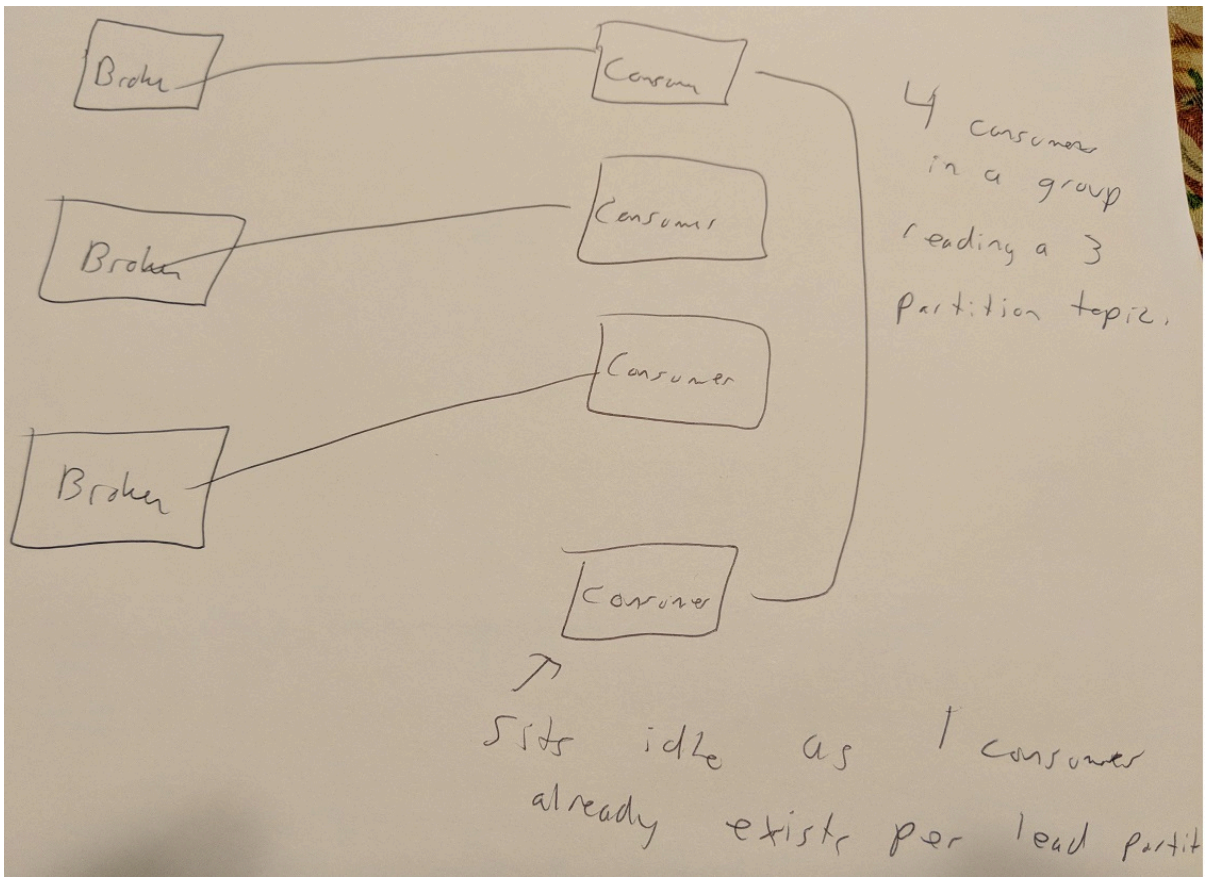

Figure 5.4 Idle Kafka Consumer

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, we how want our

consumers to interact with a different REST endpoint than we do today due to vendor changes. We already saw the infinite loop of processing that we had just used Ctrl-C to end our processing. However, the proper way includes calling close. The group coordinator should be sent notification about membership of the group being changed due to the client leaving.

Listing 5.2 Stopping a Consumer

```
public class KafkaConsumerThread implements Runnable {
    private final AtomicBoolean stopping = new AtomicBoolean(false);
    ...

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

    public void run() {
        try {
            consumer.subscribe(Arrays.asList("webclicks"));

            while (!stopping.get()) {                          (1)
                ConsumerRecords<String, String> records = consumer.poll(100);
                ...
            }

        } catch (WakeupException e) {                          (2)
            if (!stopping.get()) throw e;
        } finally {

            consumer.close();                                  (3)
        }
    }


    public void shutdown() {                                   (4)
        stopping.set(true);
        consumer.wakeup();
    }
}
```

(1) The stopping variable will be our guard condition on if we keep processing or not

(2) The WakupException is what is trigger by the client shutdown hook

(3) Close properly stops the client and informs the broker

(4) This method can be called from a different thread in order to stop the client in the proper way

If your process does not shutdown correctly, Kafka still tries to handle being informed by a down consumer by using a heartbeat. If the coordinator does not receive a heartbeat within a specific time, then the coordinator considers the client as dead and reassigns the partitions to the existing consumer clients. This heartbeat timeout is set by using the property `session.timeout.ms`.

If you run across older documentation for Kafka, you might notice consumer client

configuration in regards to Zookeeper. However, with the new consumer client, Kafka does not have consumers rely directly on Zookeeper. In fact, consumers used to store the offsets that they had consumed to at that point to Zookeeper. However, now the offsets are stored inside a Kafka internal topic. As a side note, consumer clients do not have to store their offsets in either or these locations, but will likely be the case.

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.

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 is determined by group.id.

The importance of Listing 5.3 is that it shows you the important property `group.id`. If you make 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. 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.

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 an ad tracking event stream. One area is wondering about the number of clicks to collect the monetary impact of a campaign and the analytics side is more interested in the data for the impact on click rates after an ad promotion. So would anyone on the Ad team care about what the analytics team was doing or want to have each other consume only a portion of the messages that they each wanted to see? 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 really have not talked too much about how we keep track of that 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 subscriptions will have missed this message entirely since they were not actively part of that receiver list.

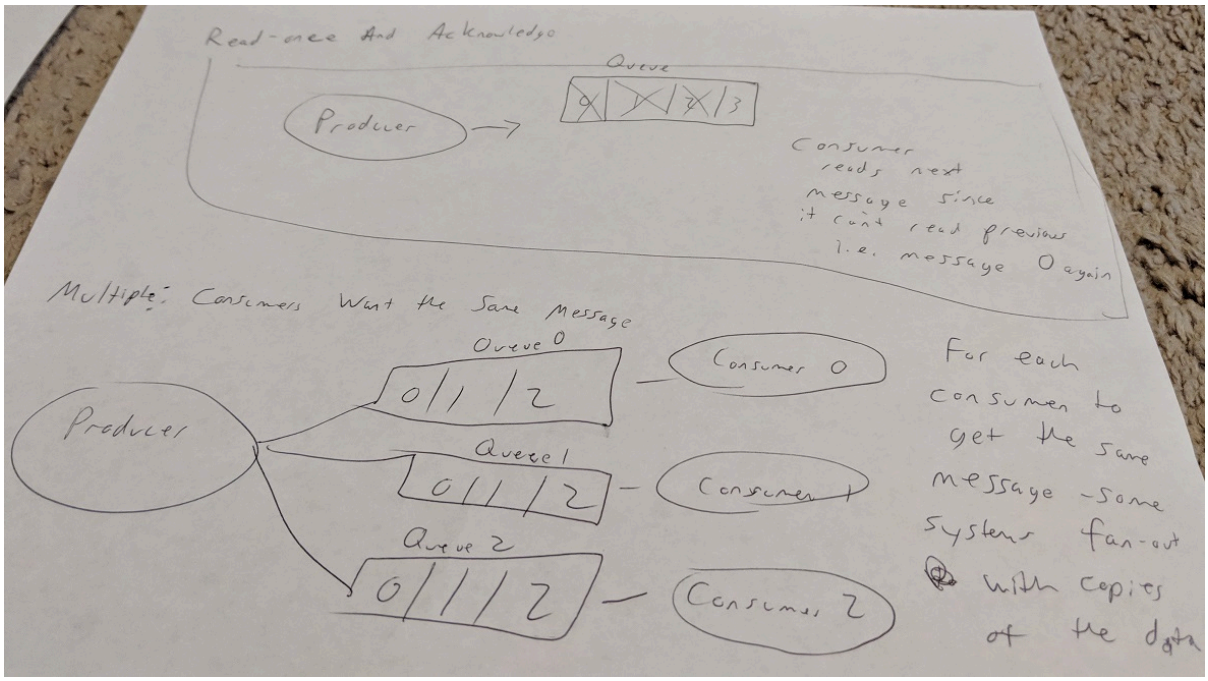

Figure 5.5 Other Broker Scenarios

You might also be used to patterns 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. 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 leader partition.

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 maybe 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 to letting your consumer clients work together is a unique combination of the following: group, topic, and partition number.

5.4.1 GroupCoordinator

There is usually one broker that takes over the important jobs of working with offset commits for a specific group. The offset commit can be though of the last message consumed by that group. These duties can be thought of as the a GroupCoordinator and offset manager. The group coordinator works with the consumer clients to keep track of where that specific group has read from the topic. These coordinates of partition of a topic and group id make it specific to an offset value.

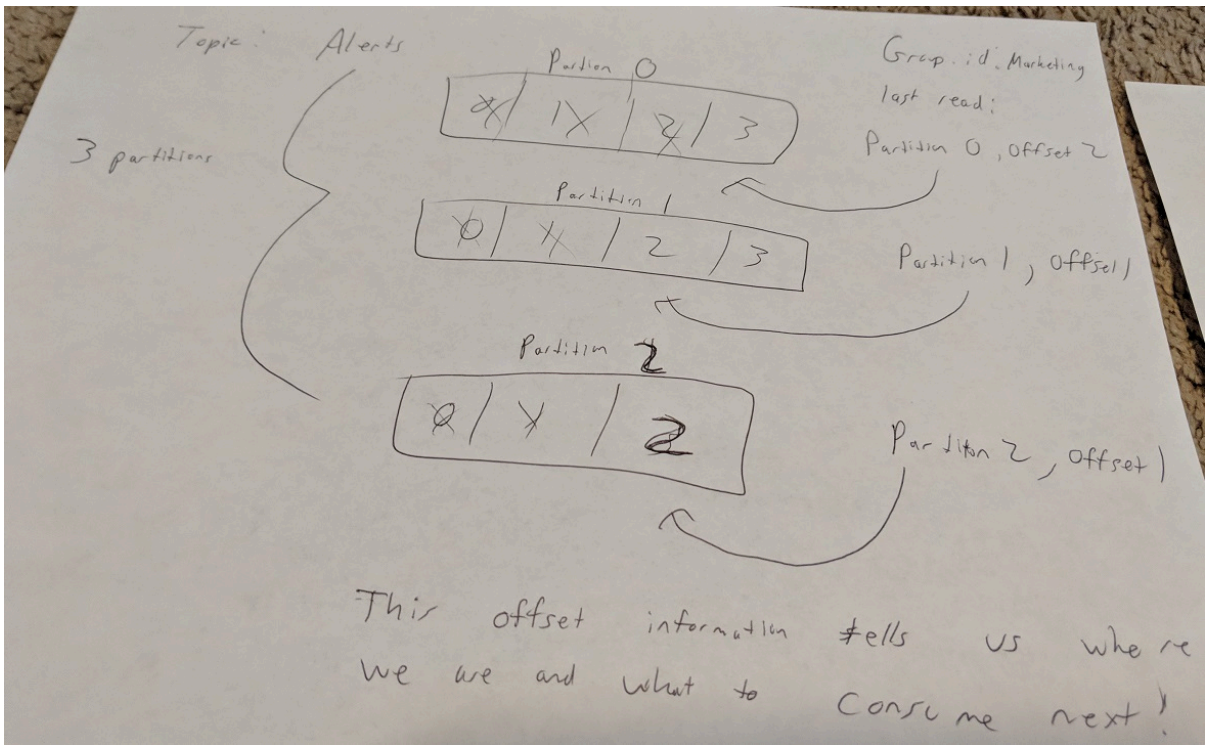

Figure 5.6 Coordinates

Looking at Figure 5.6, notice that we can use the offset commits as coordinates to figure out where to read from next.

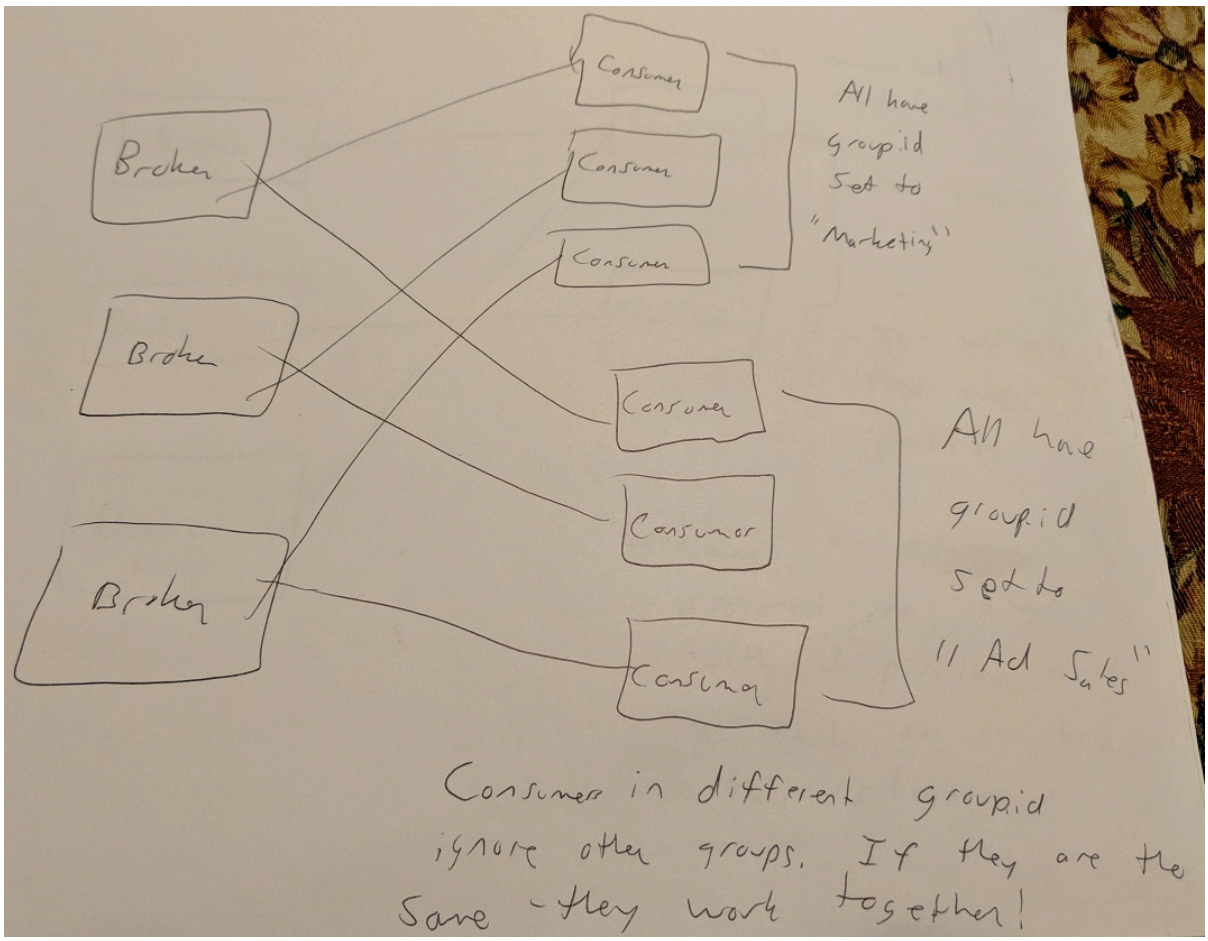

Figure 5.7 Consumers in Separate Groups

As a general rule, one partition is consumed by only one consumer for any given consumer group. In other words, while a partitions might be read by many consumers, it will only be read by at most one consumer from each group at a point in time.

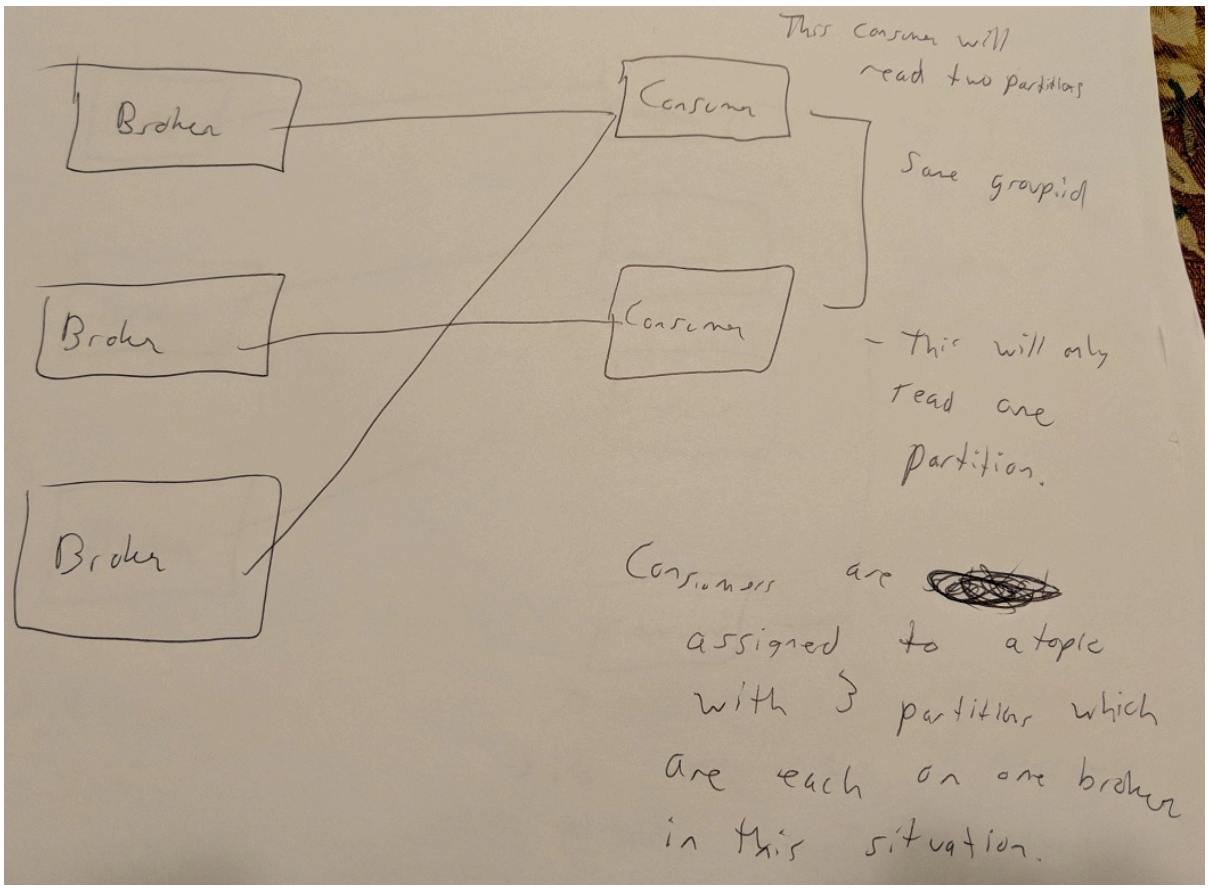

Figure 5.8 Kafka Consumers in a Group

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

One way a consumer can drop out of a group is by failure to send a heartbeat to the GroupCoordinator. Failure to send these heartbeats can happen in a couple of ways. One easy way for this is to stop the consumer client by either termination of the process or failure due to a fatal exception. Another common issue is the time that you client code could be using to process the last batch of messages before another poll loop. `max.poll.interval.ms` was introduced as part of Kafka 0.10.1. This setting is specific go how long it takes for your messages processing. If you take longer than your setting (say you go over your 1 minute setting), the consumer will be considered dead and drop out of the group.

Consumer groups also help in the instances of consumers being added. Take an example of one consumer client reading from a topic made up of 2 partitions. Adding a second consumer client should how have each consumer handle and process one partiiton each.

One major exception is when there are more consumers than number of partitions. If more consumers are present, then those consumers without assignments will be sitting idle. One reason to have idle consumers standing by is to make sure a consumer is ready to handle a partition in case of a failure.

One thing to keep in mind is that during a rebalance, consumption is paused. So if you have a requirement that would make you worry about timely processing, it is important that you make sure you do not have rebalances occurring all of the time. Adding and losing consumer from a group are all key details to watch and make sure that your leaders are not being rebalanced across clients and repeatly causing pauses in your consumption.

5.4.2 ConsumerRebalanceListener

Besides looking at log files or printing offset metadata from consumer clients, another way that your code can keep informed of consumer rebalances is by implementing the interface ConsumerRebalanceListener. You provide your code when you subscribe in your code. Actions can be invoked when partitions are revoked (unassigned) as well as when they are assigned to the consumer.

All consumer processes will call the onPartitionsRevoked methods before calling any onPartitionsAssigned code. This is key to doing cleanup or committing any state that you need before any partition assignments are being made since you can not usually be sure that your client would handle the same partitions after the rebalance that they did before.

Listing 5.4 Manual Partition Assign

```
public class MaintainOffsetsOnRebalance
implements ConsumerRebalanceListener {                                    (1)
    ...

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {   (2)
        for(TopicPartition partition: partitions)
           saveOffsetInStorage(consumer.position(partition));
    }

    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {  (3)
        for(TopicPartition partition: partitions)
           consumer.seek(partition, readOffsetFromStorage(partition));
    }
}
```

(1) The ConsumerRebalanceListener interface is needed to add our custom logic

(2) onPartitionsRevoked takes the topic and partition information in order to save our commits to a custom storage platform

(3)

- When the client is assigned to a specific topic and partition, we read the offset we stored previous in order to start reading where we left off

Listing 5.4 shows implementing our own custom ConsumerRebalanceListener. In this case, we are using our logic due to the fact that we are wanting to store our offsets to our own storage. Instead of letting Kafka take care of it by defauly to its internal topics, we are using our own logic to meet our specific storage requirements.

5.4.3 Partition Assignment Strategy

The property `partition.assignment.strategy` is what determines which partitions are assigned to each consumer in the group. 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.

- 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 will get the remaining partitions. Note, if you have a group and partition numbers that often get assigned to the same lexicographical order, you might see some consumer taking a larger number of partitions over time.
- 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.

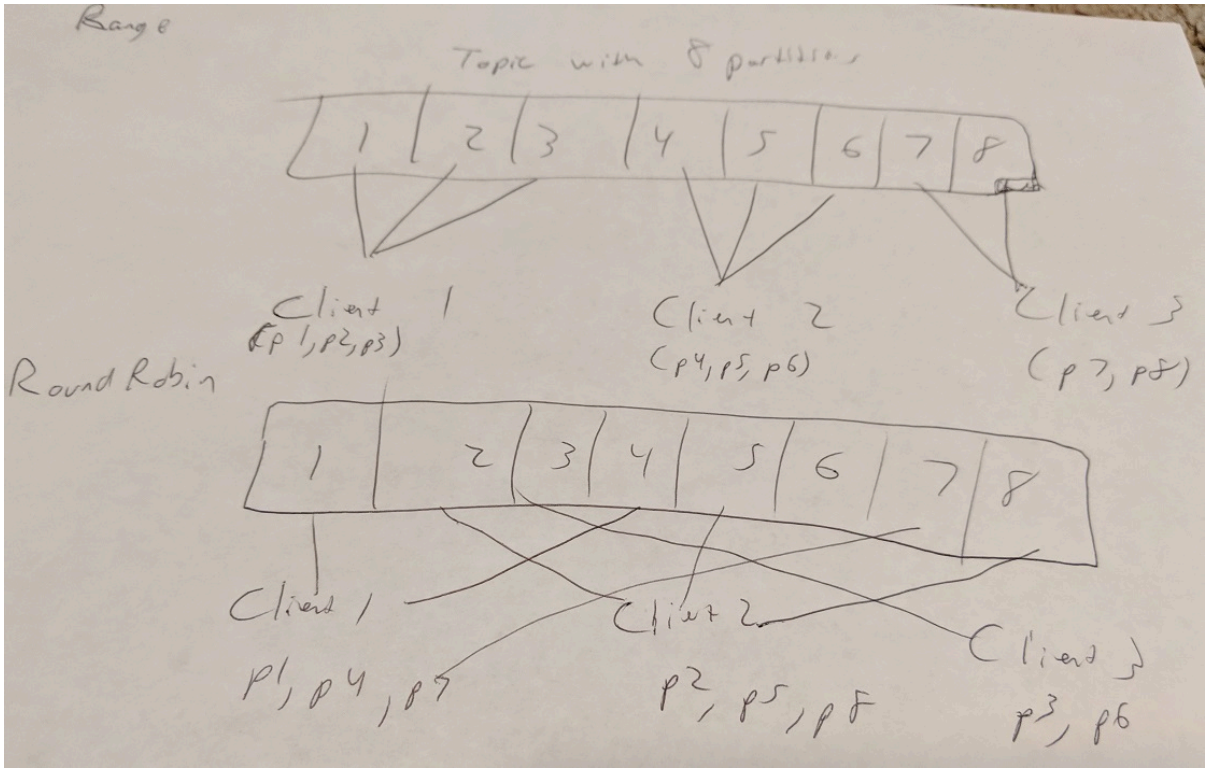

Figure 5.9 Partition Assignment

- Sticky: This is a newer strategy that was added in version 0.11.0. One of the major reasons that it came about is to not only try and distribute topic partitions as evenly as possible, but also to have partitions stay with their existing consumers if possible. Allowing partition assignments to stay with the same consumers could help speed up a rebalance which could occur during any consumer group changes.

Of course, you can also create your own strategy. Currently, `AbstractPartitionAssignor` is an abstract class that does some of the common work with mining data about topics to help implement your logic.

5.4.4 Standalone Consumer

When would it make sense to just use one consumer? Sometimes it might be more straightforward and avoid the logic of group changes. The important factor that you would need to be aware of is that by avoiding consumer groups, you will need to know which specific partitions you are going to be consuming. With this simpler method, you will have to do more of your own code if you really think that your topic might change. One of the more obvious changes is if you added another partition to your topic. It would be up to your code to deal with this change and ability to consume from this new item. Standalone consumer can be used when you do not simple have that much data or you only have one partition anyway.

5.4.5 Manual Partition Assignment

If the assignment of partitions is important to your applications, then you will want to look at how to assign specific partitions. As part of this manual assignment, you can not mix using assign with subscribe.

Listing 5.5 Manual Partition Assign

```
String topic = "webclicks";
TopicPartition partition0 = new TopicPartition(topic, 0);
TopicPartition partition1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(partition0, partition1));    (1)
```

(1) You can use the assign method in order to have the consumer client retrieve messages from specific topic and partition combinations.

Due to this assignment decision, it is important to not that each consumer acts on its own. Sharing a group.id with another consumer does not give you any benefits. 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.

5.5 Auto or Manual Commit of Offsets

As with many parts of a system, one of the important things to think about is the tradeoffs you need to make for making sure 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 ans 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`. This means that offsets are committed automatically for you at a frequency controlled by `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-most-once delivery is one pattern that is possible with automatic commits. But what sort of trouble could we get into? If we are processing messages that we got from our last poll, the automatic commit offset could be marked as being read even as we 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. As far as our next poll, we would 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.

Let's talk about using manual commits enabled by: `enable.auto.commit=false`. This method can be used to exercise the most control over when you logic has actually consumed a message. At-least-once delivery guarantees is a pattern that can be easily achieved in 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 of offset 100. During processing, you fail with an exception and the consumer stops. Since the code never actually committed that the offset 100 was read, 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 offset. 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 need logic to deal with 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 5.6 Synchronous Commit

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s",
        record.offset(), record.key(), record.value());
        consumer.commitSync();   (1)
    }
}
```

(1) The commitSync method is a blocking call that will wait until the commit is successful or failed.

Looking at the example Listing 5.6 for commitSync, it is important to note that the commit is taking place in a manner than will block until a success or failure.

Listing 5.7 Asynchronous Commit

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s", record.offset(),
            record.key(), record.value());
        consumer.commitAsync(callback);   (1)
    }
}
```

(1) The commitAsync method is a non-blocking call that allow the logic to continue onto the next iteration

CommitAsync is our path to manually commit without blocking our next iteration. One of the options that you can send in this method call is the ability to send in a callback. If you think back to Chapter 4, this is similar to how we used asynchronous sends with a callback as well. 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 5.8 Asynchronous Commit with Callback

```
public void commitOffset(long offset) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(offset + 1, "");

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


    OffsetCommitCallback callback = new OffsetCommitCallback() {   (1)
        @Override
```

```
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> map,
            Exception e) {                                                    ❷
            if (e != null) {
                LOG.warn(String.format("Commit failed: offset %d, Topic %s", offset,
                    statTopic));
            }
            else {
                logger.info(String.format("OK. offset %d, Topic %s", record.offset(),
                    record.topic()));
            }
        }
    };
    consumer.commitAsync(offsetMap, callback);
}
```

❶ Creating a OffsetCommitCallback instance

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

Listing 5.8 shows how to create a OffsetCommitCallback. This instance allows us to have log messages to determine out success or failure even though our code is not waiting for a response before moving on to the next instruction.

Why would you want to choose synchronous or asynchronous commit patterns? For one, we need to keep in mind that our latency will be higher if we are waiting for a blocking call. This time factor might be worth it with your requirements include needs for data consistency. What does consistency mean in this instance? It really is a focus on how your commit

5.6 Reading From a Compacted Topic

When we talked about not needing a history of messages, but rather just the last value, we look at the concepts of how an immutable log deals with this. The log is compacted by Kafka on a background process. Same key names can be removed except for the last one. One gotcha on read, consumers might still 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.

Have you noticed one of the topics that is internal to Kafka that uses compaction? The topics that hold the consumer clients offsets are titled: __consumer_offsets. Compaction makes sense here since for a specific 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 the key, it is possible to seek to a specific offset. Thinking about our log of messages being an ever increasing array with each one having an index, we have a couple of options including: starting from the beginning, going to the end, starting at a given offset, or 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 or a failure in our data pipeline flow after Kafka. The important thing to note is the `auto.offset.reset` is set to earliest. However, this property really only matters even Kafka can not find an offset that matches its current offset. In this case, one can 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.

Listing 5.9 Earliest Offset

```
Properties props = new Properties();
props.put("group.id", UUID.randomUUID().toString());          (1)
props.put("auto.offset.reset", "earliest");                   (2)
```

(1) Create a group.id that Kafka would not have a stored offset for

(2) Earliest is the policy we want for starting at the earliest offset we still have retained in our logs

Listing 5.9 is an example of setting the property `auto.offset.reset` to 'earliest'. However, the Java client consumer also has the following method available: seekToBeginning. This is another way to achieve the same as the code in Listing 5.9.

5.7.2 Going to the end

Sometimes you really just want to start your logic from when the consumers start-up and forget about the past messages. Maybe the data is already too late to have business value in your topic.

Listing 5.10 Latest Offset

```
Properties props = new Properties();
props.put("group.id", UUID.randomUUID().toString());          (1)
props.put("auto.offset.reset", "latest");                     (2)
```

1. Create a group.id that Kafka would not have a stored offset for
2. Latest is the policy we want starting with latest messages and moving forward

The figure shows the properties you would set to get this behavior. We want to make sure that we don't find a consumer offset from before and will instead default to the latest offset Kafka has for our subscriptions.

The Java client consumer also has the following method available: seekToEnd. This is another way to achieve the same as above.

5.7.3 Seek to an Offset

Kafka does give 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 is needed as the coordinates for giving your consumer the information it needs to set that offset.

Listing 5.11 Seek Offset

```
consumer.seek(topicPartition, offset);
```

5.7.4 Offsets For Times

One of the trickier offset search is for `offsetsForTimes`. This method allows you to send a map of topic and partitions as well as a timestamp for each in order to get a map back of the offset and timestamp of those topic and partitions given. This can be used in situations where a logical offset it 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.

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 producer resending messages on failures or variations in when timestamps are added (by consumers, prehaps), this timestamp is not an official order. Offsets are ordered in the log as they have always been and timestamps do not have this ordering.

Listing 5.12 Seek Offset

```
Map<TopicPartition, OffsetAndTimestamp> offsetMap =
kafkaConsumer.offsetsForTimes(timeStampMapper);          (1)
```

```
...
// We need to use the map we get
consumer.seek(topicPartition, offsetMap.offset());  ❷
```

❶ We are using timestamps to find the first offset greater or equal to that time

❷ We can use the seek method once we have the information we need from the above call

As Listing 5.12 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, we then can seek directly to the offset we are interested in.

5.8 Reading Concerns

One thing that we have talked about is that the consumer is in the drivers 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 data the server should return per a request. One gotcha to note is that the consumer might make parallel fetch calls at the same time from different partitions.

5.8.1 Broker use of Consumers

If you think about how many replicas of a partition you have, you will notice that the data of a leader partition has to be copied to other brokers. For this replication, Kafka brokers use fetch requests like the consumer client we have used so far! It is interesting to see how understanding the usage of parts of Kafka explain 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.

5.9 Summary

In this chapter we learned:

- Why offsets are needed and how to use them to get to the data we need
- How consumer groups and ids impact how we process data
- To analysis tradeoffs on synchronous or asynchronous commit and read patterns

B Installation

Despite being a complex feature set, Kafka is generally pretty easy to install. There are just a couple of things that you need in order to get started.

> **TIP** **Run on more than one machine for production!**
>
> Our focus on these setup steps is to get up and running with ease. Both ZooKeeper and Kafka should be ran on more than one physical server in a production environment if you want fault-tolerance. However, we want to make sure that we can focus on learning Kafka and not on managing multiple servers at this point. The main chapters will cover what we might consider for production settings.

B.1 Which Operating System to use

One question might be what operating system (OS) do I need to use? While Kafka can usually run wherever Java can, it is nice to use something production like in our testing. Linux is the most likely home for Kafka in production and seems to be where many user support forums will continue to focus their questions and answers in that context. The examples in this book will take place on a Linux OS.

B.2 Installing Prerequisite: Java

For the examples in this book we will be using JDK 1.8. The latest version is fine but I would suggest something later than u5. That version has the G1 collector.

B.3 Installing Prerequisite: ZooKeeper

Even with the reduced dependency on ZooKeeper from the client side in recent versions, at this time, Kafka will need a running installation of ZooKeeper to work. At the time of this writing, the current stable version is 3.4. ZooKeeper itself can be an interesting and complex topic, but since it is not the main focus to get us up and running, we can leverage the scripts that are included in Kafka already. This will help us create a single-node instance.

B.4 Installing Kafka

At the time of this book's creation, Kafka 0.11.0 is the latest and is used in our examples. The Apache project has mirrors that it maintains and you can search for the version to download in that way (see: kafka.apache.org/downloads). When you look at the actual binary name, it might seem a little confusing at first. For example, kafka_2.11-0.11.0.0.tgz means the Kafka version is 0.11.0 (the information after the hyphen) that uses Scala 2.11. There seemed to be issues with compatibility between Scala versions that led to versions being part of the package name but this should not be a concern for your new install!

To get the most out of the examples while still making things easy to get started, we are going to have a 3 node cluster on a single machine. This is not a recommended strategy for Production, but it will allow us to understand key concepts without the overhead of spending a lot of time on setup.

NOTE **Windows script location**

For Windows users, you will find bat scripts under the bin/windows folder that have the same names as the shell scripts used in the examples below.

First, you need to unpack the binary and locate the bin directory.

Listing B.1 Unpacking the Kafka binary

```
tar -xzf kafka_2.11-0.11.0.0.tgz
cd kafka_2.11-0.11.0.0
```

For our examples, we are going to use a single local ZooKeeper server. This is not how you would run in a Production type setting - but will serve our purpose for keeping up and going. The main chapters will cover what we might consider for production settings for ZooKeeper as well as Kafka.

Listing B.2 Starting ZooKeeper

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

NOTE **Broker unique ids**

Each Kafka broker will run on its own port and use a separate log directory. It is also important that each configuration file has a unique id for each broker. You will usually see your broker ids start at 0 following traditional computer science counting. It also is how the partitions are numbered so it might make sense to make it consistent.

Next, we are going to set the foundation and create the 3 message brokers that will be our Kafka cluster. We need to create 3 separate configuration files and change the following information.

Listing B.3 Creating Multiple Kafka Broker Configs

```
cp config/server.properties config/server0.properties      (1)
cp config/server.properties config/server1.properties
cp config/server.properties config/server2.properties

# Edit each file above to have the following changed properties respectively
vi config/server.properties config/server0.properties      (2)
broker.id=0
listeners=PLAINTEXT://localhost:9092
log.dir=/tmp/kafka-logs-0

vi config/server.properties config/server1.properties      (3)
broker.id=1
listeners=PLAINTEXT://localhost:9093
log.dir=/tmp/kafka-logs-1

vi config/server.properties config/server2.properties      (4)
broker.id=2
listeners=PLAINTEXT://localhost:9094
log.dir=/tmp/kafka-logs-2
```

(1) Make three copies of the default server properties files.

(2) Update id, port, and log directory for broker id 0

(3) Update id, port, and log directory for broker id 1

(4) Update id, port, and log directory for broker id 2

After this, we can attempt to start up each broker!

Listing B.4 Starting Kafka - Run the following in their own console

```
bin/kafka-server-start.sh config/server0.properties      (1)
bin/kafka-server-start.sh config/server1.properties
bin/kafka-server-start.sh config/server2.properties
```

(1) Command to start up each broker from the command line

It usually works best if you start each broker in a separate terminal tab or window and leave them running. While you can background the process, you might see some confusing output scroll through the terminal if you are just using one!

TIP **Java process information**

If you close a terminal or have a hanging process, do not forgot about running the `jps` command. You might need to use `sudo`. That command will help you find the java processes you might need to kill. Below is an example on my machine where you can see the PIDs of the brokers as well as ZooKepper (QuorumPeerMain).

Listing B.5 jps output for Zookeeper (QuorumPeerMain) and 3 Brokers

```
2532 Kafka              (1)
2745 Kafka              (1)
2318 Kafka              (1)
2085 QuorumPeerMain     (2)
```

(1) The three Kafka jvm process label and id for each broker

(2) ZooKeeper jvm process label and id

B.5 Confluent CLI

If you are ok with using more than the base OSS version of Apache Kafka and are interested in the Confluent Platform, Confluent has command line tools to easily start and manage their Confluent Platform from the command line. Their Github README.md has more details on the convience script usage.