# Kafka IN ACTION

Dylan D. Scott

MANNING

**MEAP Edition**
**Manning Early Access Program**
# Kafka in Action
**Version 12**

# *welcome*

Thank you for purchasing the MEAP for Kafka in Action!

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

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

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

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

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

Thanks again for your interest and for purchasing the MEAP!

—Dylan Scott

# *brief contents*

# 1 Introduction to Kafka

***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 that 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.

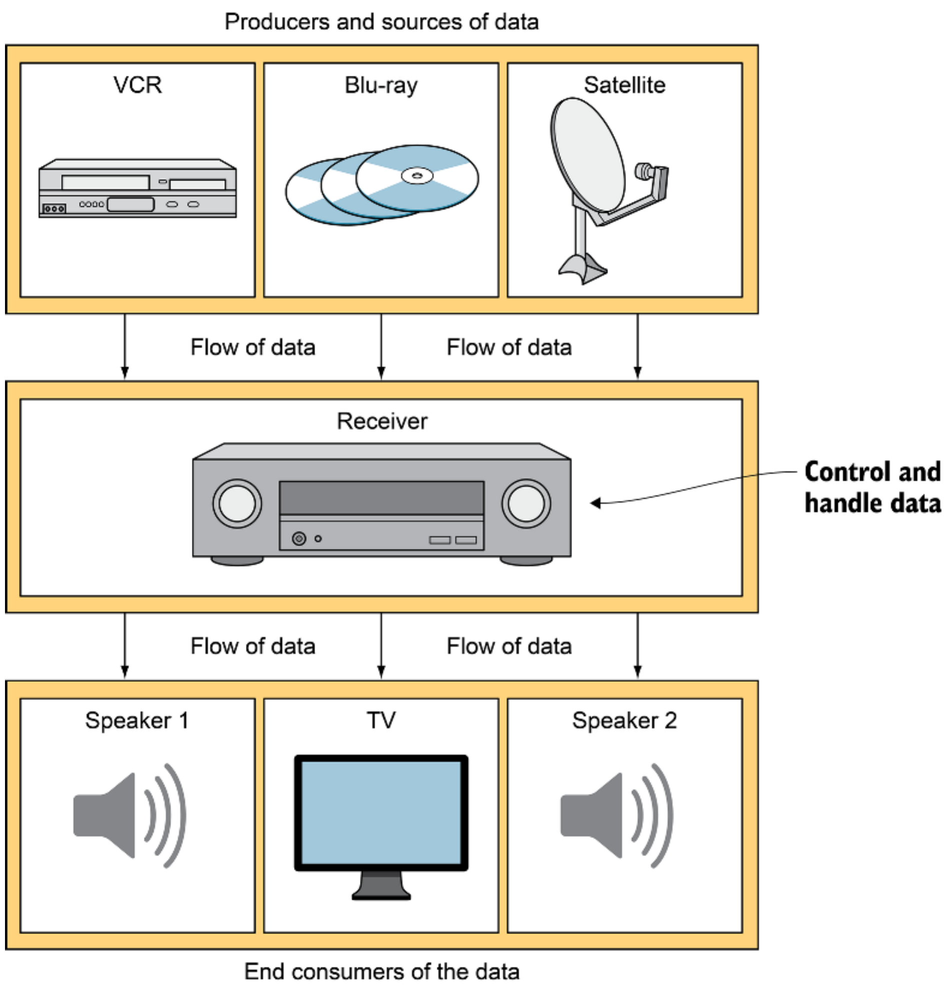


**Figure 1.1 Receiver overview**

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.

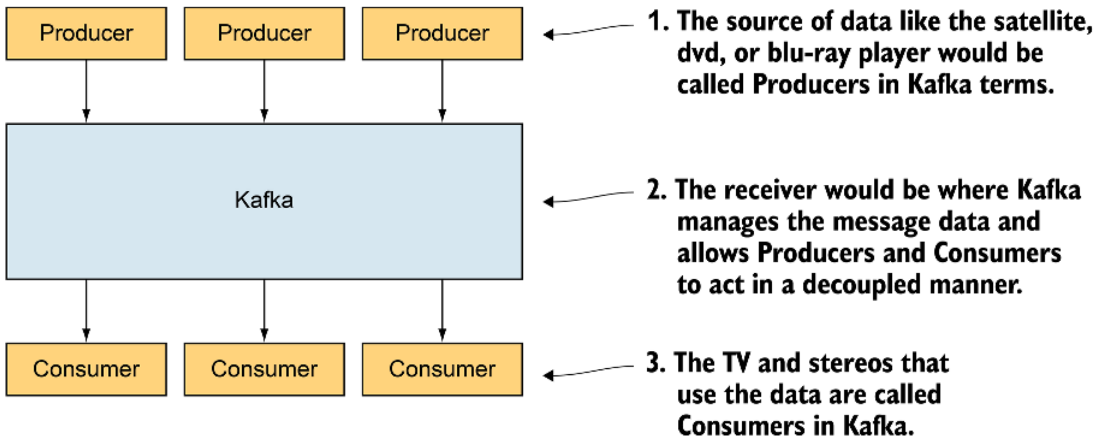

Producer
Producer
Producer
1. The source of data like the satellite, dvd, or blu-ray player would be called Producers in Kafka terms.
Kafka
2. The receiver would be where Kafka manages the message data and allows Producers and Consumers to act in a decoupled manner.
Consumer
Consumer
Consumer
3. The TV and stereos that use the data are called Consumers in Kafka.

**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.

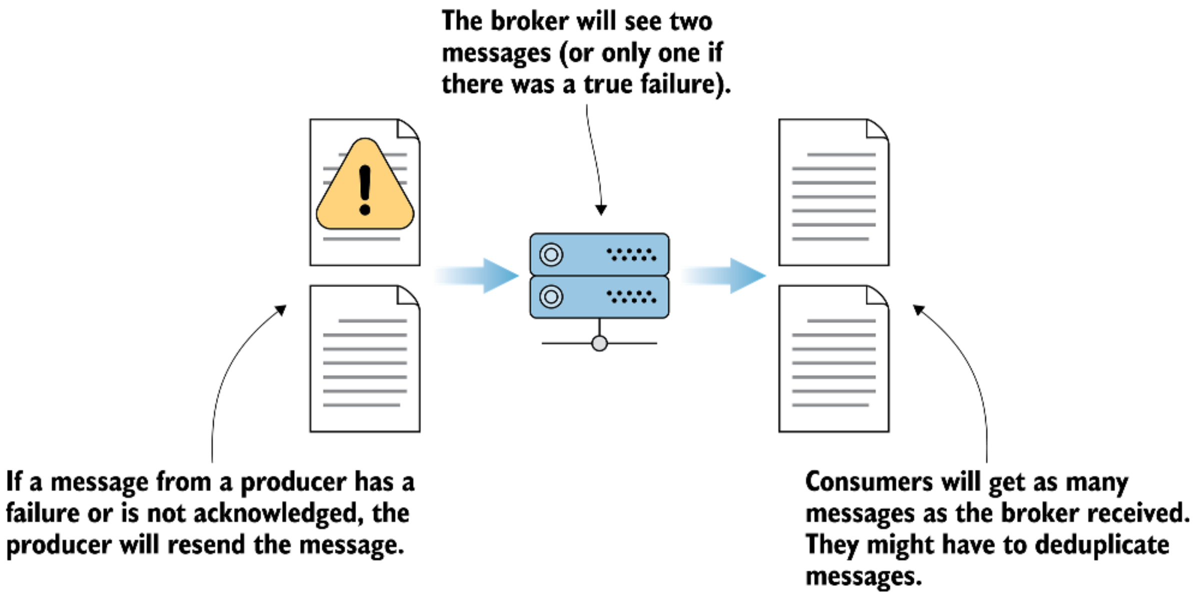

The broker will see two messages (or only one if there was a true failure).
If a message from a producer has a failure or is not acknowledged, the producer will resend the message.
Consumers will get as many messages as the broker received. They might have to deduplicate messages.

**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.

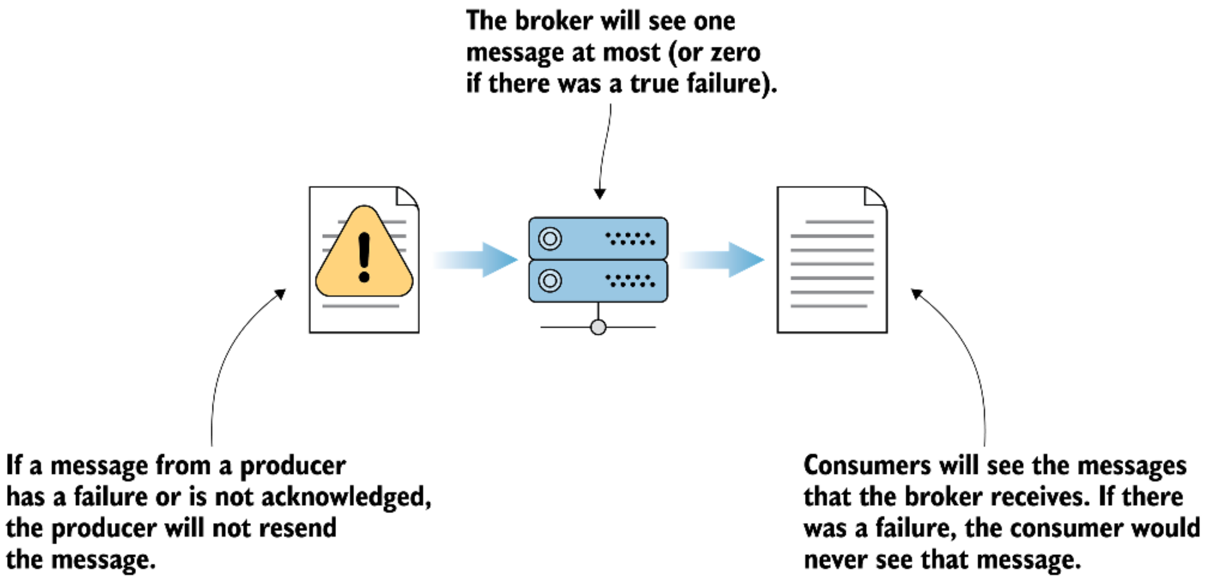

The broker will see one message at most (or zero if there was a true failure).
If a message from a producer has a failure or is not acknowledged, the producer will not resend the message.
Consumers will see the messages that the broker receives. If there was a failure, the consumer would never see that message.

**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.

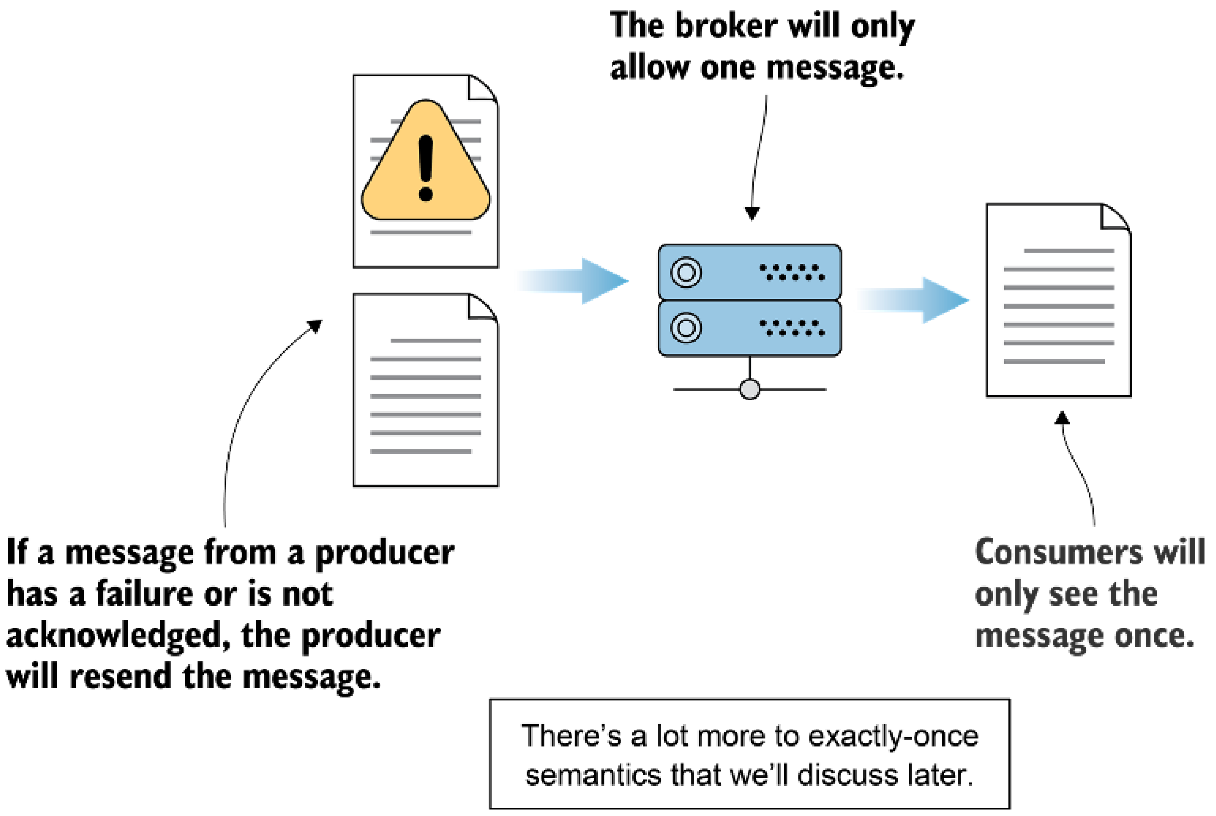

The broker will only allow one message.
If a message from a producer has a failure or is not acknowledged, the producer will resend the message.
Consumers will only see the message once.
There's a lot more to exactly-once semantics that we'll discuss later.

**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.

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.

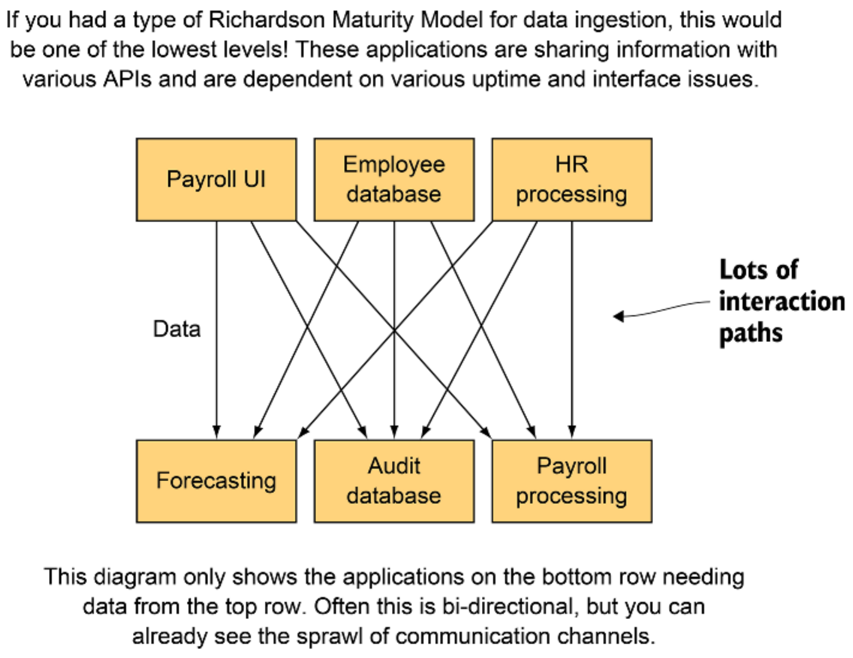

If you had a type of Richardson Maturity Model for data ingestion, this would be one of the lowest levels! These applications are sharing information with various APIs and are dependent on various uptime and interface issues.
Payroll UI
Employee database
HR processing
Data
Lots of interaction paths
Forecasting
Audit database
Payroll processing
This diagram only shows the applications on the bottom row needing data from the top row. Often this is bi-directional, but you can already see the sprawl of communication channels.

**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.

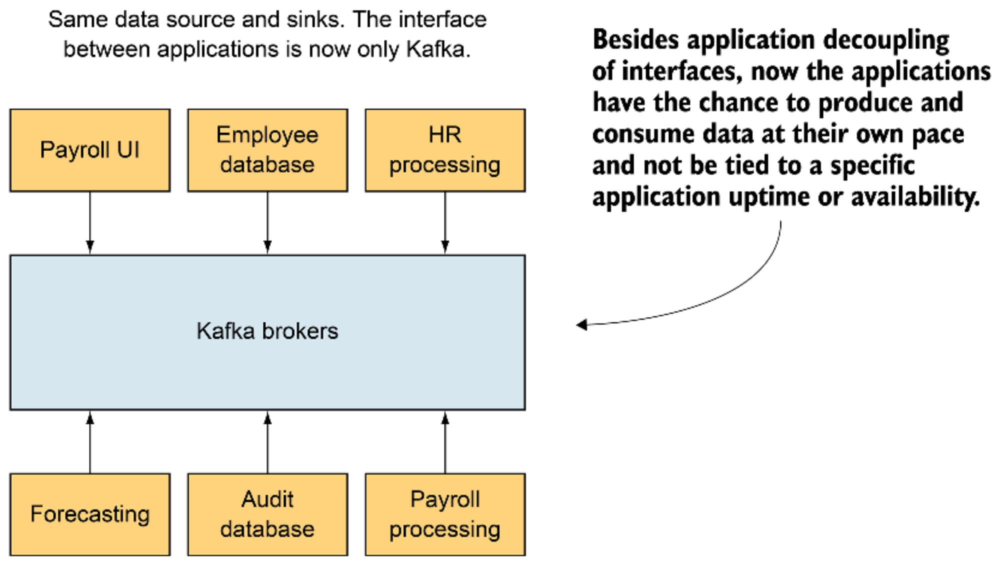

Same data source and sinks. The interface between applications is now only Kafka.
Besides application decoupling of interfaces, now the applications have the chance to produce and consume data at their own pace and not be tied to a specific application uptime or availability.
Payroll UI
Employee database
HR processing
Kafka brokers
Forecasting
Audit database
Payroll processing

**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 ( apache-kafka-purgatory-hierarchical-timing-whe els. 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

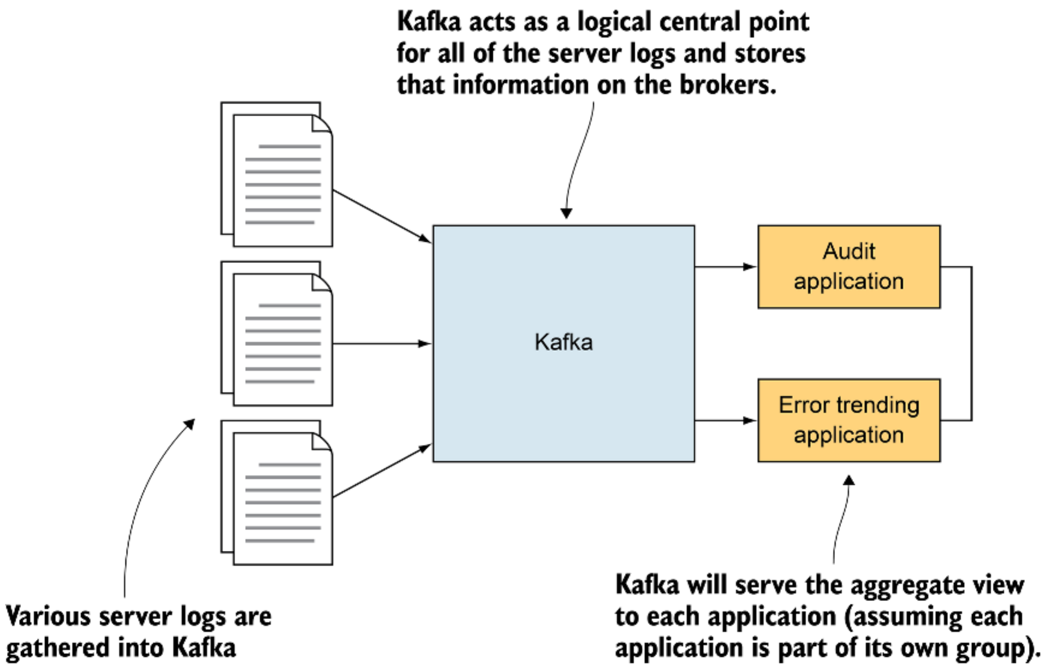

Kafka acts as a logical central point for all of the server logs and stores that information on the brokers.
Kafka
Audit application
Error trending application
Various server logs are gathered into Kafka
Kafka will serve the aggregate view to each application (assuming each application is part of its own group).

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

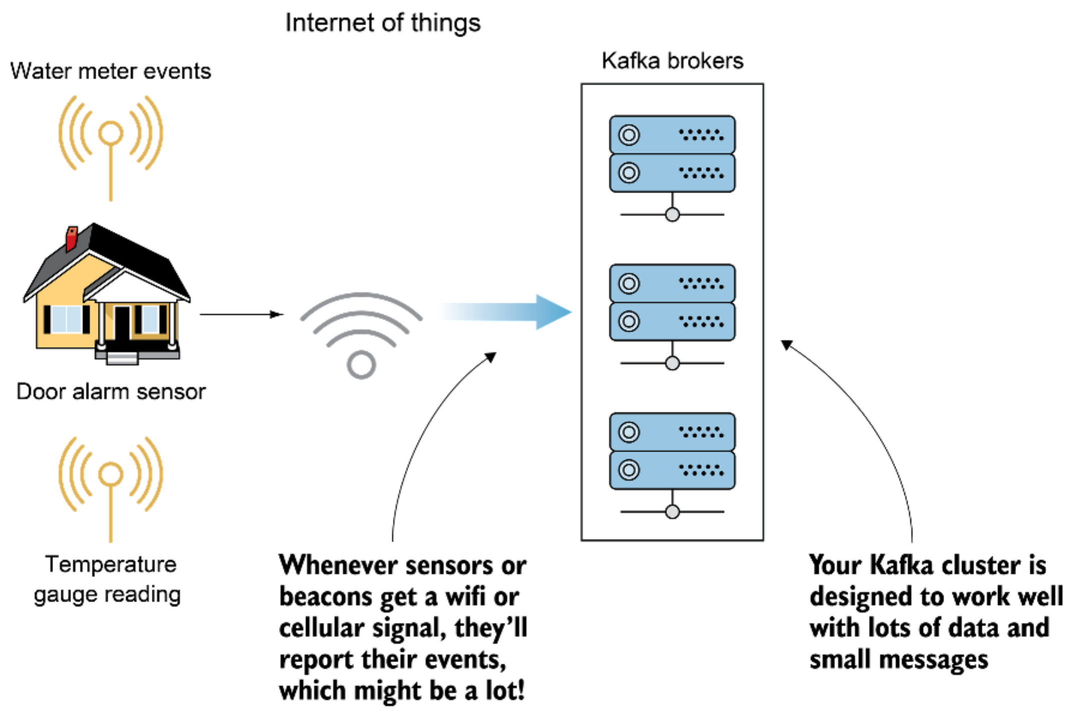

Internet of things
Water meter events
Door alarm sensor
Temperature gauge reading
Kafka brokers
Whenever sensors or beacons get a wifi or cellular signal, they'll report their events, which might be a lot!
Your Kafka cluster is designed to work well with lots of data and small messages

**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/resource s). 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.

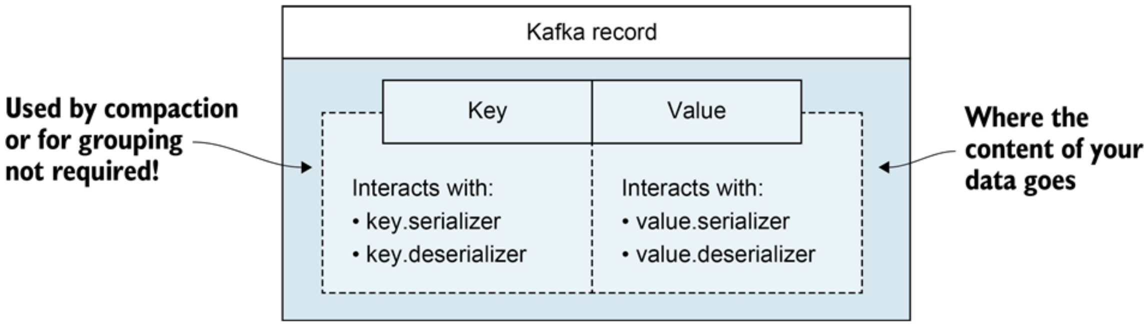


**Figure 2.1 Kafka messages are made up a key and value**

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    ❶

Topic:helloworld    PartitionCount:3    ReplicationFactor:3    Configs:            ❷
    Topic: helloworld    Partition: 0  ❸ Leader: 0  ❹ Replicas: 0,1,2  ❺ Isr: 0,1,2  ❻
    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
```

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

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

❸ Each partition in tht topic is shown. This is for partition labeled 0

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

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

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

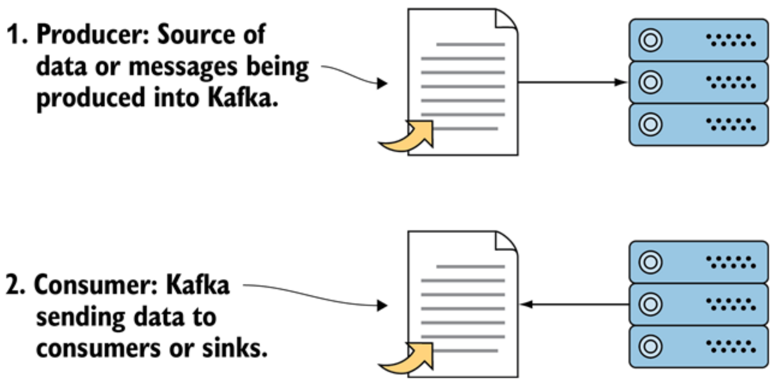

1. Producer: Source of data or messages being produced into Kafka.
2. Consumer: Kafka sending data to consumers or sinks.

**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.

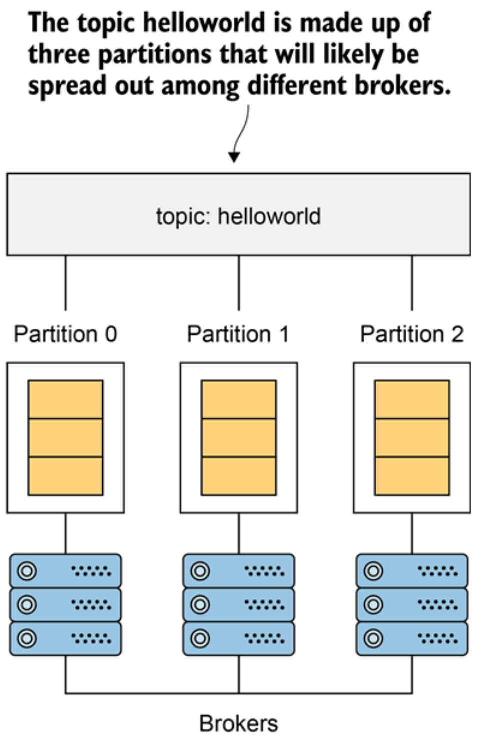

The topic helloworld is made up of three partitions that will likely be spread out among different brokers.
topic: helloworld
Partition 0
Partition 1
Partition 2
Brokers

**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.

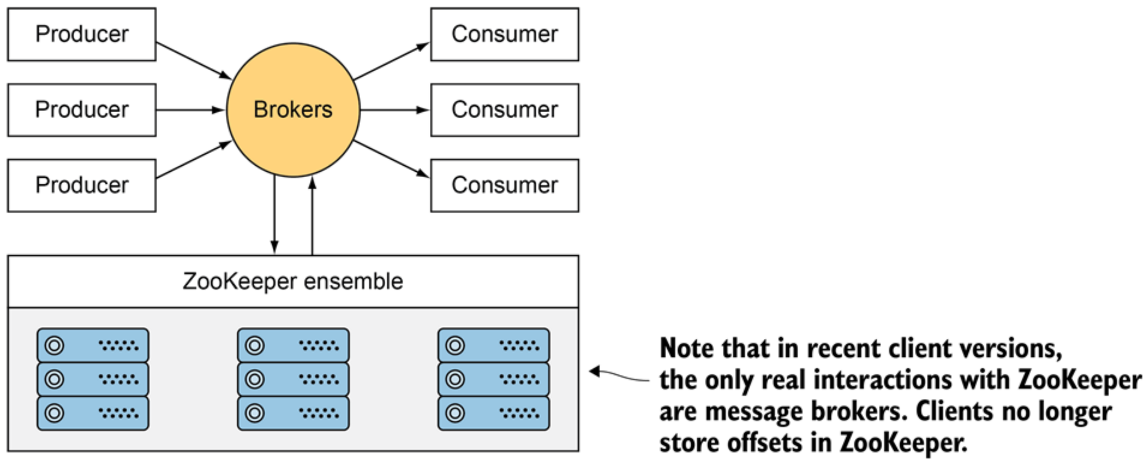

Producer
Producer
Producer
Brokers
Consumer
Consumer
Consumer
ZooKeeper ensemble
Note that in recent client versions, the only real interactions with ZooKeeper are message brokers. Clients no longer store offsets in ZooKeeper.

**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,     ❷
    new Callback() {                ❸
        @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;
} catch (RetriableException e) {  ❹
  log.warn("{} Failed to send {}, backing off before retrying:", this, producerRecord, e);
  toSend = toSend.subList(processed, toSend.size());
  lastSendFailed = true;
  counter.retryRemaining();
  return false;
} catch (KafkaException e) {      ❹
 throw new ConnectException("Unrecoverable exception trying to send", e);
}
processed++;
}
toSend = null;
return true;
}
```

❶ The ProducerRecord will be what holds each message being sent into Kafka

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

❸ 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

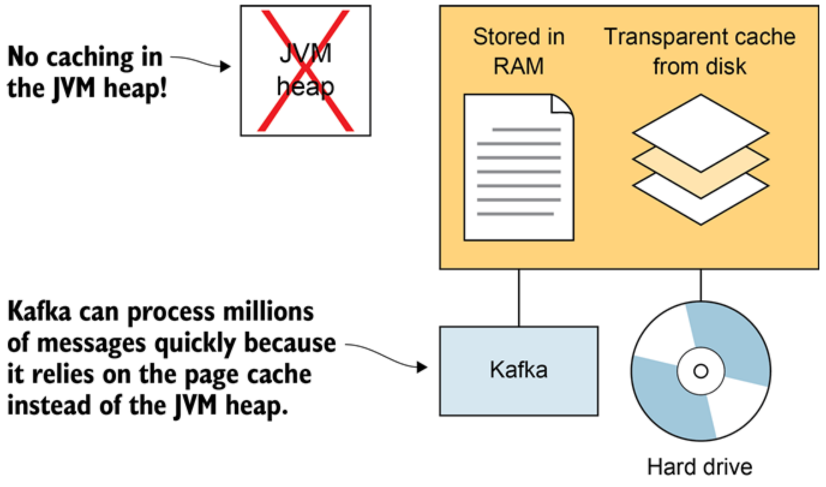

No caching in the JVM heap!
JVM heap
Stored in RAM
Transparent cache from disk
Kafka can process millions of messages quickly because it relies on the page cache instead of the JVM heap.
Kafka
Hard drive

**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

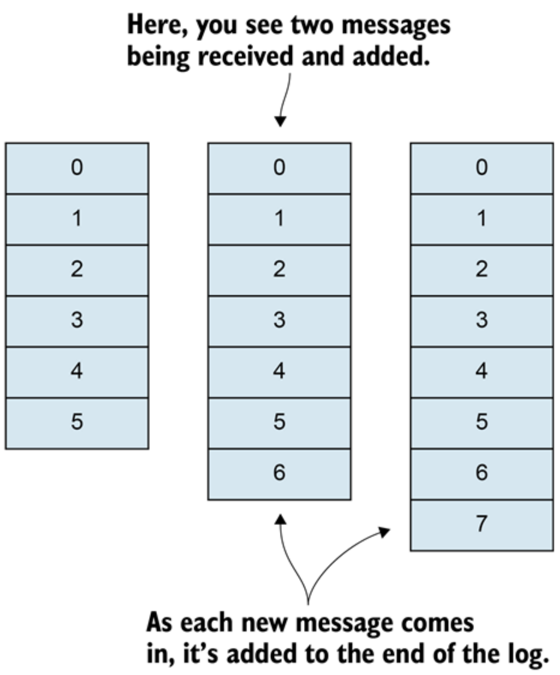

Here, you see two messages being received and added.
0
1
2
3
4
5
0
1
2
3
4
5
6
0
1
2
3
4
5
6
7
As each new message comes in, it's added to the end of the log.

**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.

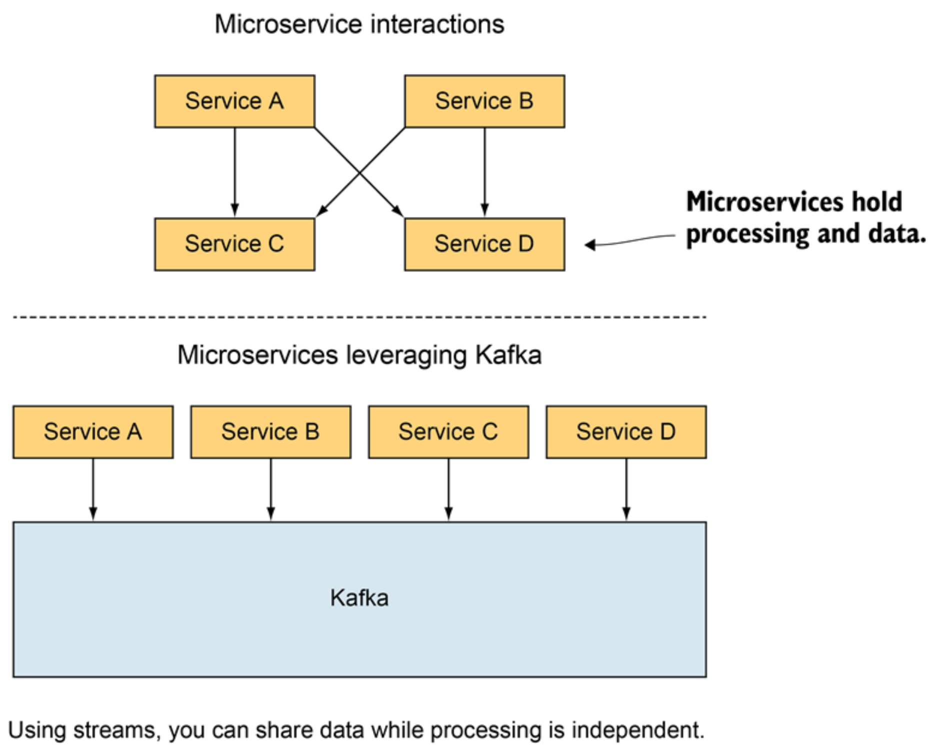

Microservice interactions
Service A
Service B
Service C
Service D
Microservices hold processing and data.
Microservices leveraging Kafka
Service A
Service B
Service C
Service D
Kafka
Using streams, you can share data while processing is independent.

**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.

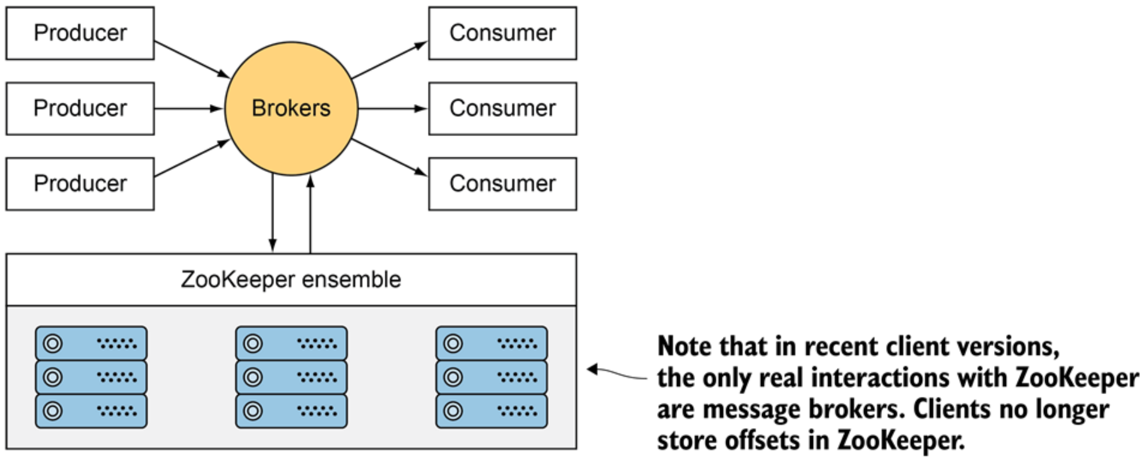

Producer
Producer
Producer
Brokers
Consumer
Consumer
Consumer
ZooKeeper ensemble
Note that in recent client versions, the only real interactions with ZooKeeper are message brokers. Clients no longer store offsets in ZooKeeper.

**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?

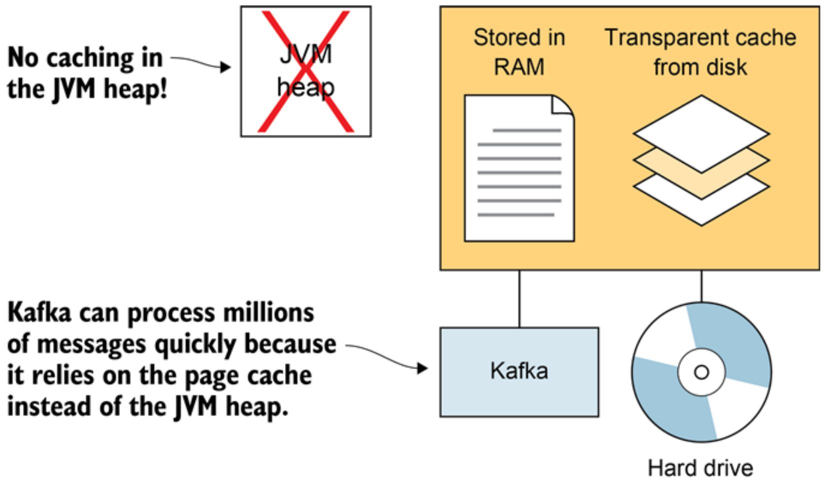

No caching in the JVM heap!
JVM heap
Stored in RAM
Transparent cache from disk
Kafka can process millions of messages quickly because it relies on the page cache instead of the JVM heap.
Kafka
Hard drive

**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 (A Guide To The Kafka Protocol). 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.

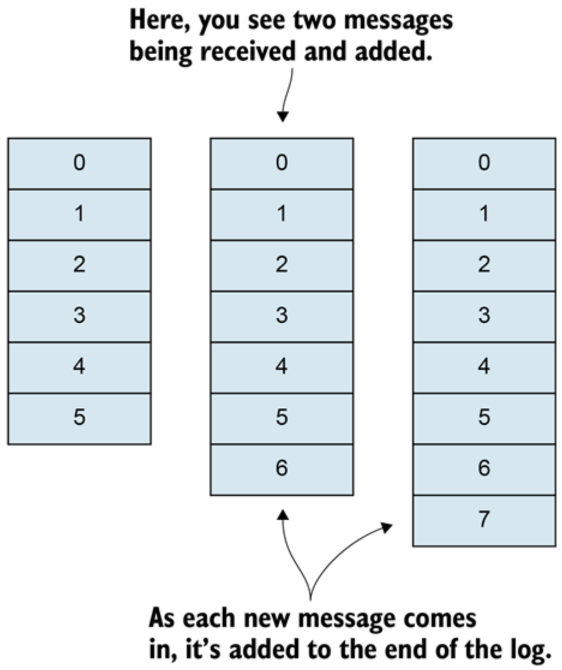

Here, you see two messages being received and added.
0
1
2
3
4
5
6
7
As each new message comes in, it's added to the end of the log.

**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.

## 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.

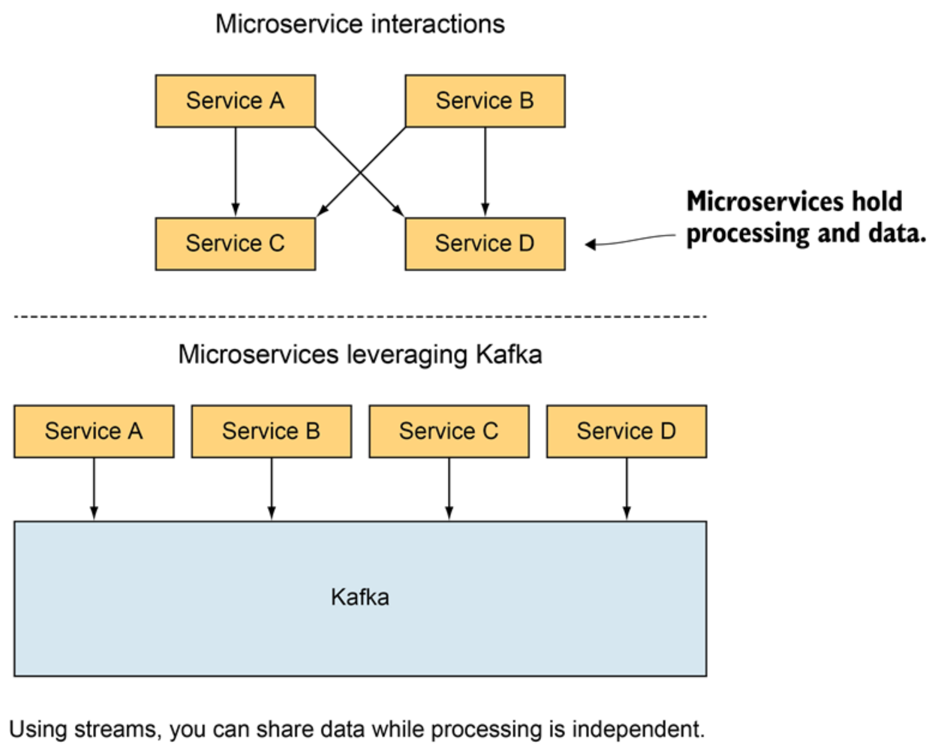
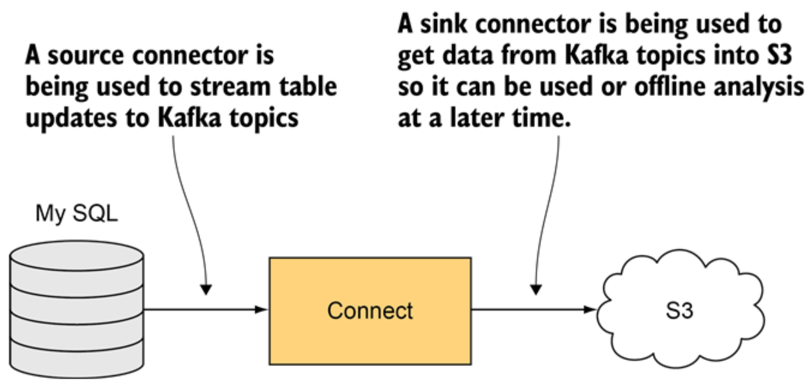

Microservice interactions
Service A
Service B
Service C
Service D
Microservices hold processing and data.
Microservices leveraging Kafka
Service A
Service B
Service C
Service D
Kafka
Using streams, you can share data while processing is independent.

**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.


A source connector is being used to stream table updates to Kafka topics
A sink connector is being used to get data from Kafka topics into S3 so it can be used or offline analysis at a later time.
My SQL
Connect
S3
**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/../Clients 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();                                                                  ❶
props.put("bootstrap.servers", "localhost:9092,localhost:9093");                                     ❷

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

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

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

producer.send(producerRecord);                                                                        ❻

producer.close();                                                                                     ❼
```

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

❷ This property can take a list of message brokers.

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

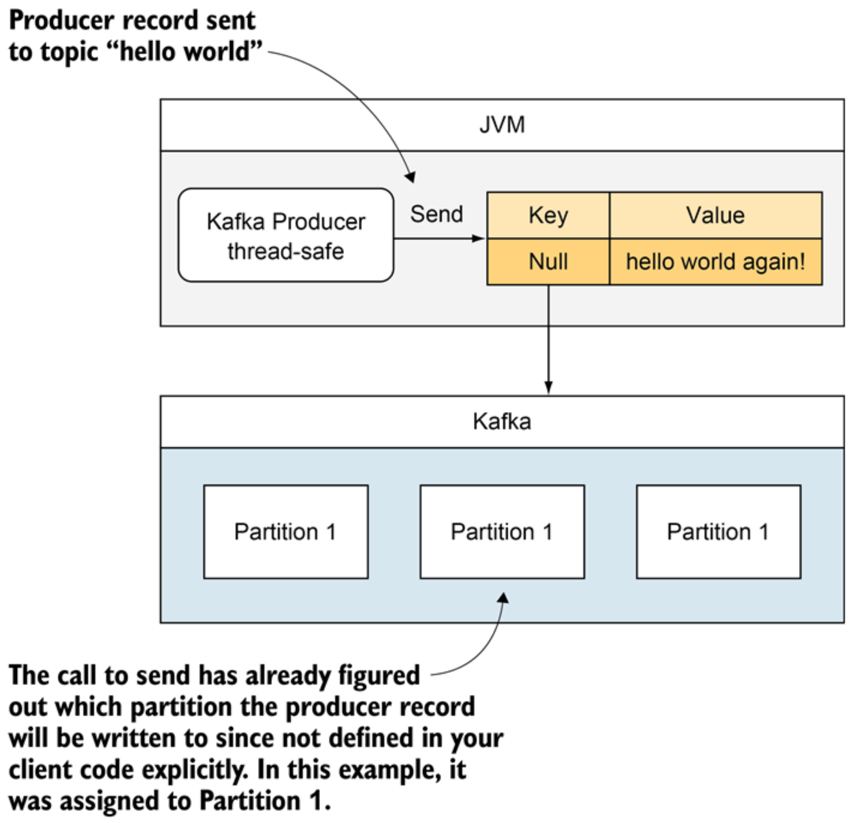
❹ This creates a producer instance. Producers are thread-safe!

❺ This is what represents our message.

❻ 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.



Producer record sent to topic "hello world"
JVM
Kafka Producer thread-safe
Send
Key
Value
Null
hello world again!
Kafka
Partition 1
Partition 1
Partition 1
The call to send has already figured out which partition the producer record will be written to since not defined in your client code explicitly. In this example, it was assigned to Partition 1.

**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();                                          ❶
 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"));                              ❷

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

❶ Properties are set the same way as producers.

❷ The consumer needs to tell Kafka what topics it is interested in.

❸ An infinite loop! But we want to keep polling for new messages as they come in.

❹ 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.

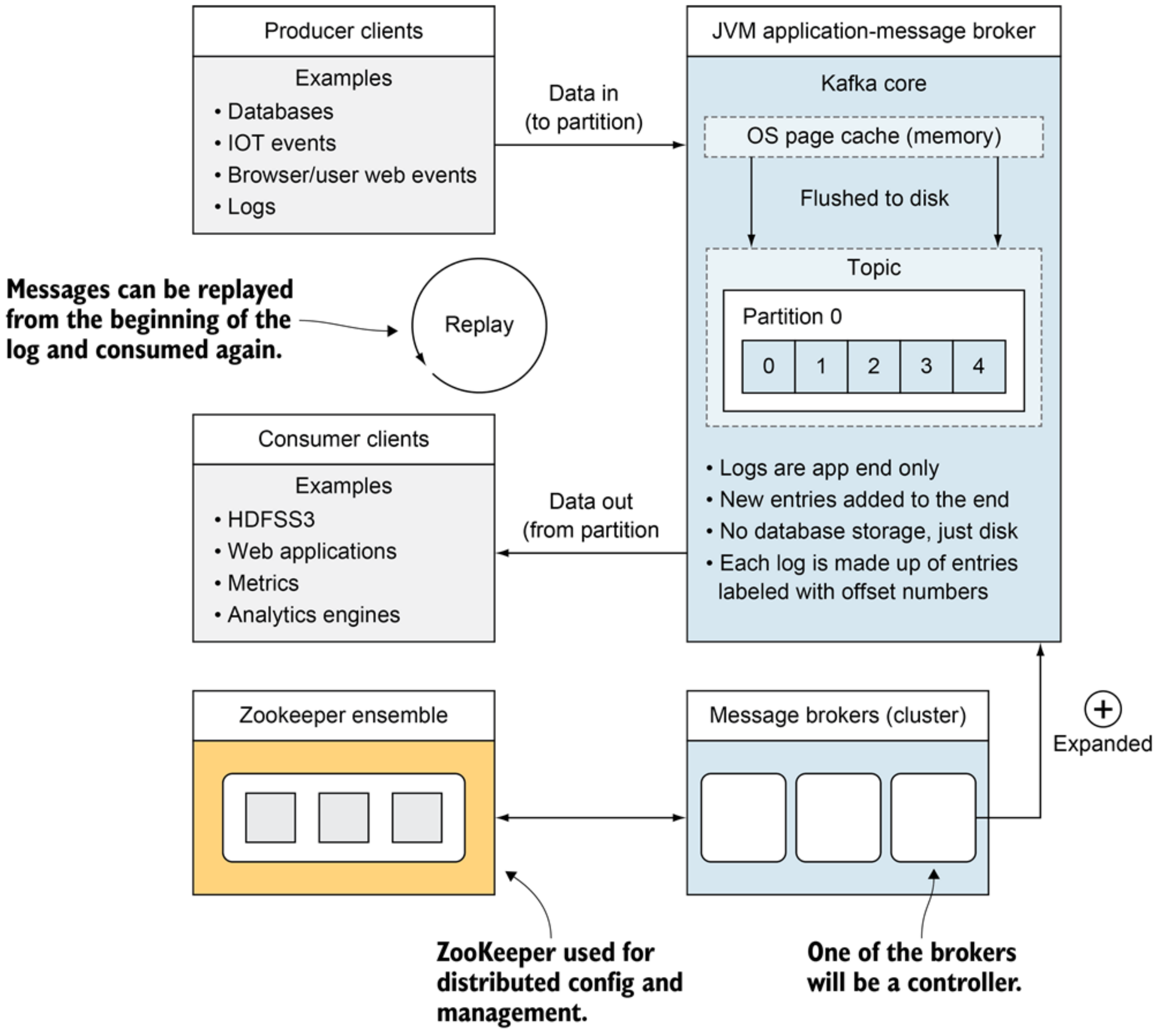

Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Topic
Partition 0
0 1 2 3 4
Messages can be replayed from the beginning of the log and consumed again.
Replay
Consumer clients
Examples
• HDFSS3
• Web applications
• Metrics
• Analytics engines
Data out (from partition
• Logs are app end only
• New entries added to the end
• No database storage, just disk
• Each log is made up of entries labeled with offset numbers
Zookeeper ensemble
Message brokers (cluster)
Expanded
ZooKeeper used for distributed config and management.
One of the brokers will be a controller.

**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.

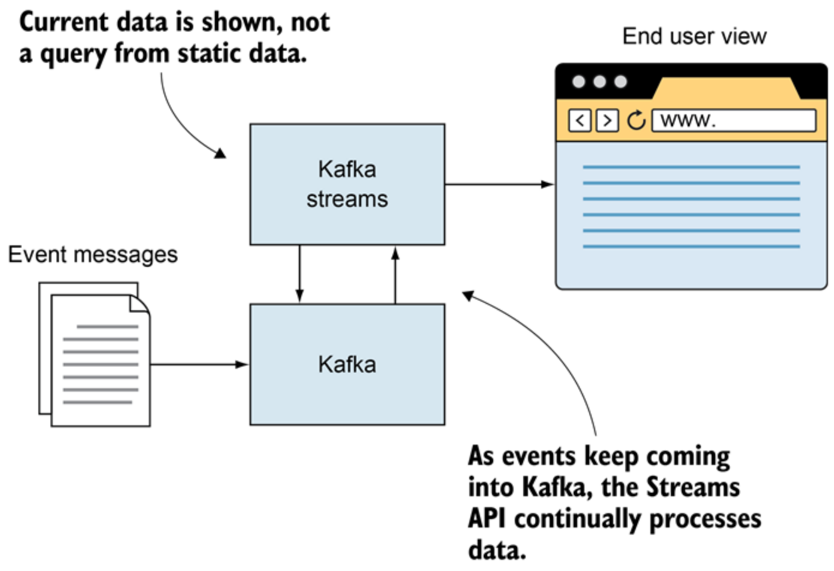

Current data is shown, not a query from static data.
End user view
www.
Kafka streams
Event messages
Kafka
As events keep coming into Kafka, the Streams API continually processes data.

**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     ❶
tasks.max=1                          ❷
file=test.txt                        ❸
topic=connect-test                   ❹
```

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

❷ 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!

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

❹ 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     ❶
tasks.max=1                        ❷
file=test.sink.txt                 ❸
topics=connect-test                ❹
```

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

❷ 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!

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

❹ 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(                             ❶
   id INT PRIMARY KEY     NOT NULL,                ❷
   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 );    ❸
```

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

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

❸ 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    ❶
```

❶ 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.

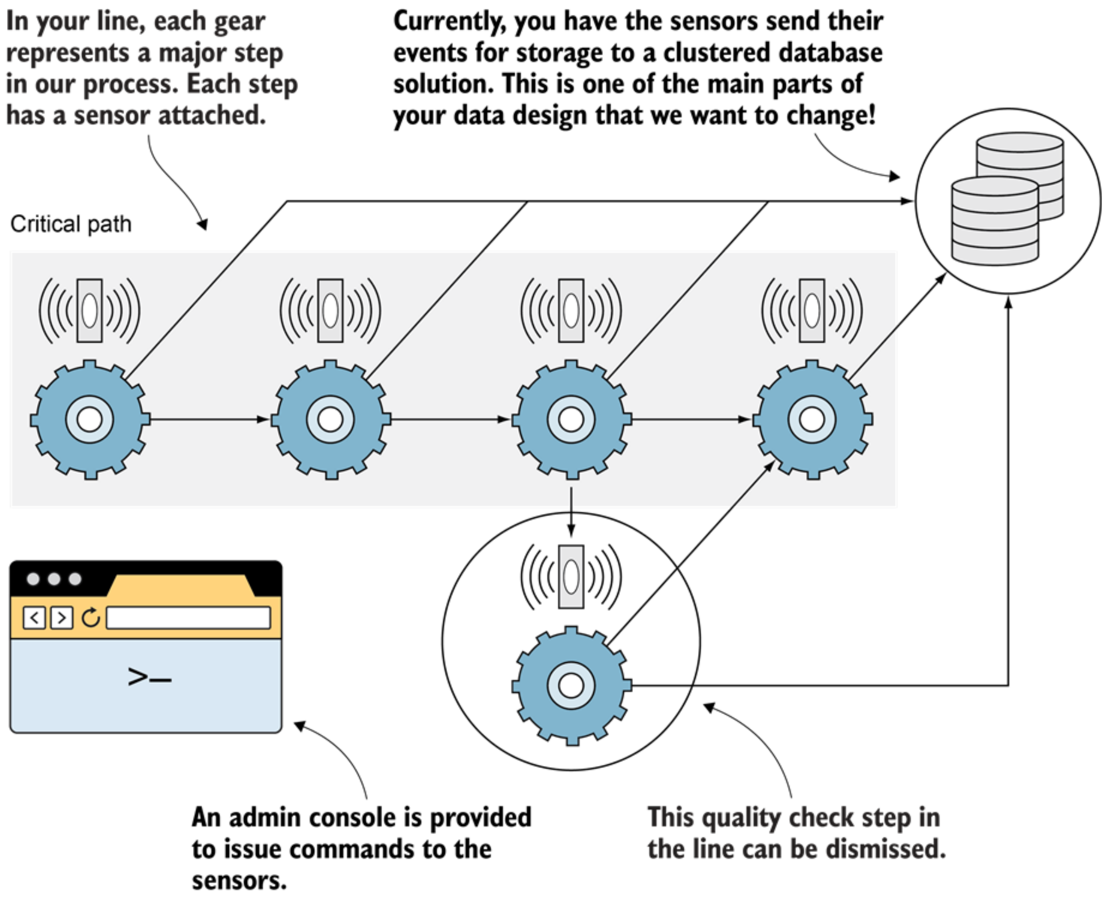

In your line, each gear represents a major step in our process. Each step has a sensor attached.
Currently, you have the sensors send their events for storage to a clustered database solution. This is one of the main parts of your data design that we want to change!
Critical path
>_
An admin console is provided to issue commands to the sensors.
This quality check step in the line can be dismissed.

**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.

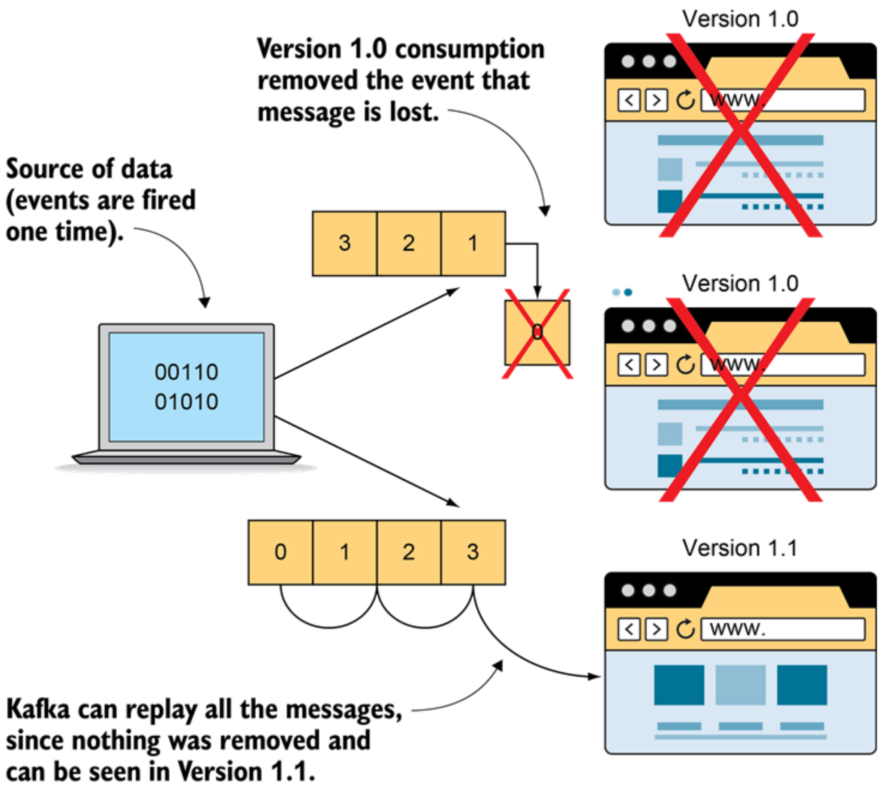
### 3.2.1 Existing issues

#### 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.


Version 1.0 consumption removed the event that message is lost.
Version 1.0
Source of data (events are fired one time).
3
2
1
0
00110
01010
Version 1.0
0
1
2
3
Version 1.1
www.
Kafka can replay all the messages, since nothing was removed and can be seen in Version 1.1.
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.

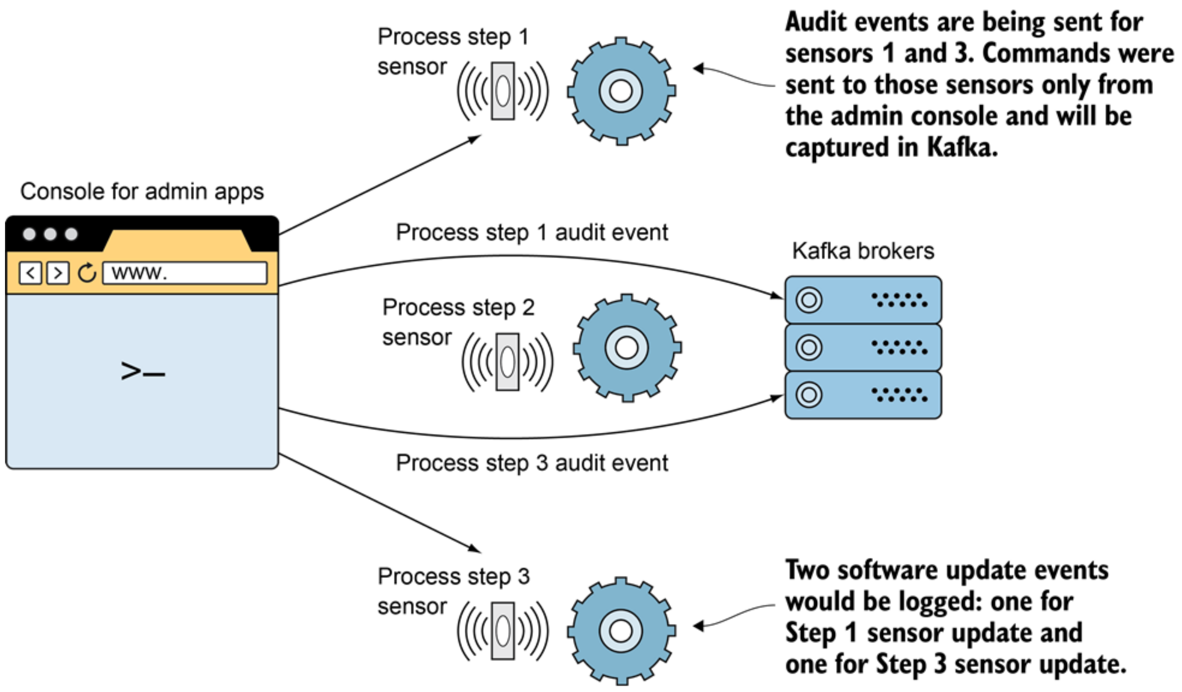

Process step 1 sensor
Audit events are being sent for sensors 1 and 3. Commands were sent to those sensors only from the admin console and will be captured in Kafka.
Console for admin apps
www.
>_
Process step 1 audit event
Kafka brokers
Process step 2 sensor
Process step 3 audit event
Process step 3 sensor
Two software update events would be logged: one for Step 1 sensor update and one for Step 3 sensor update.

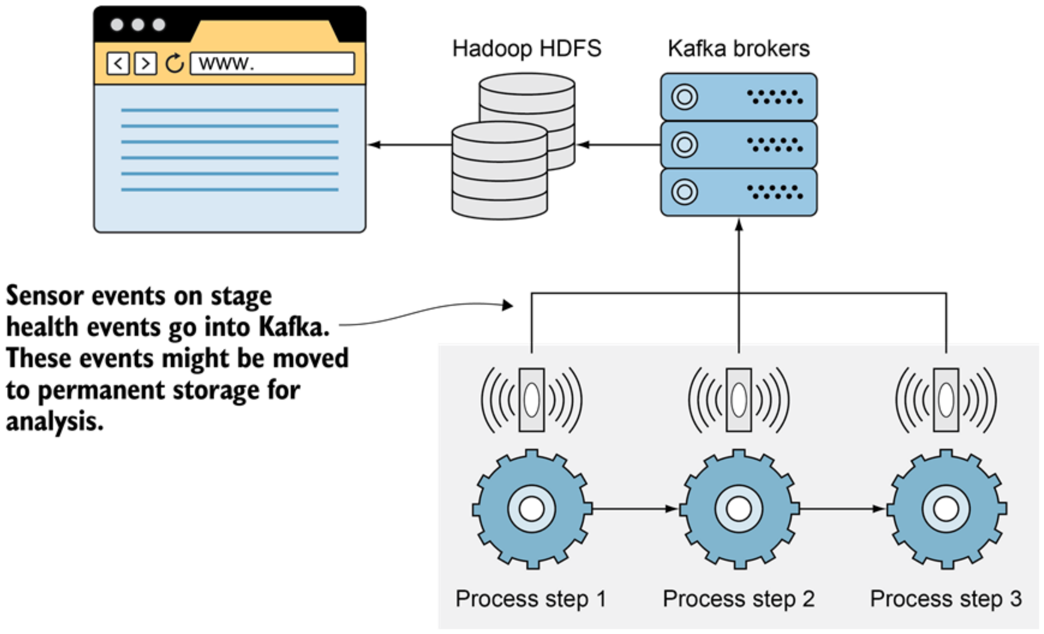
**Figure 3.3 Audit Use Case**

## 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.



www.
Hadoop HDFS
Kafka brokers
Sensor events on stage health events go into Kafka. These events might be moved to permanent storage for analysis.
Process step 1
Process step 2
Process step 3

**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).

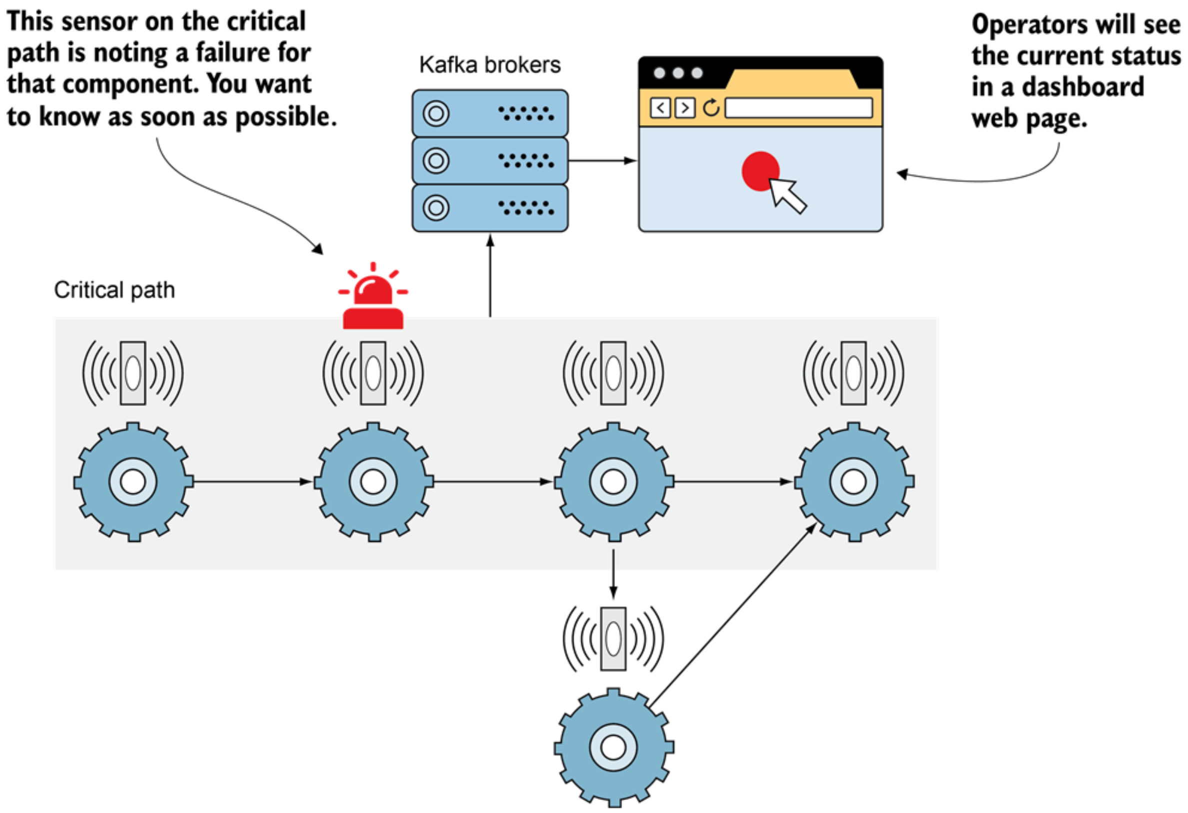

This sensor on the critical path is noting a failure for that component. You want to know as soon as possible.
Kafka brokers
Operators will see the current status in a dashboard web page.
Critical path

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",       ❷
                 "type" : "int",       ❸
                 "default" : -1},      ❹

                 {"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" : {}
                }
    ]
}
```

❶ The schema is defined as json

❷ Usually a mapping to a field name

❸ 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.

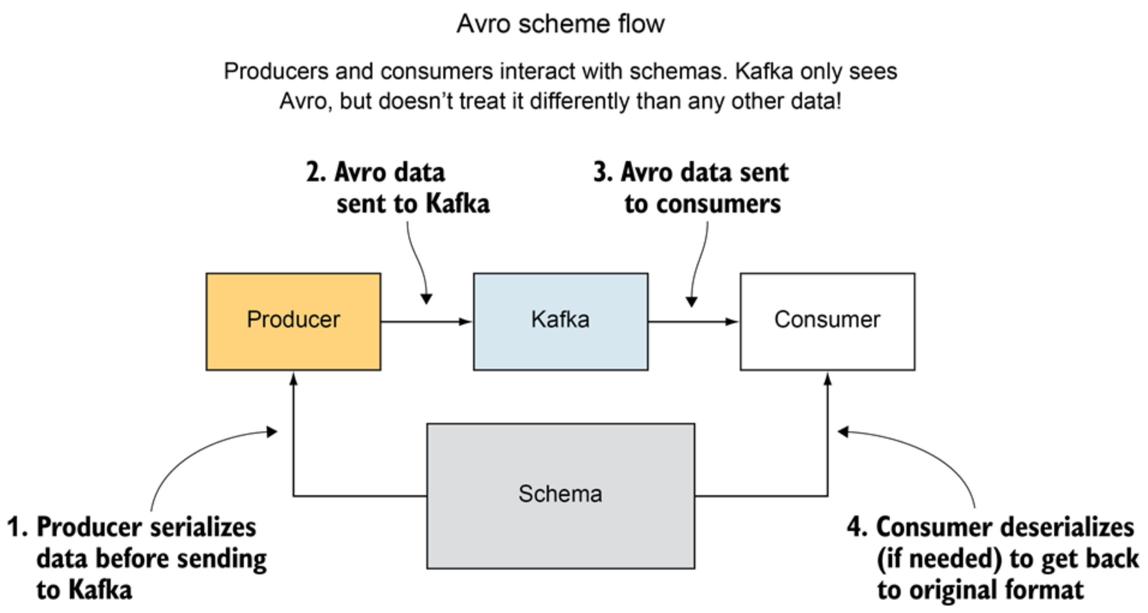

Avro scheme flow
Producers and consumers interact with schemas. Kafka only sees Avro, but doesn't treat it differently than any other data!
2. Avro data sent to Kafka
3. Avro data sent to consumers
Producer
Kafka
Consumer
Schema
1. Producer serializes data before sending to Kafka
4. Consumer deserializes (if needed) to get back to original format

**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>   ❶
  <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>                                   ❶
  <version>1.8.2</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>                                          ❷
      <goals>
        <goal>schema</goal>                                                    ❸
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>  ❹
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>       ❺
      </configuration>
    </execution>
  </executions>
</plugin>
```

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

❷ The Maven phase configuration.

❸ The Maven goal configuration.

❹ The directory that will be scanned for avro files.

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

❶ Namespace will be the generated package you want the source code to be created in.

❷ Alert will be the name of the Java class you will interact with.

❸ 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>    ❶
    <version>3.3.1</version>
</dependency>
```

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

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

## Listing 3.13 Confluent Avro Maven Repository

```
<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 3.14 Producer using Avro Serialization**

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

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

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

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

❶ Now we are sending the Alert as a value and using the KafkaAvroSerializer.

❷ 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.

❸ Creating a alert to show using the generated Java classes from our schema creation.

❹ 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.

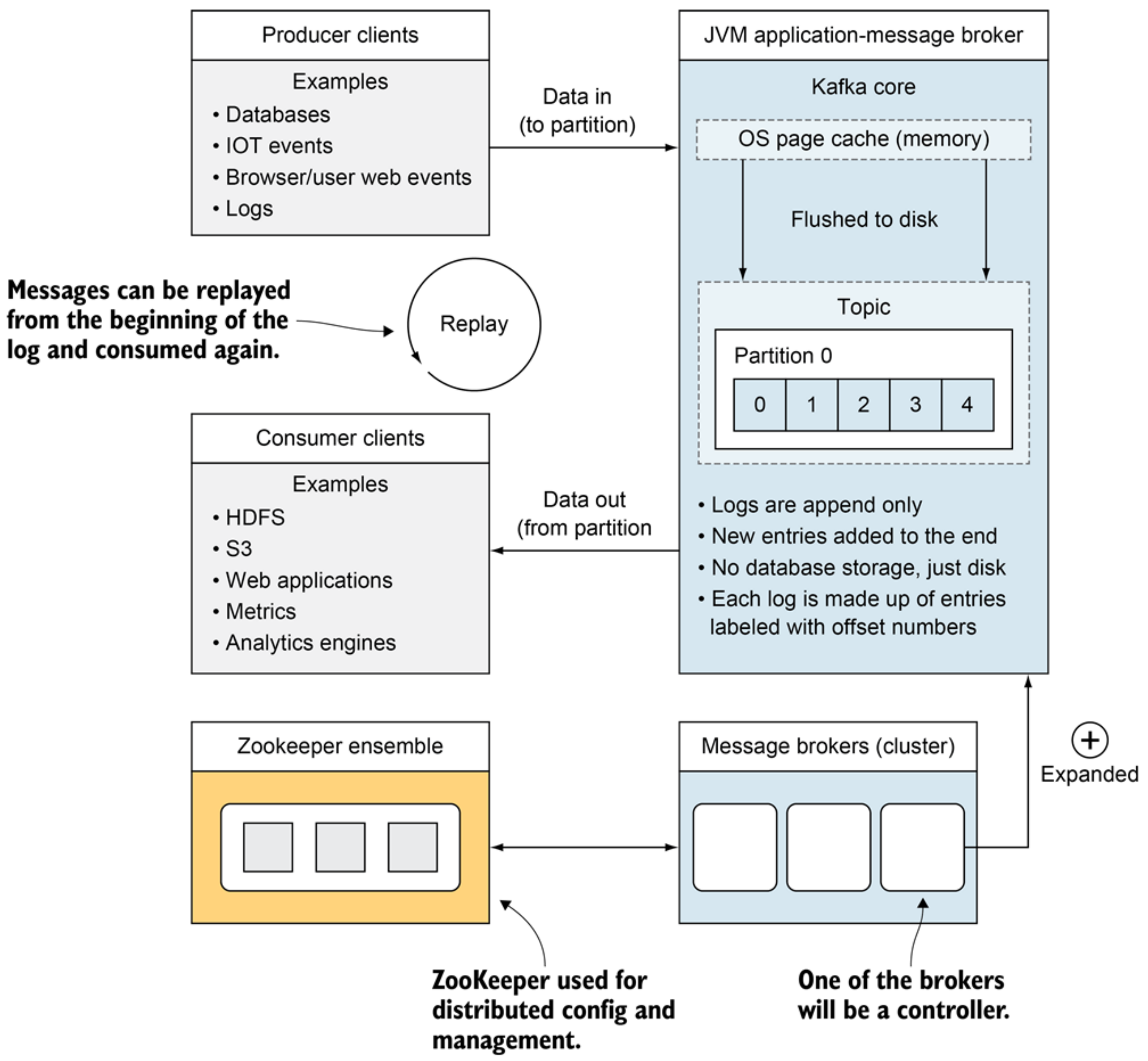

Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Topic
Partition 0
0
1
2
3
4
• Logs are append only
• New entries added to the end
• No database storage, just disk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

**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.

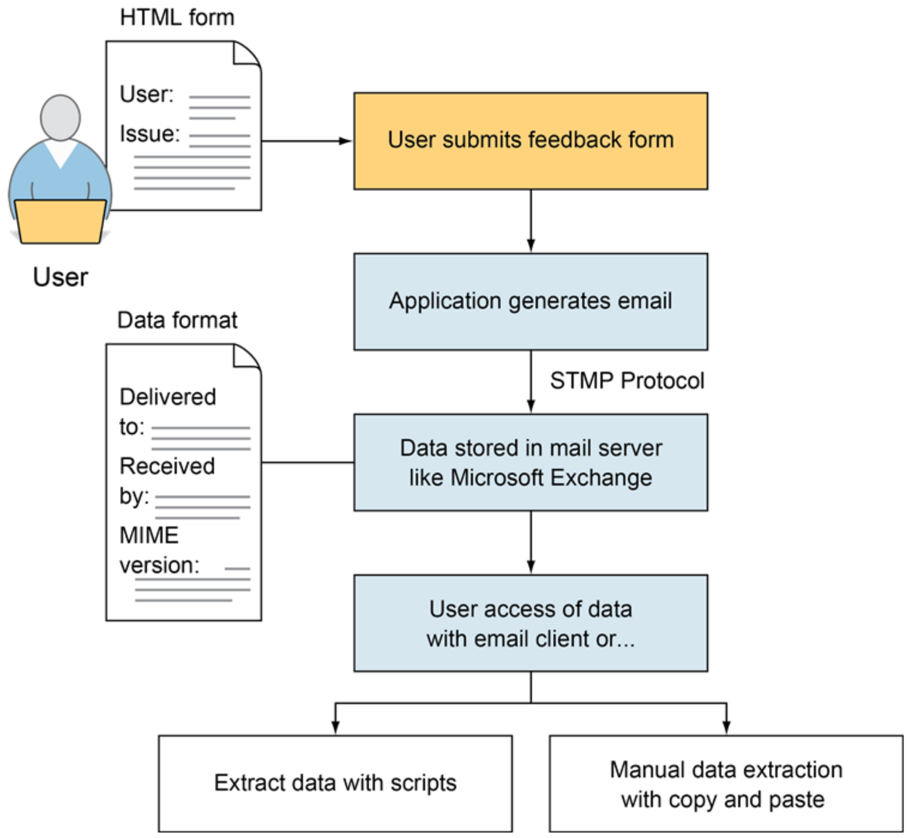

HTML form
User:
Issue:
User
User submits feedback form
Application generates email
STMP Protocol
Data format
Delivered to:
Received by:
MIME version:
Data stored in mail server like Microsoft Exchange
User access of data with email client or...
Extract data with scripts
Manual data extraction with copy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

**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.

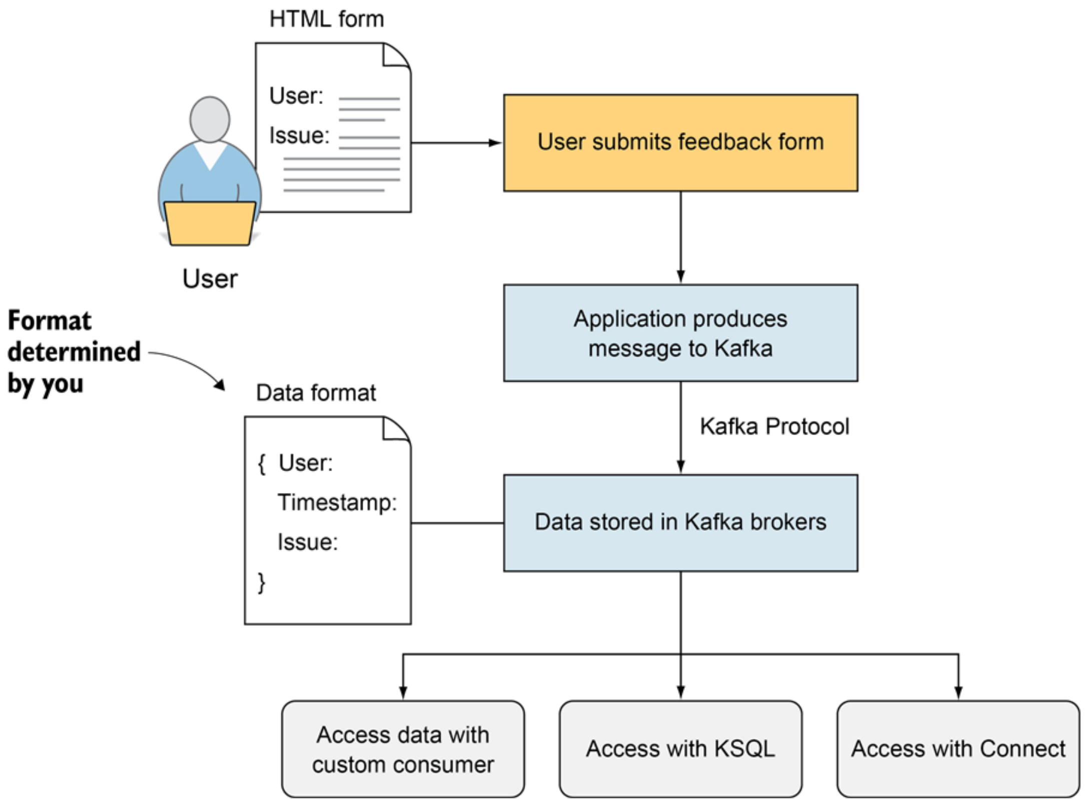


**Figure 4.3 Sending data to Kafka**

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.

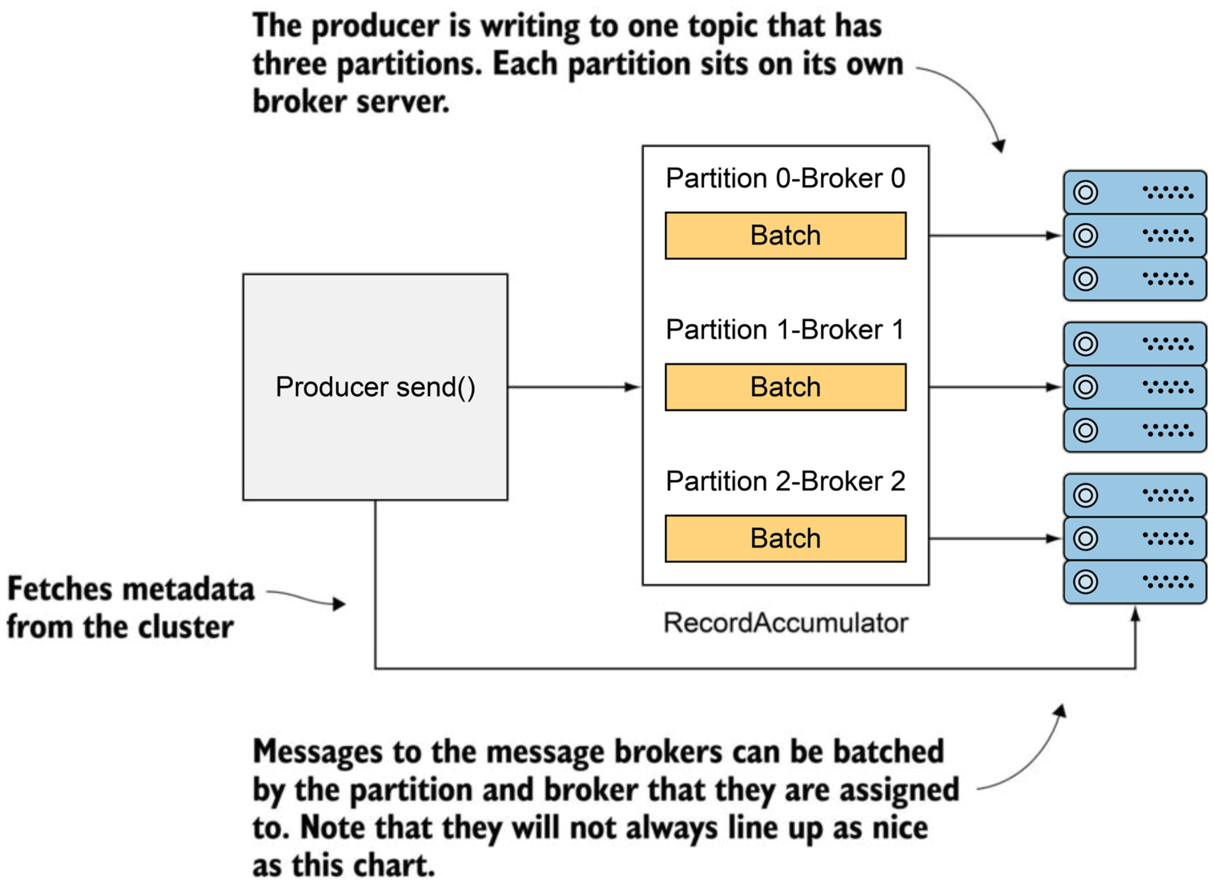

The producer is writing to one topic that has three partitions. Each partition sits on its own broker server.
Partition 0-Broker 0
Batch
Partition 1-Broker 1
Batch
Partition 2-Broker 2
Batch
Producer send()
RecordAccumulator
Fetches metadata from the cluster
Messages to the message brokers can be batched by the partition and broker that they are assigned to. Note that they will not always line up as nice as this chart.

**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 producerconfigs . 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

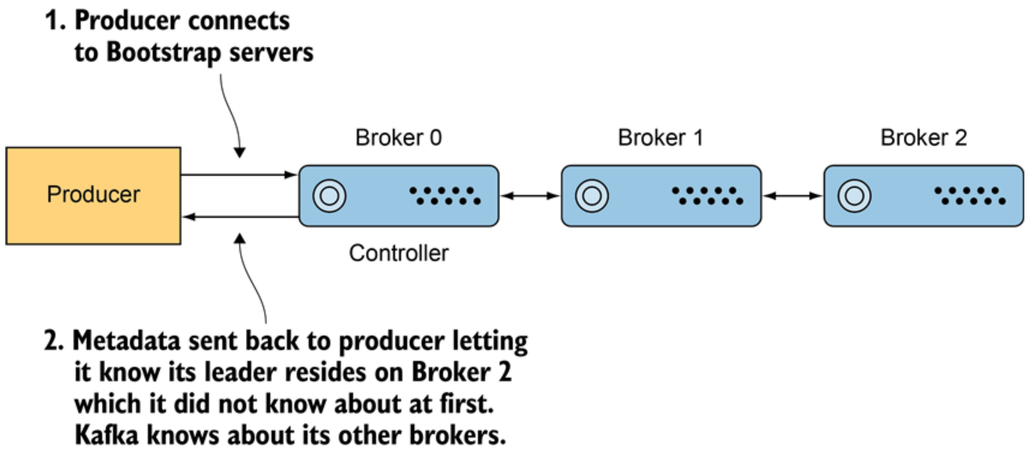

1. Producer connects to Bootstrap servers
Broker 0
Broker 1
Broker 2
Producer
Controller
2. Metadata sent back to producer letting it know its leader resides on Broker 2 which it did not know about at first. Kafka knows about its other brokers.

**Figure 4.5 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.

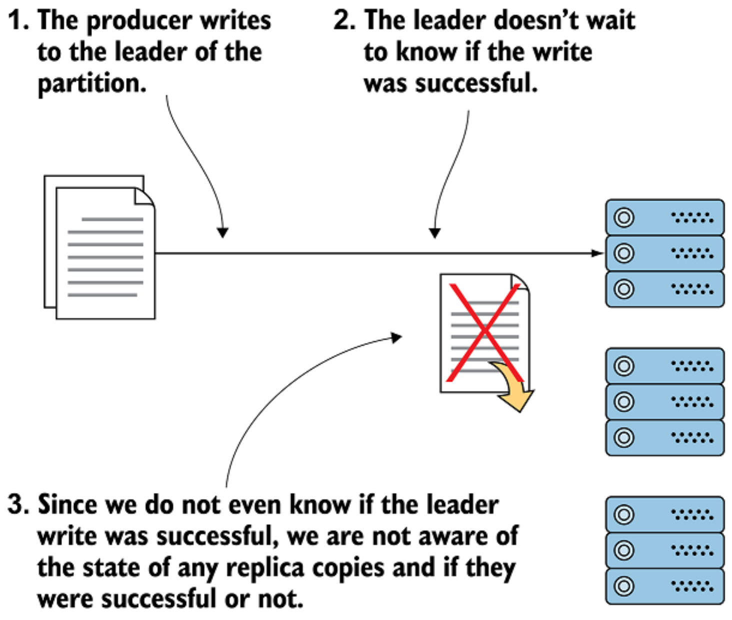

1. The producer writes to the leader of the partition.
2. The leader doesn't wait to know if the write was successful.
3. Since we do not even know if the leader write was successful, we are not aware of the state of any replica copies and if they were successful or not.

**Figure 4.6 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.

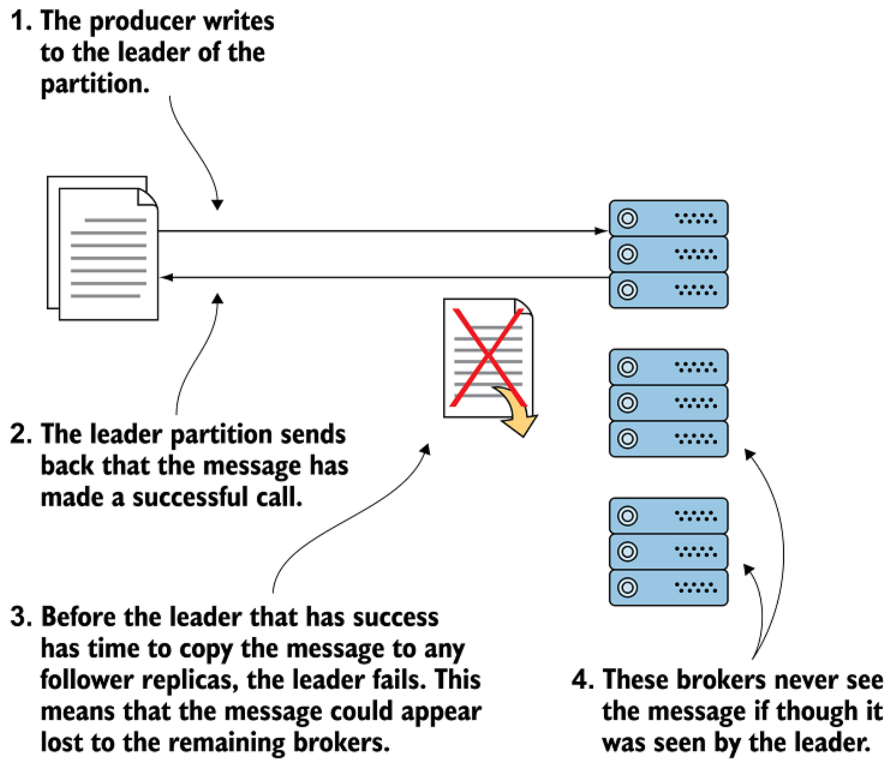

1. The producer writes to the leader of the partition.
2. The leader partition sends back that the message has made a successful call.
3. Before the leader that has success has time to copy the message to any follower replicas, the leader fails. This means that the message could appear lost to the remaining brokers.
4. These brokers never see the message if though it was seen by the leader.

**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.

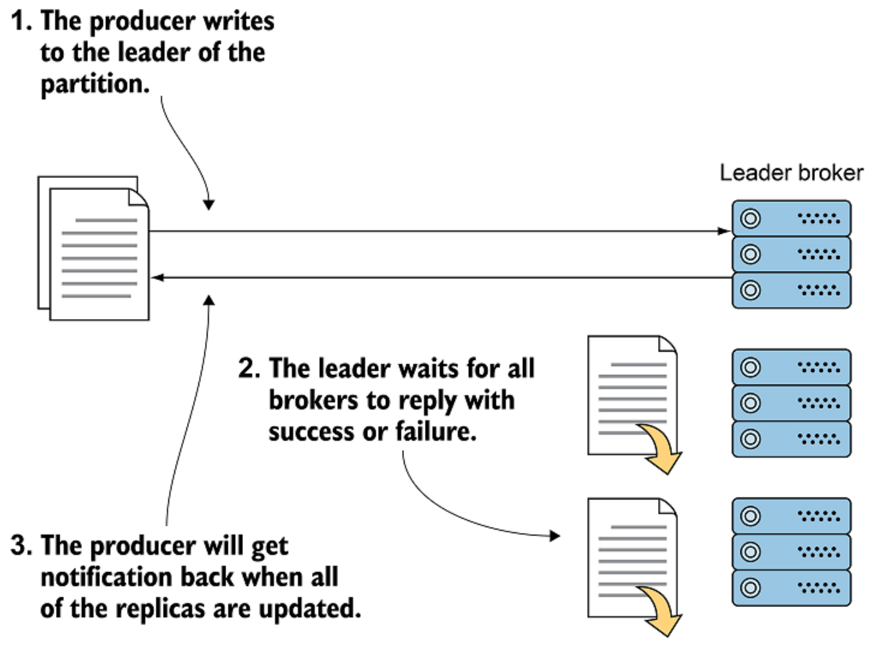

1. The producer writes to the leader of the partition.
Leader broker
2. The leader waits for all brokers to reply with success or failure.
3. The producer will get notification back when all of the replicas are updated.

**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.

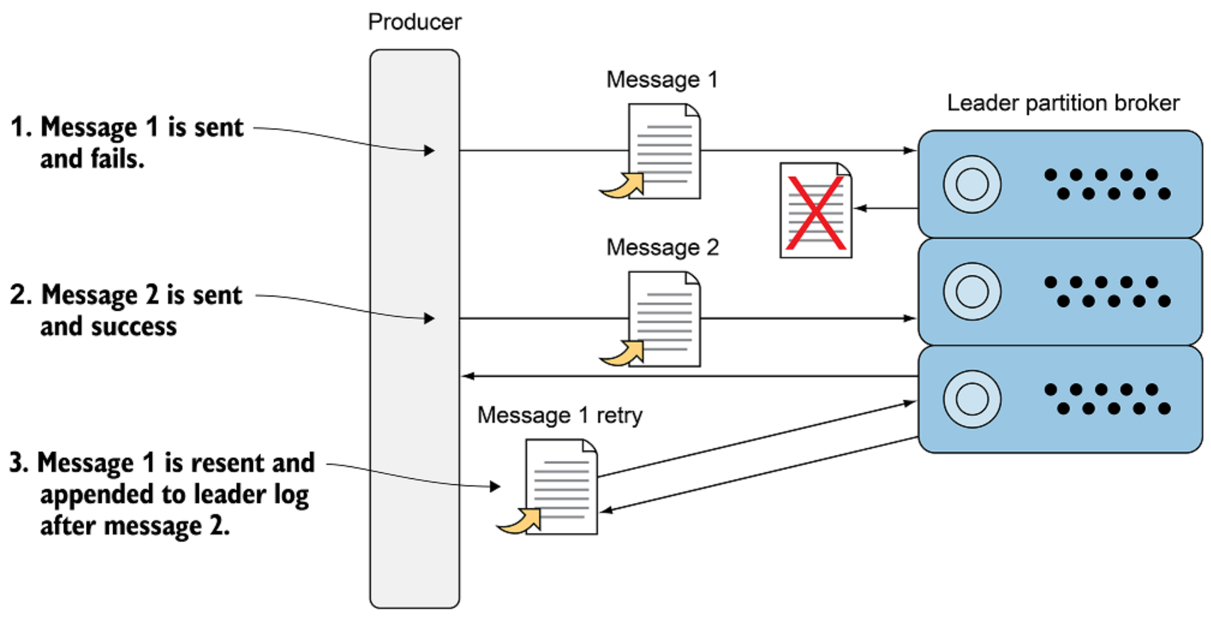

Producer
Message 1
Leader partition broker
1. Message 1 is sent and fails.
Message 2
2. Message 2 is sent and success
Message 1 retry
3. Message 1 is resent and appended to leader log after message 2.

**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) {   ❶
        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;
    }
}
```

❶ 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> {   ❶

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

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

    }

    public void close() {   ❹
        // 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;
    }

}
```

❶ We implement the Serializer interface to be a valid extension point.

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

❸ Our end goal is to convert objects to bytes.

❹ 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

## 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 {   ❶

    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")) {   ❷
            return criticalPartition;
        } else {
            return Math.abs(key.hashCode()) % partitionCount;
        }

    }

    public void close() {
        // nothing needed
    }

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

}
```

❶ We need to implement the Partitioner interface.

❷ 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",    ❶
"com.kafkainaction.partitioner.AlertLevelPartitioner");
```

❶ 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();    ❶
props.put("bootstrap.servers", "localhost:9092,localhost:9093");
props.put("acks", "all");    ❷
props.put("retries", "3");    ❸
props.put("max.in.flight.requests.per.connection", "5");
```

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

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

❸ We are letting the client do retries for us in cases of failure so we don’t have to 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();   ❶
System.out.printf("offset = %d, topic = %s,
timestamp = %Tc %n", result.offset(), result.topic(),
result.timestamp());
```

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

Waiting on the response directly 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");    ❶
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());   ❷

producer.send(producerRecord);
```

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

❷ Instead of null for the 2nd parameter, we put the actual object we wish to 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");     ❶

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

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());     ❸
```

❶ We are reusing our custom Alert key serializer

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

❸ This is the first time we have used 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 {   ❶
        public void onCompletion(RecordMetadata metadata, Exception exception) {   ❷
            if(exception != null){
               System.out.printf("Error sending message: "+ "offset = %d, topic = %s,
               timestamp = %Tc %n", metadata.offset(), metadata.topic(), metadata.timestamp());
            }
        }
}
```

❶ We need to implement the Kafka interface Callback

❷ 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/../kafka/KafkaSink.java. 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));
          }   ❶
          kafkaFutures.add(producer.send(record, new SinkCallback(startTime)));    ❷
   ...
      }
   ...
      producer.flush();
  }
  public synchronized void start() {
    producer = new KafkaProducer<String,byte[]>(kafkaProps);   ❸
    ...
  }
...
  private void setProducerProps(Context context, String bootStrapServers) {
    kafkaProps.clear();
    kafkaProps.put(ProducerConfig.ACKS_CONFIG, DEFAULT_ACKS);   ❹
    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);
  }

}
```

❶ Logic to choose which ProducerRecord constructor to create the message.

❷ Our familiar producer.send with a callback.

❸ Creating our producer with properties.

❹ 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 {   ❶
  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);
    }
  }
}
```

❶ 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.

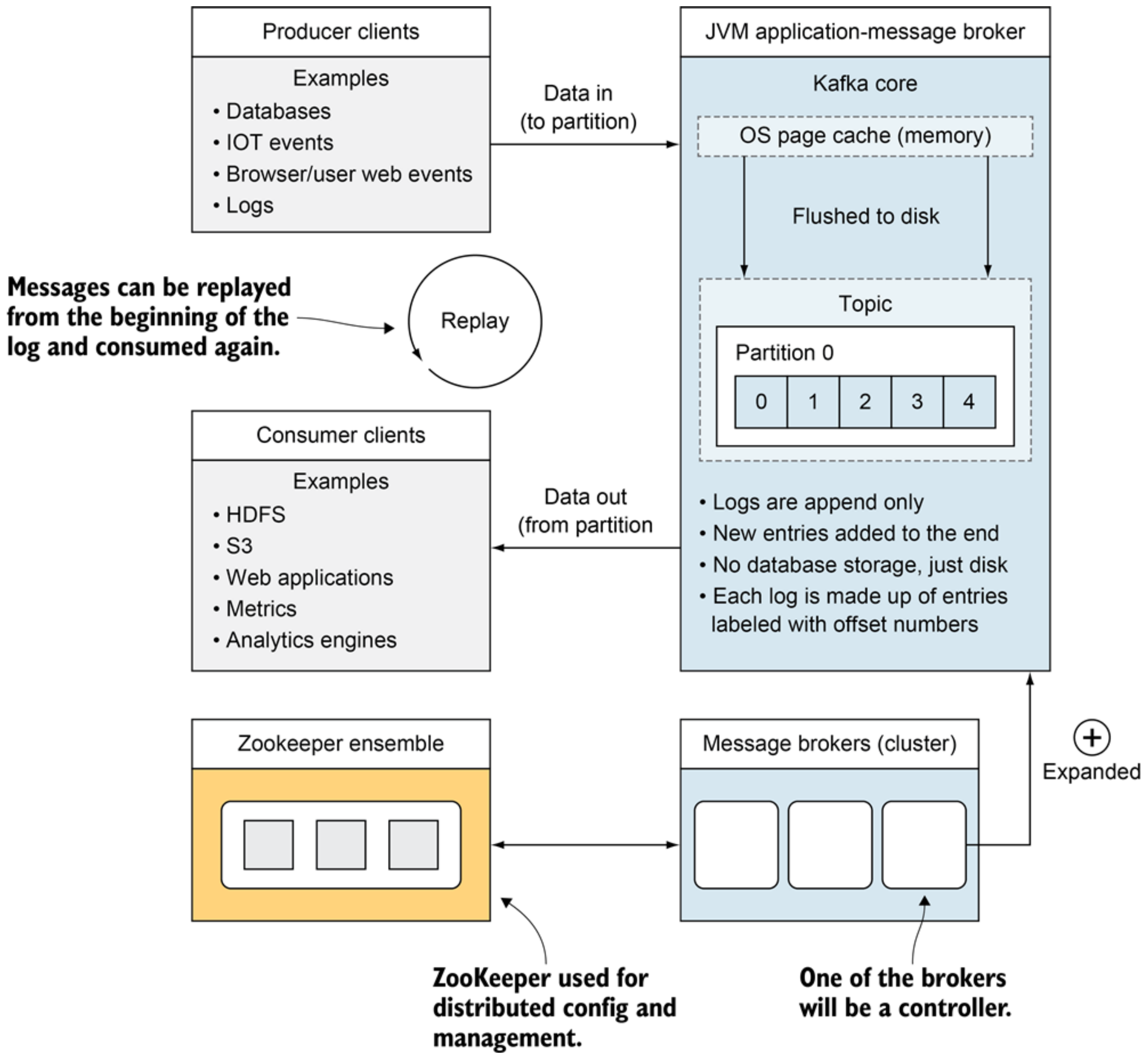

Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Topic
Partition 0
0
1
2
3
4
Messages can be replayed from the beginning of the log and consumed again.
Replay
Consumer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

**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");   ❶
    props.put("key.deserializer",   ❷
    "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",   ❷
    "org.apache.kafka.common.serialization.StringDeserializer");

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

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

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);   ❺
        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
}
```

❶ 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.

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

❸ We use our properties to create an instance of a consumer

❹ 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.

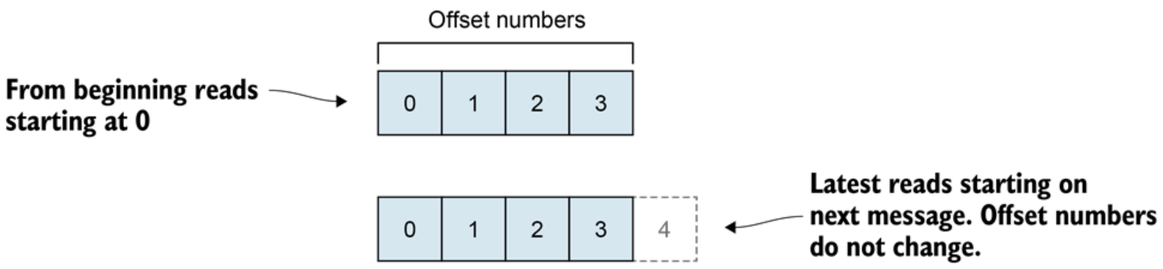

Offset numbers
From beginning reads starting at 0
0
1
2
3
0
1
2
3
4
Latest reads starting on next message. Offset numbers do not change.

**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.

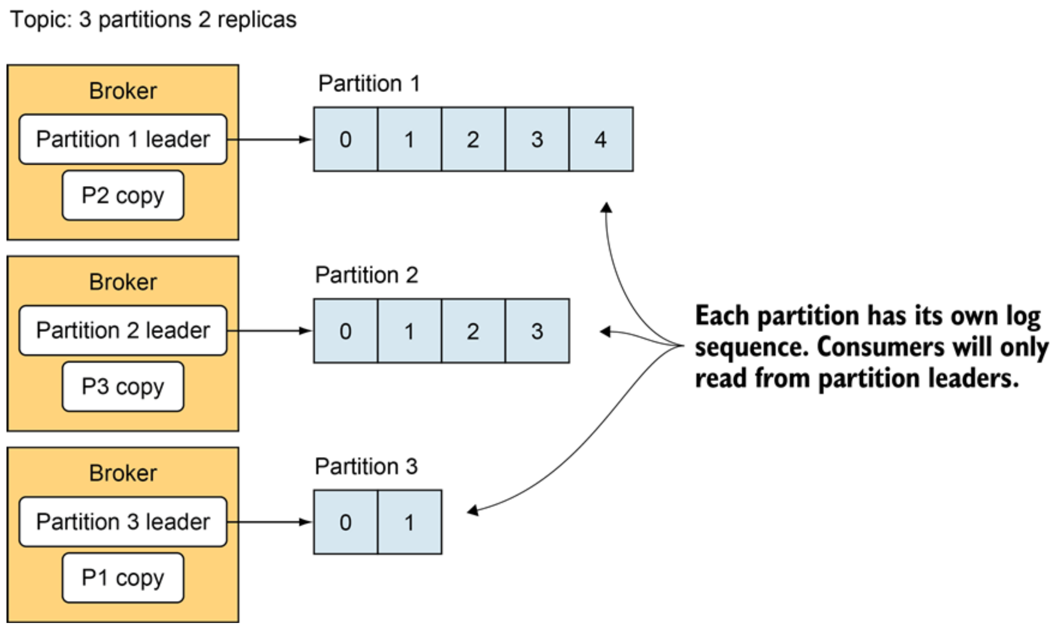


**Figure 5.3 Partition Leaders**

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.

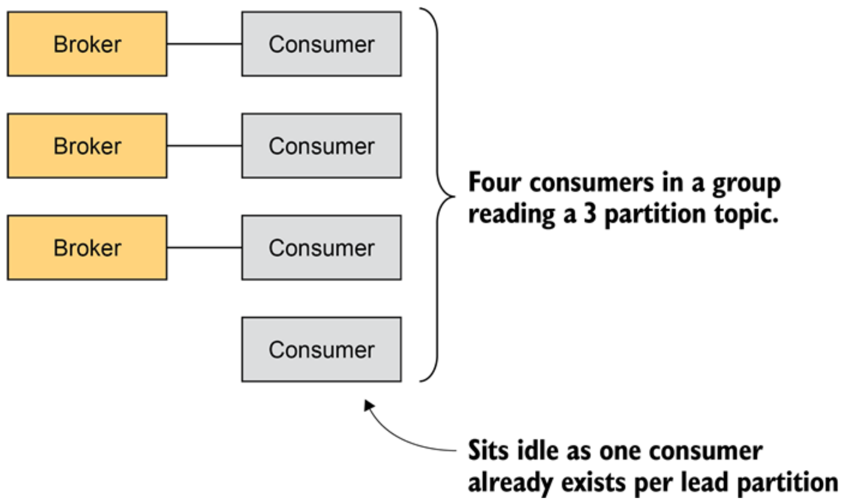

Broker
Consumer
Broker
Consumer
Broker
Consumer
Consumer
Four consumers in a group reading a 3 partition topic.
Sits idle as one consumer already exists per lead partition

**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()) {   ❶
                ConsumerRecords<String, String> records = consumer.poll(100);
                ...
            }
        } catch (WakeupException e) {    ❷
            if (!stopping.get()) throw e;
        } finally {
            consumer.close();    ❸
        }
    }

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

❶ The stopping variable will be our guard condition on if we keep processing or not

❷ The WakupException is what is trigger by the client shutdown hook

❸ Close properly stops the client and informs the broker

❹ 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.

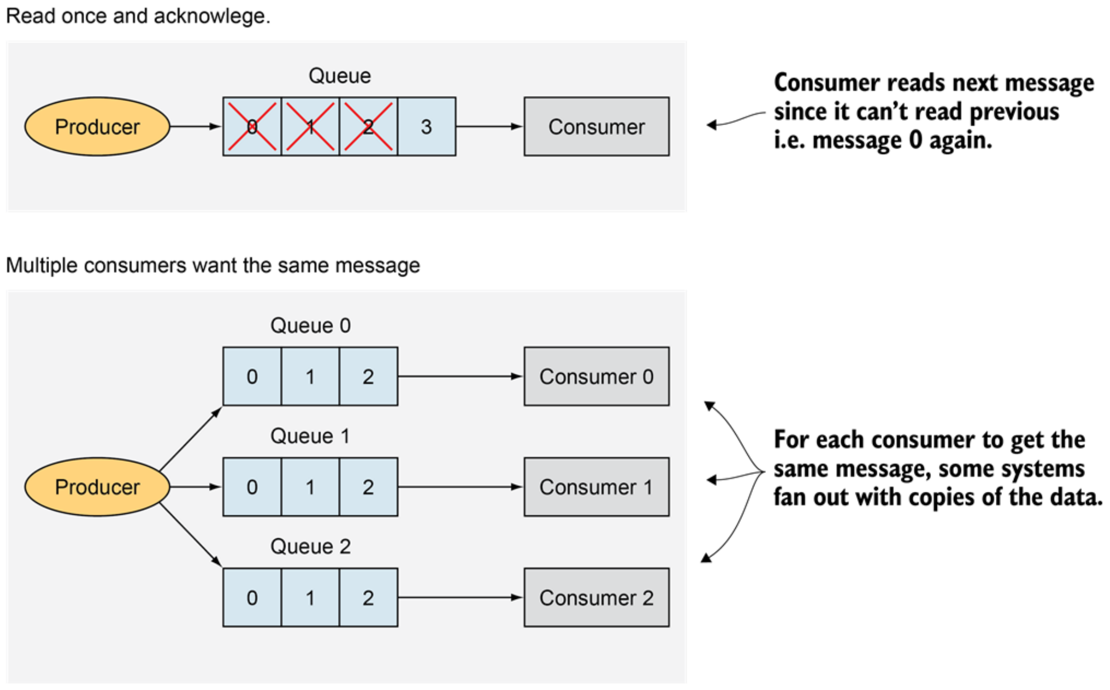

Read once and acknowlege.
Queue
Producer
0
1
2
3
Consumer
Consumer reads next message since it can't read previous i.e. message 0 again.
Multiple consumers want the same message
Queue 0
Queue 1
Queue 2
Consumer 0
Consumer 1
Consumer 2
For each consumer to get the same message, some systems fan out with copies of the data.

**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.

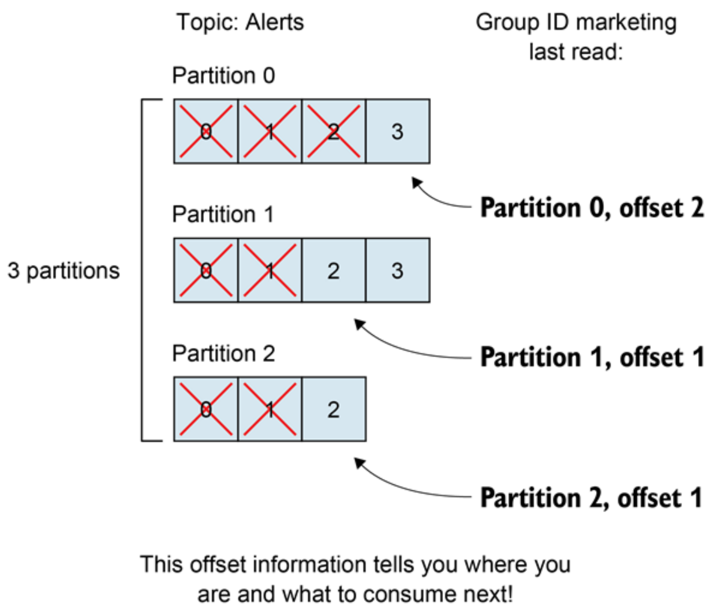
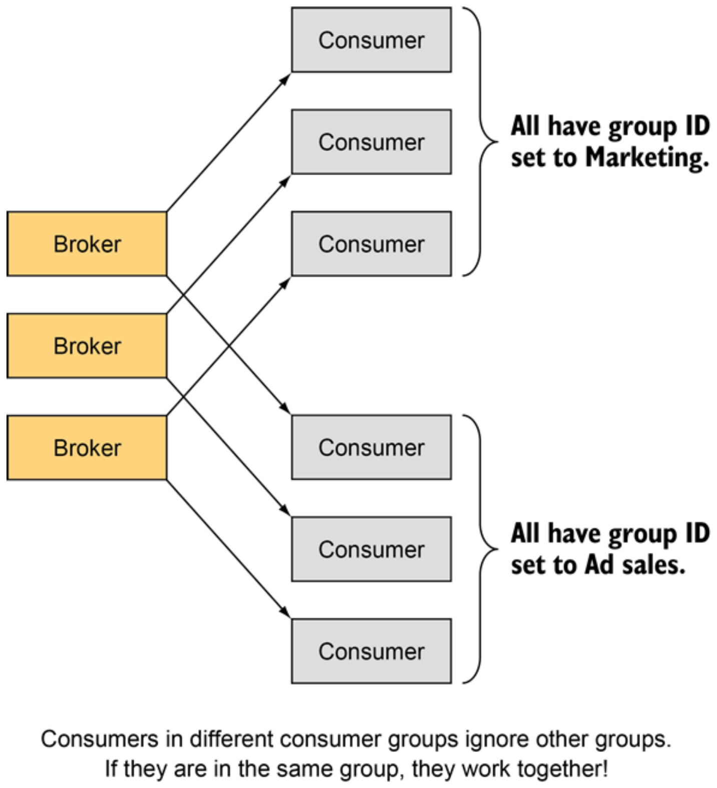


**Figure 5.6 Coordinates**

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



Consumer
Consumer
Consumer
All have group ID set to Marketing.
Broker
Broker
Broker
Consumer
Consumer
Consumer
All have group ID set to Ad sales.
Consumers in different consumer groups ignore other groups. If they are in the same group, they work together!

**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.

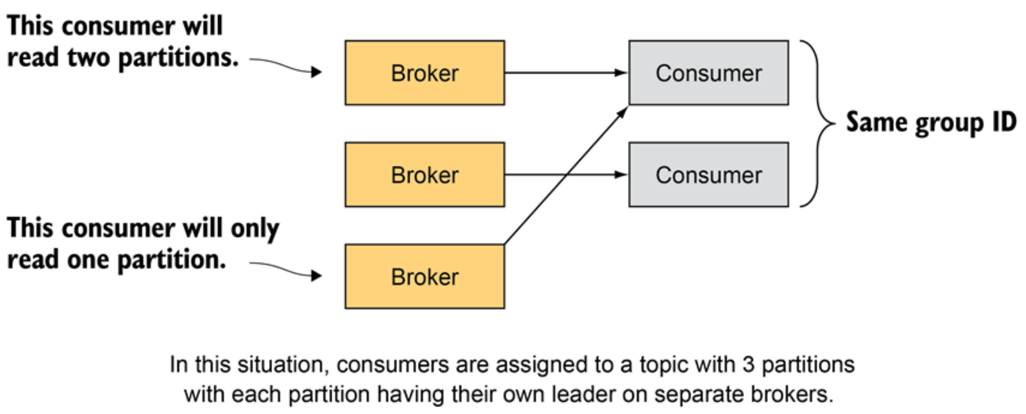

This consumer will read two partitions.
Broker
Consumer
Same group ID
Broker
Consumer
This consumer will only read one partition.
Broker
In this situation, consumers are assigned to a topic with 3 partitions with each partition having their own leader on separate brokers.

**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 {    ❶
    ...

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

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

❶ The ConsumerRebalanceListener interface is needed to add our custom logic

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

❸ 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.

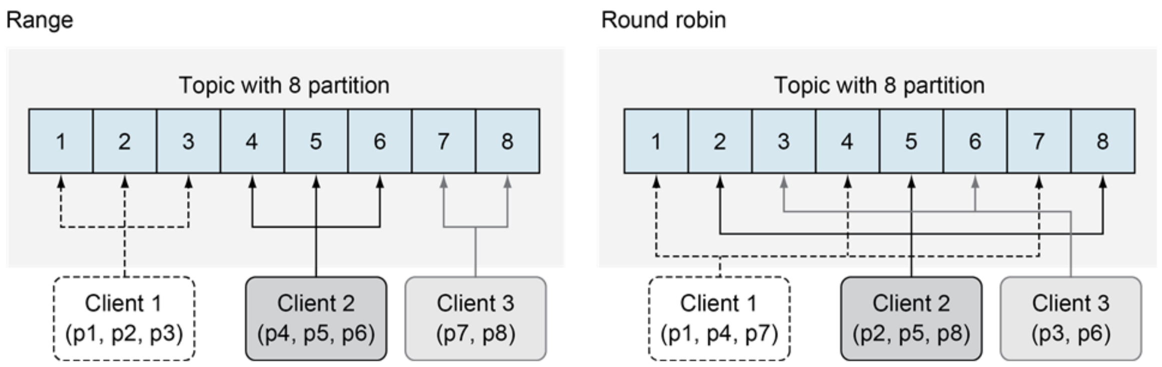


**Figure 5.9 Partition Assignment**

- 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));   ❶
```

❶ 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();    ❶
    }
}
```

❶ 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);    ❶
    }
}
```

❶ 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() {    ❶
        @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());    ❶
props.put("auto.offset.reset", "earliest");    ❷
```

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

❷ 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());    ❶
props.put("auto.offset.reset", "latest");    ❷
```

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

❷ 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);    ❶
...
// 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

# 6 Brokers

***This chapters covers:***

- Exploring the roles of brokers in our context
- Important metrics to monitor in our cluster
- Evaluating tradeoffs for certain broker configuration values

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

## 6.1 Introducing the Broker

Some queues have the concept of a 'smart' broker and a 'dumb' client or the reverse: a 'dumb' broker and a 'smart' client. These terms might fit other queue and/or topic systems well but I would not get too involved in trying to make Kafka brokers and clients fit in either category. In my opinion, I think that both parts have a lot of power. Being a broker in Kafka means being a part of a cluster of machines.

If you are familiar with Big Data concepts or worked with Hadoop before, you might near similar terminology such as rack awareness and partitions. For example, Kafka has a rack awareness feature that will make replicas of the same partition exist physically across different racks. This is important since if all of the servers that make up your cluster are on the same rack, and the entire rack is offline, it would be the same as if you had lost your entire cluster.

If you are setting up your own Kafka cluster, it is important to know that you have another cluster to be aware of: Zookeeper. Since Zookeeper needs to have a quorum in order to elect leaders and reach a decision, this cluster is indeed important for our brokers. One of the most common questions we all have at some point is how many nodes in my cluster make sense. One of your drivers should be how much you are willing to work with. For example, if you have 3 Zookeeper nodes you can afford to lose one, but if you have 5 nodes you can afford to lose two. So if you account for taking a node off for maintenance, upgrades, or general failures, keep that in mind when choosing the size of your cluster. As we often experience, it is a good idea to not go overboard on the numbers in the Zookeeper cluster. When you are approaching nine nodes, you might want to verify your performance of your cluster overall. The more nodes you have, the longer it might take to reach quorum across more nodes.

## 6.2 Why Kafka needs Zookeeper

At this point, Zookeeper is a key part of how the brokers work and is a requirement for Kafka to run. Zookeeper itself holds a list of the brokers that are members of a cluster. This list should be a current membership list and can change over time. How do brokers join this list? When a broker starts, it will have a unique id that will create an ephemeral node on Zookeeper. This id can either be created with the configuration `broker.id` or `broker.id.generation.enable=true`. Nodes can also leave the group as well.

With all of the interaction with the brokers, it is very important that you have Zookeeper running when starting your brokers. The health of that cluster will impact the health of your Kafka brokers. For instance, if you got your Zookeeper instances blown away, topic metadata and configuration would be lost. Also the assignments of replicas to broker ids would be lost and would be hard to recover the data at all.

Usually you won’t expose your clients to the server details for the ZooKeeper cluster. If you are using certain frameworks, note that they still might want control that you might not want to give your clients. One example is Spring Cloud Stream that allows a parameter that `zkNodes` can be provided. It would always default to `localhost` as well and was something that most clients would not need that dependency especially for starting up. That property is marked as a deprecated configuration property as it should be in most cases.

While you might not need to know the internals of how Kafka uses ZooKeeper, it is nice to have some ideas of how to look at various items in case you have to troubleshoot any issues. Using the Kafka tool `zookeeper-shell.sh` we can connect to a ZooKeeper host in our cluster and look at how data is stored. If we look at the path `/brokers/topics` you should see a list of the topics that you have created. At this point, we should have the `helloworld` topic in our list. You should also be able to see the topic `__consumer_offsets`. That topic is one that you did not create, it is in fact a private topic used internal by Kafka itself. This topic stored the committed offsets for each topic and partition per group id. You should be aware of this topic so you are not surprised or concerned why topics you have not created are appearing in topic lists on your cluster.

**Listing 6.1 Listing our topics**

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

❶ Connect to the your zookeeper instance

❷ List all of the topics with `ls`

Your output should include the topic that we created in our previous chapter: `[helloworld]`

Using a different Kafka tool `kafka-topics.sh --list --zookeeper host:port` you should be able to see the list of topics as well and get the same results!

## 6.3 What does it mean to be a message broker

Being a message broker means being able to coordinate with the other message brokers as well as talking to ZooKeeper. In testing or your proof-of-concept clusters, you might have only one node. However, this is something that should not take place in production. Figure 6.1 shows how brokers will exist as a cluster and will be where the logs for Kafka are actually stored. Clients will also be writing and reading from brokers in order to get information into and out of Kafka. In general, we can think of grouping our requests that are processed into the following types: produce and fetch requests. While there are many more actual types (like metadata requests), these form a pretty solid base for our understanding of how we have sent messages to and from the cluster so far as clients.

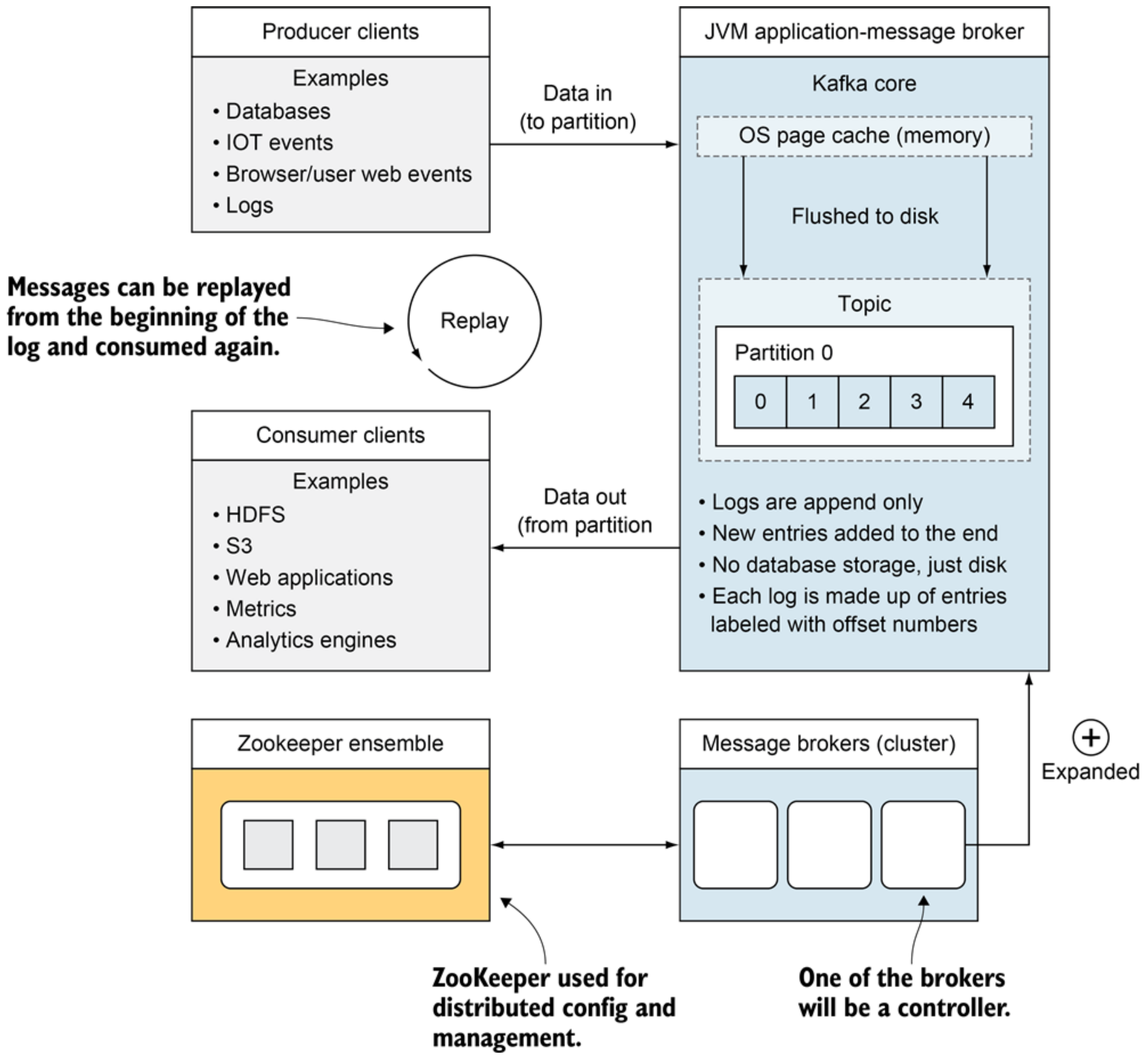

Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Topic
Partition 0
0 1 2 3 4
Messages can be replayed from the beginning of the log and consumed again.
Replay
Consumer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

**Figure 6.1 Brokers**

Produce requests are in fact sent by the producer clients! When a broker starts to handle a request for a lead partition that it owns, validations are performed for access as well as logic around `acks`. The broker will reply with a response to the producer client when the acks configuration is satisfied.

Fetch requests are handled by the brokers and are received from consumer clients as well as from other replicas that are considered the followers of leader partitions. In essence, fetch requests are used by outside consumer in a similar manner of how followers of a partition get the same data.

## 6.4 Configuration at the Broker Level

Configuration is an important part of working with Kafka clients, topics, and also at the broker level. Setting configuration as part of the server properties is a good way to provide sensible defaults for your topics. However, it also provides the flexibility for specific topics to override these default for their own needs. One of the most important and clear examples of a default would be the property: `default.replication.factor`. This configuration property controls the number of replicas that will be used when a topic is created in an automatic fashion. So let's say that you go with three copies of the data. Over one is needed in order to make sure that a single broker failure for that topic doesn't result in a loss of data. On the other hand, let's say that you have specific topics that you value the data as less important in a recovery situation. To make this more concrete, let's think of a website that handles traffic for an online bakery. The order information from customers is considered by the owners as important enough to have multiple copies of: you probably wouldn't want to mess up someone's wedding cake details! However, the section of their website that handles chat history would be something not worth attempting to recover in a failure. Tradeoffs occur in every system, and Kafka allows you to do both reliable and not-as-reliable structures for your topics if you desire. What tradeoffs do we have for always making a bunch of replicas and why wouldn't we just always have more than one copy? The more copies of messages you have the more disk you use over the cluster. Is the extra reliability worth the price of that extra hardware that might be needed? Business value should usually help you with that decision.

Let's take a look at an example of what happens when you have only one copy of your data and that broker goes down.

Make sure that you have your local test Kafka cluster running with three nodes and let's create a new topic with a `replication.factor` of 1.

When you run the following command, you see that there is only one value in the fields: Partition, Leader, Replicas, and In-Sync Replicas (ISR) and that the broker will be the same id. This means that the entire topic depends on that one broker being up. So let's take that broker down and see what happens. You can either kill or Control-C that process if you still have it up in a terminal window. Since there was no copies of your topic, there is no way to keep going without data loss.

### 6.4.1 Kafka's Core: The Log

As with most applications, Kafka provides logs for letting us know what is going on inside of the application. Please keep in mind that there is a difference from the logs where the application logs are kept versus the log files that make up the partitions for the messages themselves. When you start your broker and pass it configuration (usually something like the file server.properties), there are the properties `log.dir` or `log.dirs`. These will be the locations where the logfile data is kept for your partitions. The application log directory, on the other hand, is in your Kafka base directory under the folder `logs/` by default. You could modify this location by listing another directory for `kafka.logs.dir` in a log4j.properties file.

**TIP** **Log Location**

Make sure that you look at your server.properties file configuration before production. The default value has /tmp/kafka-logs. This location is meant to be temporary and is not a good location if you want to do anything but test with the idea that your data will not be permanent.

### 6.4.2 Application Logs

You will want to take a look at some of the logging messages to be able to be familiar with what are normal messages in the life of your application. Here are a few to pay attention to on your servers.

- server.log
- controller.log
- state-change.log

The server log is where you would look if you have a startup error or an exception that terminates your broker. It seems to be the most naturally place to look first for any issues. A dump of the configuration values will be

In addition, your servers that host a controller will also have a controller.log. The controller log file is interesting to look at and see the changes that a controller observed and reacted to. You can also see when a controller was moved from a broker with a message such as the following: `DEBUG [Controller id=0] Resigning (kafka.controller.KafkaController)`. New topics and new partition creation messages will also pepper the log. While the controller usually takes care of itself, if you are working on a restart of the cluster to apply patches or updates, it might be a good place to look to see the controller's actions if you run into any issues. For example, you should be able to see any broker failures and what actions where a result of those failures.

The state-change log is directly related to the actions in the controller log file. Every controller to message broker interaction is modeled as a state change for one or more partitions. This log really gives a view of requested changes from the controller and what the broker did in response. In other words, any decisions that came from the controller should be listed in this log.

With the focus usually on the disk size of your actual message log files, it is important to not forget about these files as well. Look at your retention and see what you really need to keep as you can fill a disk randomly if you keep going without ever grooming these application logs. `logrotate` is a tool that I have used for log rotation and compression but there are many other tools available as well. In addition, it is often a good idea to have these logs on a different disk altogether than the one you have your message logs on. The less contention for writing to disk the better for your overall performance.

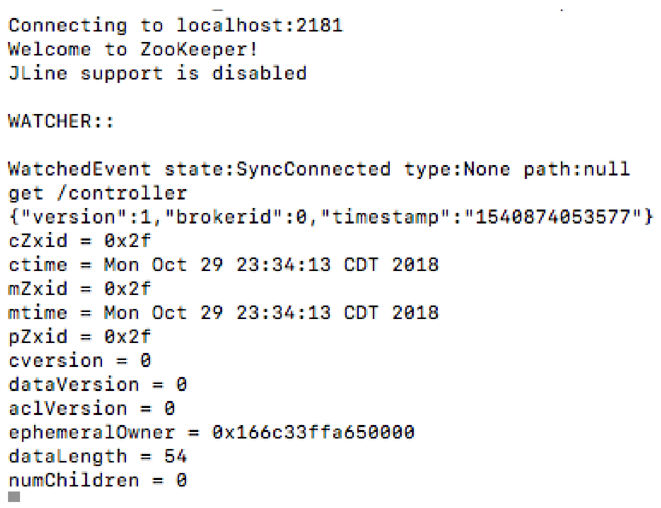
Something else to mention in regards to these logs is that they are located on each broker. They are not aggregated by default into one location. Various platforms might do this on your behalf if you are in a cloud environment in which the broker might not exist when you are trying to analyze logs.

## 6.5 What Controllers are for

While partition have a broker that will be a leader, it is important to note that a specific broker can be a leader for multiple partitions. However, within a cluster of brokers, one of the brokers will act as a controller. The role of the controller is to manage the state of partitions and replicas. The controller also will perform other administrative actions like partition reassignment.

The controller leverages ZooKeeper in order to detect restarts and failures. If a broker is restarted, a controller will send the leader information as well as the ISR information for the broker re-joining the cluster. For a broker failure, the controller will select a new leader and update the ISR. These values will be persisted to Zookeeper. In order to coordinate all of the other brokers, it also sends the new leader and In-Sync Replicas changes to all other brokers in the cluster.

When you talk about shutting down a cluster for an upgrade, it is important to know which broker is serving as the controller. One thing to note is that you would not want to keep shutting down the controller. This could happen if you shutdown a controller which would cause a new controller to start on another broker. Then, when it is time for the next shutdown, you will be stopping the controller that just moved. In other words, we would try to avoid the overhead with forcing the controller to move and startup on each broker if we can avoid it. The controller failover process is able to recover due to having data into Zookeeper.



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

WATCHER::

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

**Figure 6.2 Example Controller Output**

To figure out which broker is the current controller, you can use the zookeeper-shell script to look up the id of the broker as shown in Listing 6.2. The path `/controller` exists in zookeeper and we are running one command to look at that current value. Running that command for my cluster showed my broker with id 0 as the controller. Figure 6.2 shows all of the output from Zookeeper and how the brokerid value is returned.

**Listing 6.2 Listing our topics**

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

❶ Connect to your zookeeper instance

❷ Use `get` against the controller path

There will also be a controller log file with the name: `controller.log` as we discussed application logs in the previous section. This will be important when you are looking at broker actions and failures. The `state-change.log` is also useful as it will tell you the decisions that it received from the controller.

## 6.6 Leaders and their role

One of the replicas of the partition will have the job of being the leader. The leader is in charge of handling the writes for that partition. Since this leader is the only one with the new written data, it also has the job of propagating writes to the followers. The In-Sync Replicas (ISR) list is maintained by the leader. Thus, it knows which replicas are caught up and see all the current messages. The replicas will act as consumers of the leader partition and will fetch the messages and have offset checkpointing to track their consumption.

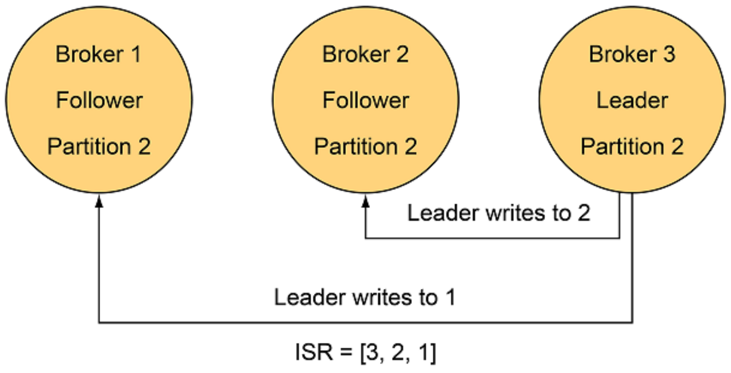


**Figure 6.3 Leader**

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

Since we are dealing with distributed computing and we know failures are expected, the leader also can notice a slower follower replica and remove it from the ISR list.

In some cases, the broker that fails could be the leader partition. The Controller and ZooKeeper could be used in this case to changes the leader for that partition. In Figure 6.4, our previous example has experienced a failure. Since this broker is not available, a new leader is elected. This new leader in this example was Broker 2. Once a follower, it was moved to a leader to keep Kafka serving and receiving data for that partition.

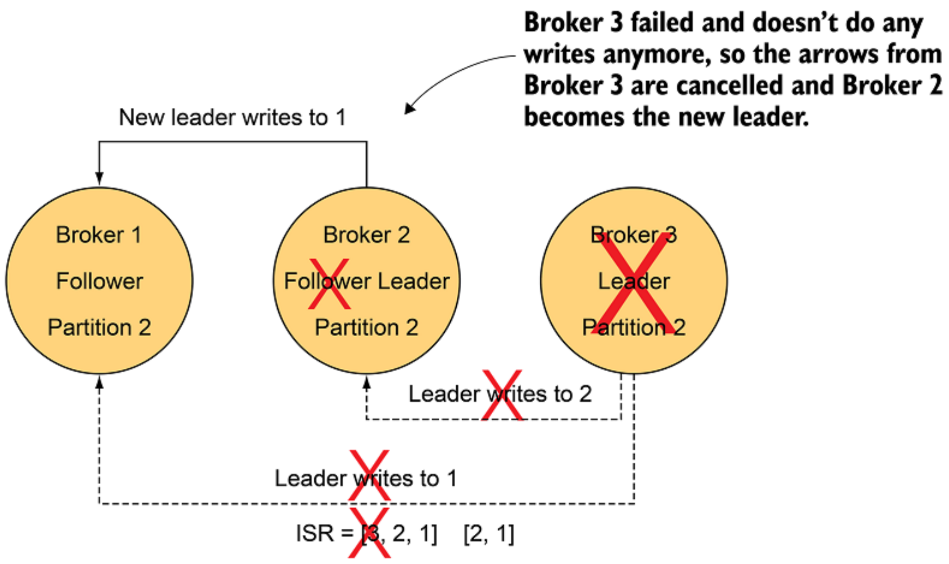


**Figure 6.4 New Leader Elected**

It is interesting to note that there is a Kafka Improvement Proposal KIP-392 ( cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replic ) which would allow consumers to fetch from the closest replica. Reading from the leader only is something that might take more time if your brokers span physical data centers and Kafka might be updated to help with these situations in the future.

### 6.6.1 Inter-Broker Communications

One of the important things to know is that the more partitions a broker holds as a leader will increase the fail-over time. As a reminder, topics are made of partitions that may or may not be replicated. When you have a replicated partition, that means that a client has written to the leader and for the other brokers to see that message, it has to get a copy from the lead broker.

### 6.6.2 The Role of Replicas

Since one brokers holds the leader for a specific partition, what does the replica brokers do for the cluster? They can not be written to by producers or even read from by consumers. Their most important task is to try to keep up with the leader partition by fetching the data from the leader and making it available in its own logs. This is in case of a failure or new leader election takes place in which the replica might before the new leader for that partition. The replica job does not have as much client interaction, however it will help in many ways if failure occur to keep the data moving in our cluster.

It is important to note that if a replica starts to get too far behind in copying messages from the leader, it might be removed from the list of in-sync replicas (ISR). This means that the leader notices that a follower is taking too long and will drop it from its list of followers. The leader will continue to operate with a new ISR list that is smaller and one node replica away from better fault tolerance.

## 6.7 In-Sync Replicas (ISR) Defined

In-Sync Replicas (ISR) is a key part to know about Kafka. When you define a new topic, a specific number of replicas will be created. This number can either be from a parameter or as a default from the broker configuration.

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

Why is watching this number so important? Let’s say that you have a topic that is only 1 partition. That partition is replicated 3 times. In the best case scenario, you would have your 2 copies of your data match the data that was on your lead partition. This of course means that they other logs are caught up with the leader. If they are lagging behind, that is a situation that could lead to a replica falling out of the ISR list. If you really need all of your data and can't miss a single message, you can see the issue with out of sync replicas. If you lose one replica and both of the other replicas are not up-to-date with the current data then you are going to be missing data.

## 6.8 Unclean Leader Election

One important tradeoff to think about is whether uptime is important vs the need to keep our data from being lost. If `unclean.leader.election.enable` is set to true, that allows the controller to elect a leader for a partition even if it is not in sync. The biggest issue with this is that data could be lost. While data loss might not be okay for payment transactions, it might be okay for a web analytics engine to miss a hover event on a couple of pages. While `unclean.leader.election.enable` is a cluster-wide setting, you can override this configuration per topic.

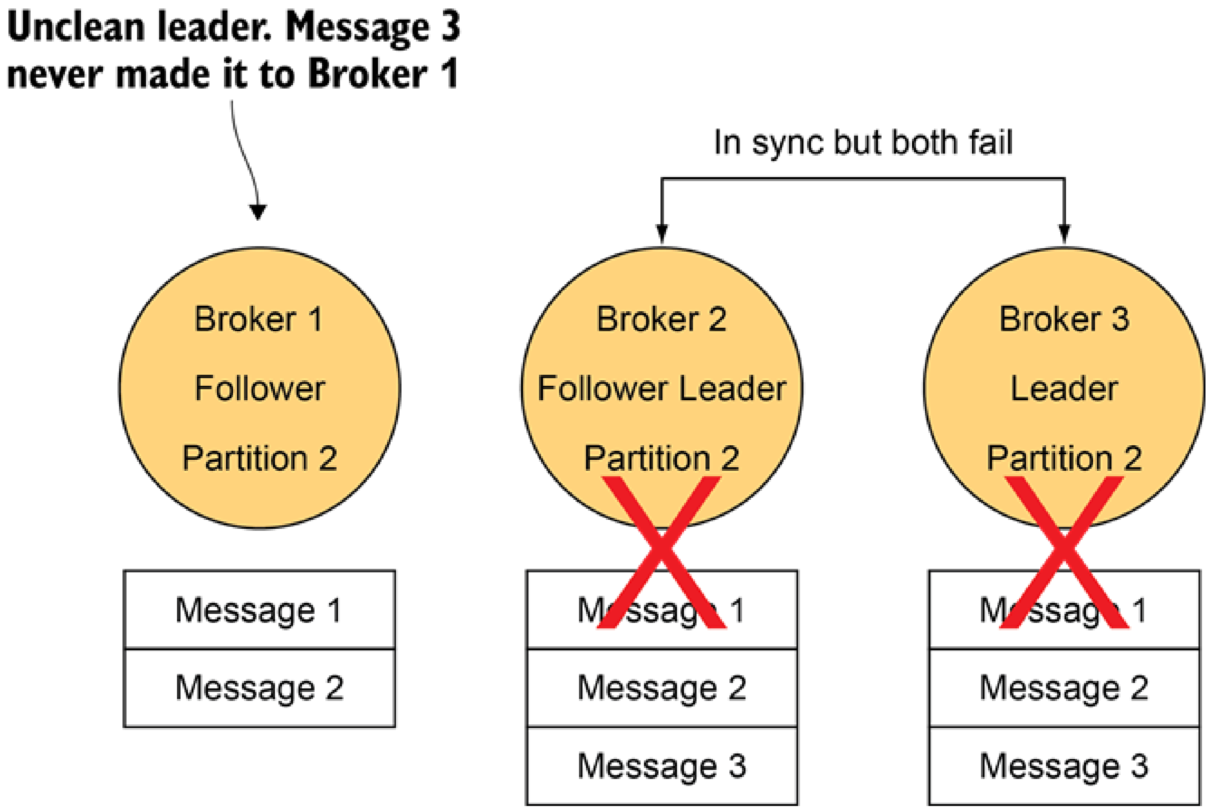


**Figure 6.5 Unclean Leader Election**

Figure 6.5 shows data lose in the case of a partition that has a total of 3 replicas. In this case, both Broker 3 and 2 failed and are not online. Since unclean leader election was allowed, Broker

1 is made the new leader even though it was not in-sync with the other brokers. For the price of missing data, this option might allow you to keep serving clients if your business case will allow this tradeoff.

## 6.9 Seeing Metrics from Kafka

Kafka, written on for the JVM, uses Java Management Extensions (JMX) so we can peek into the Kafka process. There are many options to capture this data from our applications. We will use Grafana (grafana.com/) and Prometheus (prometheus.io/) to set up a monitoring stack that will allow us to have a visual of what is going on in our cluster. We can use Prometheus which will be our data store that allows us to extract metrics from Kafka. In order to produce a visual of these metrics, it is pretty easy to send this information into Grafana and create our own graphs using this information.

To make sure we know why we are setting up all of the following tools: let's quickly make sure we know what parts are doing which work in Figure 6.6.

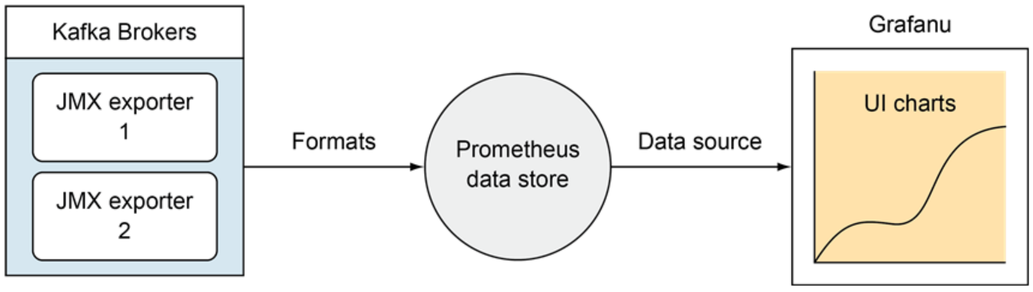
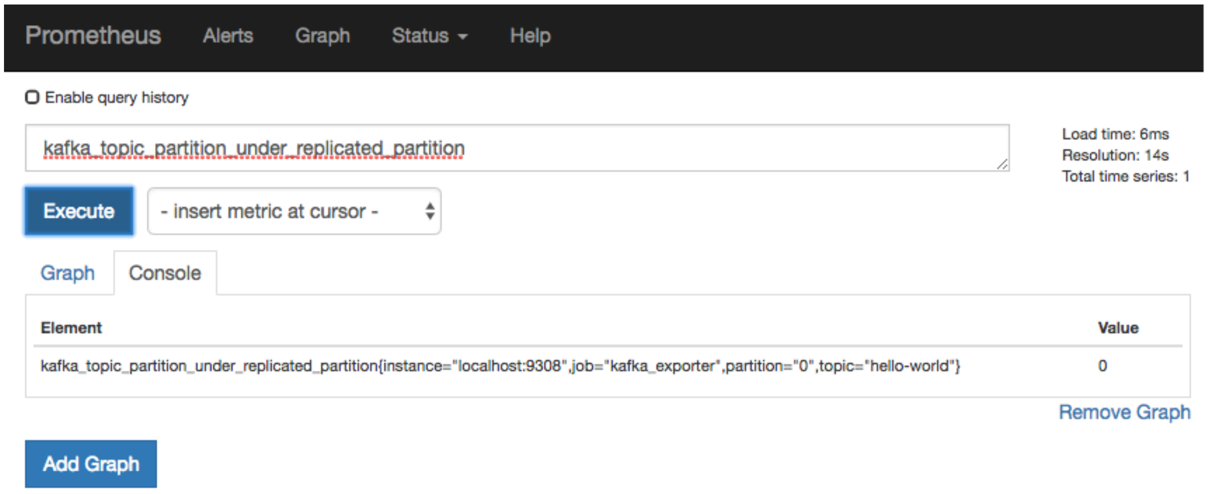


**Figure 6.6 Graph Flow**

- JMX is coming from our Kafka applications.
- The Kafka exporter is taking the JMX notifications and exporting them in the Prometheus format.
- Grafana is taking the information from Prometheus and displaying that information in a visual dashboard. If you are familiar with Docker, there are many images and docker-compose files that have all of these tools bundled up. However, I am only going to install each tool local in order to know the steps above in greater detail.

For the Kafka exporter, let's work with a tool available at github.com/danielqsj/kafka_exporter. I prefer the simplicity of this tool since you really run it and give it one (or a list) of Kafka servers to watch. Notice that we will get both client and some broker specific metrics since there are quite a few options that can be helpful to monitor. However, this is not a complete list of all the metrics available to us.

Figure 6.7 shows a local instance of Prometheus that is gathering metrics from our kafka_exporter tool port. As we talked about in Section 6.8 in regards to Kafka replicas not healing themselves, one of the things we can monitor is unreplicated partitions. If this number is above 0, then we might want to look at what is going on in the cluster and why there is a replica issue.



| Element | Value |
|---|---|
| kafka_topic_partition_under_replicated_partition{instance="localhost:9308",job="kafka_exporter",partition="0",topic="hello-world"} | 0 |

**Figure 6.7 Prometheus Example**

Figure 6.8 shows using the Prometheus instance from the previous figure and using it as a data store for Grafana. It is looking at the same metric as before, except in a graph.

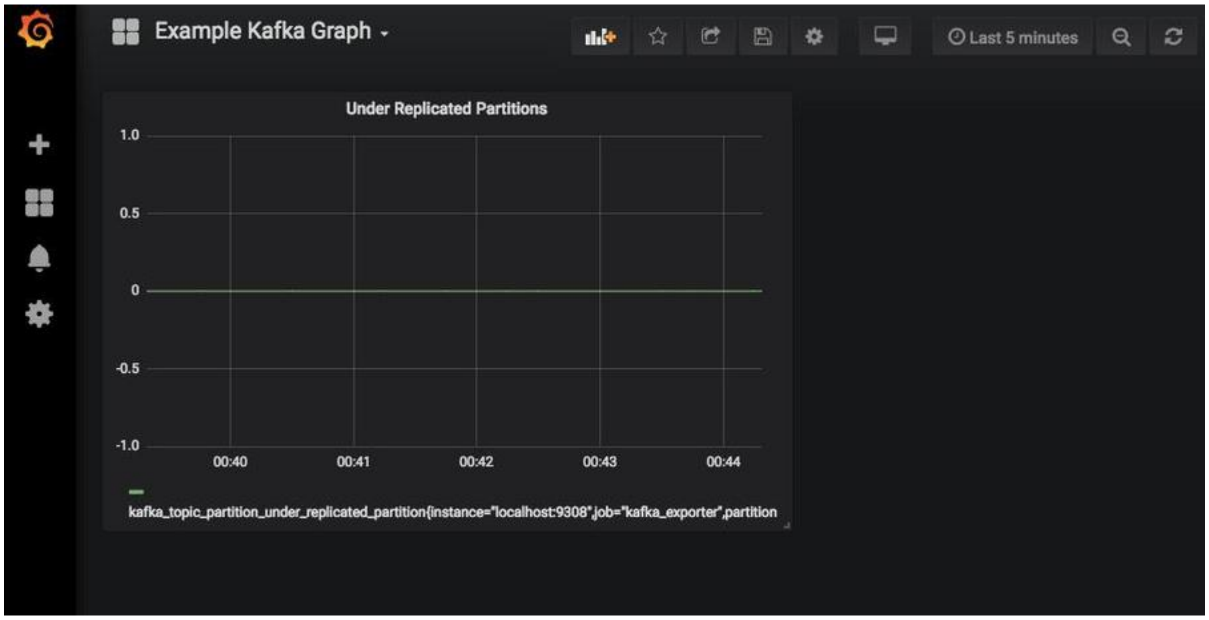

Example Kafka Graph
Last 5 minutes
Under Replicated Partitions
1.0
0.5
0
-0.5
-1.0
00:40
00:41
00:42
00:43
00:44
kafka_topic_partition_under_replicated_partition{instance="localhost:9308",job="kafka_exporter",partition

**Figure 6.8 Grafana Example**

To get more JMX metrics, one way is to expose set the `JMX_PORT` when we start our Kafka processes. Other tools might require you using a Java agent in order to produce the metrics to an endpoint or port that Prometheus can check for information.

**Listing 6.3 Starting a broker with a JMX Port**

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

❶ Adding the JMX_PORT variable when starting the cluster

Listing 6.3 shows setting the variable `JMX_PORT` when starting a broker. If you already have a broker running and do not have this port exposed, you will likely want to automate setting this variable on restarts of the brokers in order to make sure this is enabled if desired for future restarts.

### 6.9.1 Cluster Maintenance

As many people look at moving to production, we will want to make sure that you are on more than one server. In reality, there is really nothing wrong with starting small. The recommendation would be to have 3 brokers and 3 zookeeper nodes for production workloads.

Another pattern you might want to think about is whether you want one giant cluster for your entire enterprise, or one per product team. Some organizations (like Linkedin where Kafka was created) have thousands of brokers as part of one cluster. Some teams prefer to keep smaller clusters that power their smaller product suite. There are many ways to handle a production cluster and both methods listed above can work.

Another item to note is that Kafka clients, Connect clients, and the REST Proxy do not usually run on the same servers as the brokers themselves. While we might be running all of these exercises on our laptop and show that it can be done, we probably don't want all of those processes running on the same server if we are trying to handle production workloads without delays.

To draw a parallel to similarities with tools from the Hadoop ecosystem, Kafka can scale well horizontally with commodity servers. While 'commodity' can mean different things, I would say that it should not be related to 'cheap'. Let's look at what it takes to add a server to a cluster.

### 6.9.2 Adding a Broker

Starting with a small cluster is a great way to start as adding a broker is an option for growing your footprint. To add a Kafka broker, you need to start-up your new Kafka broker with a unique id. That is pretty much it! But, there is something to be aware of in this addition. The broker will not be assigned any partitions. Any topics that were created before this new server was added would still exist on the brokers that they were created on as nothing is moved automatically. If you are okay with the broker only being added for new topics that are created, then you do not need to do anything else. However, if you do want to change the existing layout, you can look at using the tool: `kafka-reassign-partitions`. This script is located in the `bin` directory of your Kafka installation. We will cover how to use this tool further in Chapter 7. There are also other tools that have been developed to help with this situation. One such alternative is the Kafka Manager (github.com/yahoo/kafka-manager) tool that also has other options to help you manage your cluster.

### 6.9.3 Upgrading your Cluster

As with all software, updates and upgrades are a way of life. Not all systems can be bought down all at once and upgraded due to production workload or business impact. One scenario to look at for Kafka in order to avoid downtime for your applications would be a rolling restart. This really means just restarting one broker at a time.

One thing to make sure you set in your broker configuration for your first deployment is that you have the property `controlled.shutdown.enable` set to 'true'. This setting helps transfer partition leadership before the broker is shutdown. Another setting to be aware of is `min.insync.replicas`. Since Kafka does not automatically create a new partition replica on a different broker, you will be short one partition from your in-sync-replica total. If you start with 3 replicas and you remove a broker that hosts one of those replicas, a minimum replica number of 3 will be an issue. Clients would not be able to continue working and would be stuck waiting for the broker to rejoin and sync up. You need to make sure that you are aware of the numbers you require in regards to the number of brokers you are willing to tolerate losing.

Another good practice is to identify what broker is serving as your controller in your cluster and leave it for the last server to upgrade. We want to do as least work as possible and not having the controller bounce between brokers helps keep our restarts faster.

Why else besides broker upgrades might you want to perform a rolling restart? Updating broker configuration sometimes requires a restart of Kafka. Applying security patches for your OS as well as other server maintenance tasks might be another reason to have an upgrade/maintenance plan in place.

And don’t forget about the version of Java you are using! Kafka and Zookeeper both depend on Java to run and Java itself might have security patches that will need to be applied. This will require broker restarts and need to be part of your overall cluster maintenance plans.

### 6.9.4 Upgrading your clients

While Kafka does its best to decouple the clients from the broker, it is always a good idea to check what version you clients are when trying to connect to various broker versions. For example, clients can usually be upgraded after all the Kafka brokers can been upgraded.

This is also the option to use newer clients without worrying about the broker version. This bidirectional client compatibility feature was a new feature in Kafka 0.10.2. It is supported by brokers that are at version 0.10.0 or later. As with any upgrade though, take a peek at the version notes to make sure you will have compatibility for newer versions.

### 6.9.5 Backups

Kafka does not really have a backup strategy like a database table. You don't take a snapshot or disk backup per se. Usually, a cluster is backed by a separate second cluster and events are replicated between topics. One of the earliest tools that you will see in the wild is called MirrorMaker.

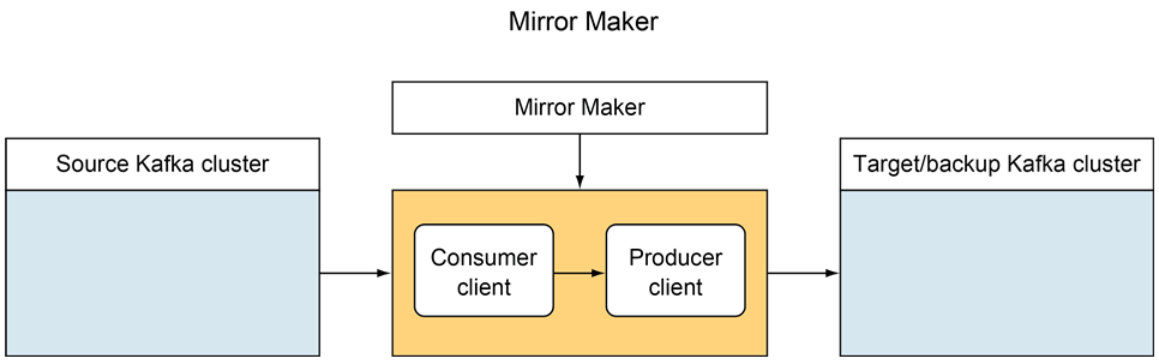


**Figure 6.9 MirrorMaker**

This tool is basically a consumer and producer hooked together that can act as a standalone tool. Figure 6.9 shows how the source cluster is read from with a consumer client and then written to the backup cluster with a producer client. The simplicity of the tool has some tradeoffs. Since the clusters are separate, data is taken from the original cluster and written to a topic with the same name on the second cluster. This independence includes offsets that could be a different index number. One detail to note is that the event's key will be the same so partitioning (and thus ordering) would be correct if your messages did use a key.

When you run the MirrorMaker tool from a terminal, shown in Listing 6.4, you will be passing it configuration for the source and destination clusters. The `consumer.config` file will hold the zookeeper server as well as the bootstrap servers for the message brokers for the cluster that you want to be the source of your data. The `producer.config` file will hold similar information for the destination or copy cluster. You can also set parameters like the key and value serializers or any other producer properties you would want to set if you were writing the client yourself.

**Listing 6.4 MirrorMaker**

```
bin/kafka-mirror-maker.sh   ❶
      --consumer.config source.properties   ❷
      --producer.config target.properties --whitelist test-topic   ❸
```

❶ Shell script to start mirror maker

❷ Passing configuration to the consumer used in mirror maker

❸ Passing configuration to the producer used in mirror maker

There are also some other open source as well as enterprise offerings for mirroring data between clusters. Confluent Replicator has a proprietary license but also some pretty nice features. This includes support for adding new partitions. Whereas MirrorMaker will not update on its own, Confluent Replicator should automatically detect and mirror those partitions. Topic configuration would also be replicated between the two clusters with this tool. uReplicator, which was created by Uber, also seems to be a popular option that I have seen used for clusters as well.

## 6.10 A Note on Stateful Systems

One of the ongoing trends for many technical companies is the drive for applications and infrastructure living in the cloud. Kafka is an application that definitely works with stateful data stores. In this book material, we will be working on our own nodes and not with any cloud deployments. Instead of working with details related to Kubernetes, Apache Mesos, Docker, and AWS, we will try to keep focused on Kafka and just know that many companies do work with Kafka in these environments. There are some great resources including Confluent's site on using Kubernetes Operator API www.confluent.io/confluent-operator / as well as Docker images available to do what you need. If you are familiar with the above tech tools, it might be a quick way for you to kick the tires on a Proof of concept (PoC) setup if you find something out in Docker Hub. Figure 6.9 shows a general outline of how Kafka brokers are setup in Kubernetes. Each broker will be in its own Pod as well as Zookeeper in its own Pod as well.

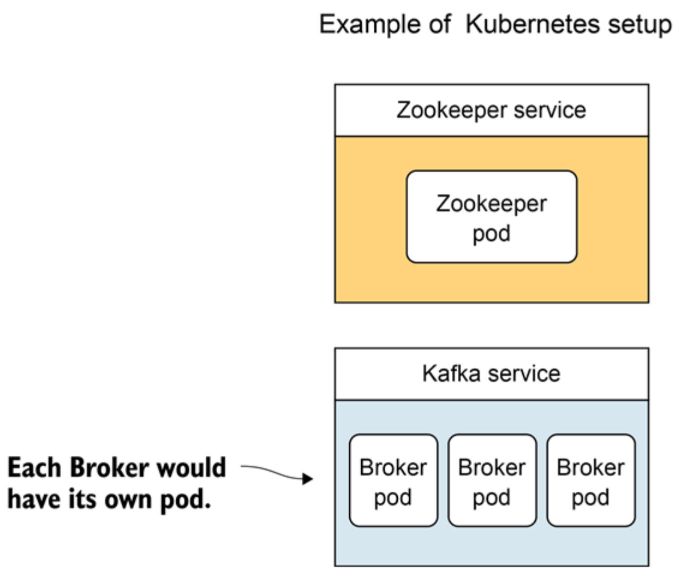


**Figure 6.10 Kubernetes on Kafka**

As with any production system, there are tradeoffs and not a one-size fits all mandate for your infrastructure. So let’s talk about some reasons that you might have Kafka in a cloud or Kuberentes enviroment: using Kuberenetes as a focus point to narrow our discussion. If you have a majority (if not all) of your applications in a specific setup, it might make sense to keep that setup for your Kafka environment. If Kubernetes runs your shop, it might not be a good idea to do something different for your Kafka setup. Another benefit of Kubernetes that stands out would be the ability to create new clusters quickly. Depending on your company, if each product owns its own cluster, it might be easier to manage than if your enterprise only has one huge cluster for the entire company.

In addition, versions of Kubernetes since 1.9 seem to have stateful as a core part of the product and not a work-around. One idea that I have seen is to run Kafka clients on a Kubernetes cluster before your actual message brokers. Besides being stateless, running your clients in this manner can help you get a feel for Kubernetes if you are just getting started in your journey.

## 6.11 Exercise

Since it can be hard to apply some of our new learning in a hands-on manner since this chapter is heavier on commands than code, it might be good to reflect on the impacts of Kafka on your application.

- Let’s say that you notice one of your topics that you created was suppose to have 3 replicas. In this case, you would want to make sure you have 3 brokers listed in the ISR list in case there was a failure. What command would you run to look at that topic and see the current status?

To start, let’s confirm the topic we are trying to look at is setup with the ReplicationFactor we assumed it was supposed to. Our results look like 3 was correct. And you can see that the Isr is only 2 brokers! That should be a list of 3.

**Listing 6.5 Kafka topic command**

```
    $ ./kafka-topics.sh --describe --zookeeper localhost:2181 --topic replica-test    ❶
Topic:replica-test    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: replica-test    Partition: 0    Leader: 0    Replicas: 1,0,2    Isr: 0,2    ❷
```

❶ Note the topic parameter and the describe flag being used

❷ Topic specific information about leader, partition, and replicas

You could have also used the --under-replicated-partitions flag in order to see any issues quickly!

**Listing 6.6 Under-replicated-partitions flag**

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --under-replicated-partitions    ❶
    Topic: replica-test    Partition: 0    Leader: 0    Replicas: 1,0,2    Isr: 0,2    ❷
```

❶ Note the under replicated partition flag being used

❷ Notice that the Isr result only lists 2 brokers!

We will explore more of the tools included with Kafka out-of-the-box when we talk about admin tools in Chapter 9. When using any of the commands in this chapter, it is always a good idea to run the command without any parameters and read the command options that are available for troubleshooting.

As we explored more about Kafka in this chapter, it is important to note that we are running a complex system. However, we have various command line tools as well as JMX metrics we can use to figure out the health of our cluster. In our next chapter, we will continue to use commands to rebalance our partitions as well as maintain this complex system throughout its life.

## 6.12 Summary

In this chapter you learned:

- About the roles of leaders, zookeeper, ISRs, and the brokers themselves
- To plan for and upgrade a cluster
- How to set up graphs to visually monitor our cluster

# 7 Topics and partitions

***This chapters covers:***

- Various topic creation parameters
- Log files written as partitions on disk
- How to see the content of those logs
- Topic compaction

In this chapter, we will look further into how we might want to store our data across topics as well as how to create and maintain the life of a topic. This will include how partitions fit into our design considerations as well as how we can view our data on the brokers. All of these specifics will help us also dig into how your topic can start to appear to show data like database tables that update data rather than append!

## 7.1 Topics

To quickly refresh our memory, it is important to know that a topic itself is more of a logical name rather than one physical instance. It does not usually exist on only one broker. In my experience, most of your consuming applications of your data will think of the data being related to a topic: no other details needed for them to subscribe. However, behind the topic name, one to many partitions actually make up a specific topic. The logs that are written to the broker filesystems that make up a topic are the result of Kafka actually writing the data in the cluster. Figure 7.1 shows a three partition topic that makes up one topic named 'helloworld'. A single partitions copy is not split between brokers and has a physical footprint on each disk. Figure 7.1 also shows how those partitions are made up of messages that are sent to a topic.

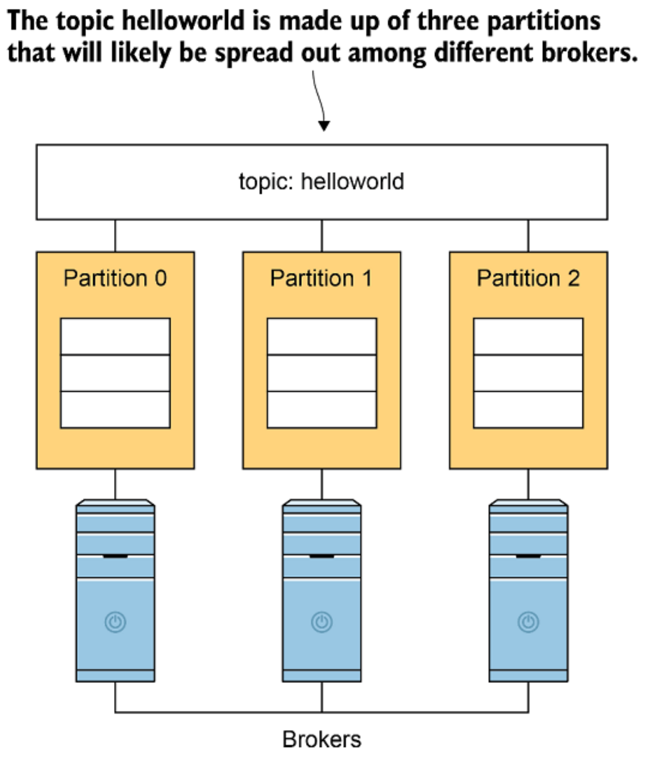

The topic helloworld is made up of three partitions that will likely be spread out among different brokers.
topic: helloworld
Partition 0
Partition 1
Partition 2
Brokers

**Figure 7.1 Example Topic With Partitions**

If writing to a topic can be so simple in getting started examples, why do we want to make sure that we understand the role and pieces that make up a topic? At the highest level, how you consumers get to the data will be impacted. Let’s say that your company is selling reservations for a hotel using a web-based application that sends events of user actions into your Kafka cluster. Your overall application process could generate a number of events. For example, there would be an event for the initial search on location, one for the specific reservation actually being selected by the customer, and a third for payment being received. Should the producing applications send all of this data to a single topic or a number of topics? Should you consider that each message is a specific type of event and should remain separated into different topics? There are tradeoffs of each approach and some thoughts that will help us form what each approach you can take with your own situations.

One common question I often still get is along the following lines: "My cluster is X many nodes, what is the number of partitions I should use?" The biggest takeaway is that partitions are a per-topic design question, not a cluster-wide limitation or mandate. While there can be a default number used in topic creation, concern should be taken to really think about the needs of your data. You should have a reason why you pick a number of partitions. Let’s say that you want to have a partition for server as a rule. However, note that just because you have a partition on a server, does not mean producers will write evenly between them. In order to do so, you would have to make sure that you have each partition leader spread out in that manner as well and stays that way. You need to get familiar with your data! Lets take a look at a checklist of items to think about in general and in this scenario:

- Data correctness

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

- Volume of messages of interest per consumer

One item to consider is the also the volume of messages of interest per consumer. For the hotel rental system above, let's look at the number of events when we consider the topic placement. The search events themselves would outnumber by far the other events. Let's say that a popular hotel would get 50,000 searches a day but only has 100 rooms. Traffic on most days produces 50,000 search events and less than 100 actual 'booked' room events. Would the Billing team have an application that would want to subscribe to a generic event topic in which it used or cared about less than 1% of the total messages? Most of the consumers time would be in effect filtering out the mass of events to process only a select few.

- How much data you will be trying to process

Another point to consider is how much data you will be trying to process. Will that number of messages require multiple consumers to be running in order to process in a timely manner as required by your applications? If so, we have to think about how the number of consumers (active client consumers in a a consumer group limit) is determined by the numbers of the partitions for your topic. In general, it is easier at this point to create more partitions that you think you might absolutely require. Having more capability, as far as allowing consumers to grow, will allow you room to grow with having to deal with re-partitioning data. However, it is important to remember that each partition is not an unlimited free resource. More partitions require more open file handles for example. Also, more brokers to migrate in case of a broker failure could be a headache in the future. Try to find a happy medium and go with it for your design as you start. Figure 7.2 shows how our design might be best suited with two topics for the three event types we used in our scenario. As always, more requirements or details could change our implementation in the future.

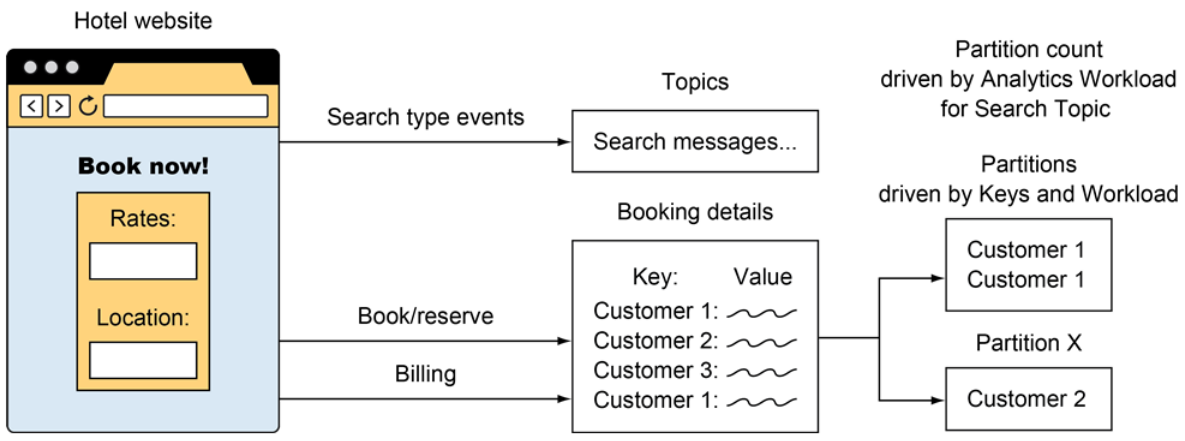


**Figure 7.2 Example Hotel Event Topic Design**

With the above details on topic design, one thing to note that you really do not have to do is reduce partition counts for a topic. While you might be able to figure out a way to do it without pulling your hair out, it is not really a supported way to manage your topic. Using your knowledge of what we have talked about so far: what do you think some of the issues might be? While consumers subscribe to a topic, they really are attached to a partition. The removal of a partition would lose context of the message order and out-of-order messages could occur when/if a consumer starts reading from a new reassigned partition. It reminds me of a parallel of how commit history on a public Git repository can seem inconsistent for clients after a rebase. If a developer rebases, the branches have to be merged together later. This results in two sets of commits and history has changed! History is a big deal for source control systems! While not exactly the same data impacts, the client view has a harder time following the upstream changes. This is where you need to make sure your keyed messages and consuming clients are able to follow any changes you make at the broker level. You can impact consumers with your actions!

Since we talked a little about how to design topics, lets start with looking at some of the options on actually creating topics that we only touched on briefly when we created topics to produce messages to in Chapter 3.

### *7.1.1 Topic Creation Options*

Kafka topics have a couple of core required options you have to complete to create a topic. These include the following:

- Zookeeper connection: The host and port of a zookeeper node in your cluster
- Topic name: What you are going to call you topic to describe your data
- Replication Factor: How many copies (of the partitions) you want to have in case of failures
- Partitions: The number of partitions which will be informed by your analysis above

The trick for some of these values such as replication-factor is that you can have cluster-level defaults. However, if you are being purposeful about your data, I would treat these decisions with thought and care. Defaults might be a good catch-all to make sure topics are created with certain levels of failover (like always have 3 replicas of your data). Table 7.1 lists some other unique creation options.

**Table 7.1 Other Topic Creation Options**

| Option | Purpose |
|---|---|
| if-not-exists | Action will only execute if the topic does not already exist |
| replica-assignment | Manually set partition-to-broker assignments. Formated value of 1:2,1:2 would set a 2 partitioned topic on brokers 1 and 2. |

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

**Listing 7.1 Listing our topic options**

```
kafka-topics.sh /  ❶
```

❶ Running the generic kafka topic related command

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

**Listing 7.2 Listing our topic options**

```
kafka-topics.sh --create  ❶
```

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

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

As we think of a name to represent our data, there are a couple of rules that might help us avoid random problems later. Like in Linux in general, I would never recommend naming files or directories with spaces. Another gotcha that we touched on when looking at Kafka managing client consumer offsets in topics itself, we saw that an informal standard exists that internal topic names are the ones that start with two underscores. To avoid any confusion, I would avoid that convention for your own topics. In addition, I would pick either underscores or periods but not both. You are even prompted with a warning when you create a new topic about metric names being a cause for your to be consistent in your topic naming conventions.

Before we create a new topic, you can take a peek of what is already out in the cluster by running the kafka-topics script with the `--list` command. This will show you topics that you have already created and help you avoid any naming collisions in your testing. Listing 7.3 shows the complete command to get a list of all the topics on your brokers.

## Listing 7.3 Listing cluster topics

```
kafka-topics.sh --zookeeper localhost:2181 --list   ❶
```

❶ The --list parameter will show all topics on the cluster

After you have thought of a name that does not conflict with any existing topics, you can create it. For our examples, I am going to create a topic called 'my_test_topic' with a replication factor of two and with two partitions. Listing 7.4 shows the syntax in to be used in your command prompt.

## Listing 7.4 Creating a topic

```
kafka-topics.sh --zookeeper localhost:2181 --create \   ❶
--topic my_test_topic \   ❷
--replication-factor 2 \    ❸
--partitions 2   ❹
```

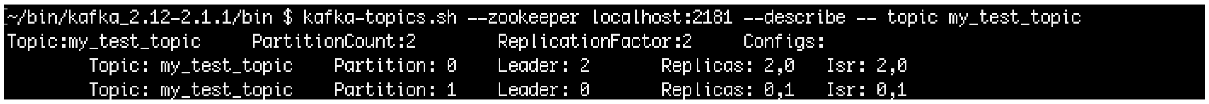
❶ Adding the create option to our command

❷ Naming your topic

❸ Making sure we have 2 copies of our data

❹ Declaring our topic will consistent of 2 partitions

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



```
~/bin/kafka_2.12-2.1.1/bin $ kafka-topics.sh --zookeeper localhost:2181 --describe -- topic my_test_topic
Topic:my_test_topic     PartitionCount:2        ReplicationFactor:2     Configs:
        Topic: my_test_topic    Partition: 0    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: my_test_topic    Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0,1
```

**Figure 7.3 Describing Topic With 2 Partitions**

Notice in figure 7.3 how your partition and replication-factor should match the command you just ran!

Another option that I prefer to set at the broker level is setting the configuration `auto.create.topics.enable` to false. Doing this makes sure that you are creating topics on purpose and not by a producer sending a message to a topic name that was mis-typed and never actually existed (before a message was attempted). While not very tightly coupled, producers and consumers do usually need to know the correct topic name of where their data should live. This automatically topic creation can cause confusion. However, note if you are creating topics in a fury of testing and playing with Kafka, you might find value in looking at relaxing that suggestion.

For a concrete example, if you run the command `kafka-console-producer.sh --broker-list localhost:9092 --topic test` without that topic existing, Kafka can create that topic for you based on your configuration above. When you run `kafka-topics.sh --zookeeper localhost:2181 --list` you would now see that topic in your cluster! It will use the broker-level defaults for the required fields you had to enter if you created them manually as we did earlier.

If you were in a hurry or ran a new create topic command without changing the partition count that you really wanted, we can change use the `--alter` option to change that detail about your topic. Altering the topic right away before you add data is the easiest option by far. However, if you have data in those partitions already, the story becomes more complex. What if you don't have a concern about messages key ending up on a specific partition? Still, in this case, it is important to note that addition of a partition will only even see messages after it is created. In other words, Kafka does not automatically move existing data on your behalf.

If your consumer depend on the message keys in order to partition data, your options might be a little more nuanced. One option is to create a new topic, republish keyed messages to the new topic and have your consumers read from the new, rather than the older topic. We will also talk about using a tool provided with Kafka called `kafka-reassign-partitions` in a later section. Listing 7.5 shows the alter command that we would use to update an existing topic to 3 (from 2) partitions.

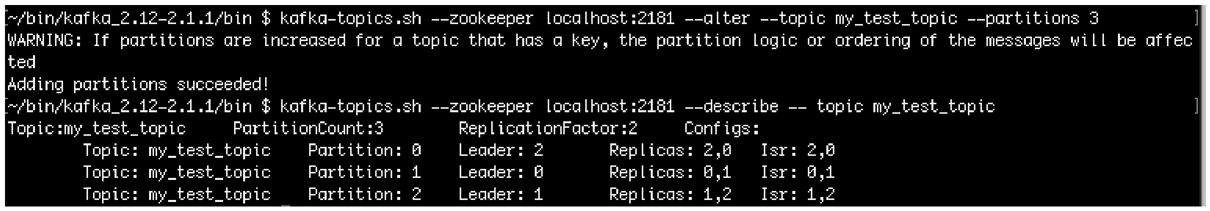
## Listing 7.5 Adding more partitions to an existing topic

```
kafka-topics.sh --zookeeper localhost:2181 --alter \   ❶
--topic my_test_topic --partitions 3   ❷
```

❶ Alter is needed since we are updating an existing topic

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

You will see the PartitionCount field value as: 3 and Partitions 0, 1, and 2 listed. Notice that even the command output reminds you how keyed messages could be impacted by your change! It is always a good reminder sanity check to make sure you are aware of your impacts to your clients.



```
~/bin/kafka_2.12-2.1.1/bin $ kafka-topics.sh --zookeeper localhost:2181 --alter --topic my_test_topic --partitions 3
WARNING: If partitions are increased for a topic that has a key, the partition logic or ordering of the messages will be affec
ted
Adding partitions succeeded!
~/bin/kafka_2.12-2.1.1/bin $ kafka-topics.sh --zookeeper localhost:2181 --describe -- topic my_test_topic
Topic:my_test_topic     PartitionCount:3        ReplicationFactor:2     Configs:
        Topic: my_test_topic    Partition: 0    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: my_test_topic    Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: my_test_topic    Partition: 2    Leader: 1       Replicas: 1,2   Isr: 1,2
```

Figure 7.4 Adding Another Partition With Alter

Running a describe command against your topic should show, like in Figure 7.4, how your partition you just added shows up after the alter is complete.

## 7.1.2 Removing a Topic

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

**Listing 7.6 Deleting a topic**

```
kafka-topics.sh --zookeeper localhost:2181 --delete \   ❶
--topic my_test_topic
```

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

After running this command you will likely see a note that the topic is marker for deletion. Since this operation is a pretty big deal that we want to make sure can not happen without a logic checkpoint, Kafka requires us to configure all of the brokers with the configuration option `delete.topic.enable`. If this is set to 'true' we will be able to successfully delete the topic and it will be removed.

## 7.1.3 Replication Factors

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

## 7.2 Partitions

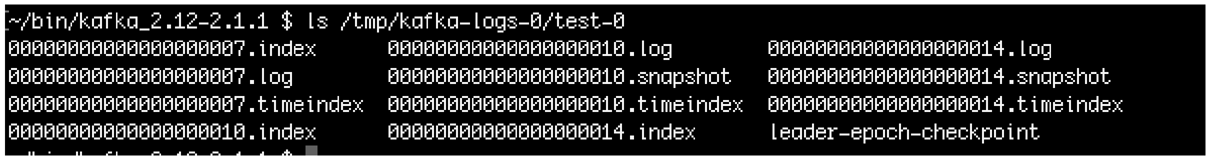
Moving on from dealing with Kafka commands at a mostly topic level, lets start to look deeper at partitions. From a consumer standpoint, I like to think of each partition being an immutable log of messages. It should only grow and append messages to our data store. While this data does not grow indefinitely in practice, thinking of the data being added to, rather than modified in place, is a good mental model to maintain. Consumer clients can not directly delete messages. After all, we understand that replaying messages from a topic is a feature that can help us in various scenarios of data consumption.

### 7.2.1 Partition Location

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

- .index
- .log
- .timeindex

Figure 7.5 shows you how partition 0 in my test topic looks by doing a directory listing.



```
~/bin/kafka_2.12-2.1.1 $ ls /tmp/kafka-logs-0/test-0
00000000000000000007.index      00000000000000000010.log        00000000000000000014.log
00000000000000000007.log        00000000000000000010.snapshot   00000000000000000014.snapshot
00000000000000000007.timeindex  00000000000000000010.timeindex  00000000000000000014.timeindex
00000000000000000010.index      00000000000000000014.index      leader-epoch-checkpoint
```

**Figure 7.5 Partition Directory Listing**

Sharp-eye readers might see the files named `leader-epoch-checkpoint` and those with a .snapshot extention. In general, `leader-epoch-checkpoint` is used to handle leader failures between replicas and the snapshot files are files that are internal Kafka snapshots that we will not spend time looking at in-depth. However, lets talk about the purpose of each of the other files. The files with the `log` extension are were your data payload will be stored. Other important information is the offset of the message as well as the `CreateTime` field. Why the need for any other files then? Since Kafka is built for speed, it uses the `index` file to store a mapping between the logical message offset and a physical position inside the `index` file. Having the index for Kafka to find a message quickly and then directly read from a position in the log file makes it easy to see how Kafka can still server consumers looking for specific offsets quickly. The `timeindex` file works similar however using a timestamp rather than offset value.

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

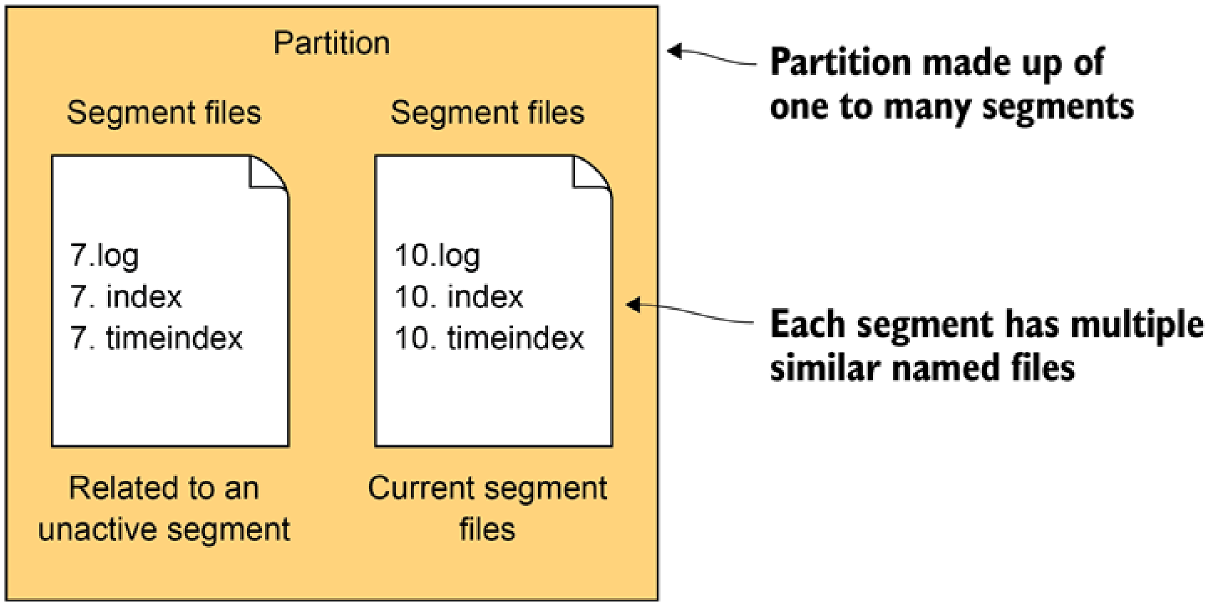

Partition
Segment files
Segment files
7.log
7. index
7. timeindex
10.log
10. index
10. timeindex
Related to an unactive segment
Current segment files
Partition made up of one to many segments
Each segment has multiple similar named files

**Figure 7.6 Segments Make up a Partition**

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

So using what we now know about segments, we know why you might have multiple files with a .index, .timeindex, and .log in your partition directory. If you have four segments for example, you should have a set of 4 files each with one of the previous three extensions for a total of twelve total. If you only see one of each file extension, you only have 1 segment.

## 7.2.2 Viewing Segments

Let's try to take a peek at a log file to see our messages we have produced to the topic so far. If you open it in your favorite text editor, you will not see those messages in an easy to read format. Due to that, the Kafka command bin includes a script `kafka-run-class.sh` in which we can leverage to look at those log segments. Listing 7.7 shows us reaching for the `DumpLogSegments` parameter to look at a segment log file for partition 0 of my topic named 'test'.

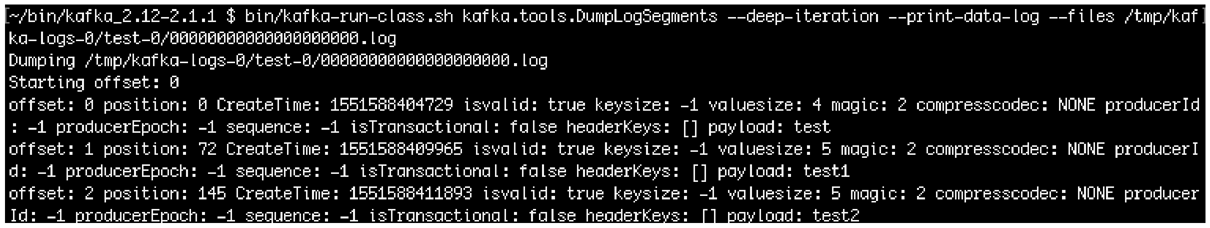
**Listing 7.7 Looking a dump of a segment**

```
kafka-run-class.sh kafka.tools.DumpLogSegments --print-data-log \   ❶
--files logs/test-0/00000000000000000000.log   ❷
```

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

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

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



```
~/bin/kafka_2.12-2.1.1 $ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --print-data-log --files /tmp/kaf
ka-logs-0/test-0/00000000000000000000.log
Dumping /tmp/kafka-logs-0/test-0/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 CreateTime: 1551588404729 isvalid: true keysize: -1 valuesize: 4 magic: 2 compresscodec: NONE producerId
: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: test
offset: 1 position: 72 CreateTime: 1551588409965 isvalid: true keysize: -1 valuesize: 5 magic: 2 compresscodec: NONE producerI
d: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: test1
offset: 2 position: 145 CreateTime: 1551588411893 isvalid: true keysize: -1 valuesize: 5 magic: 2 compresscodec: NONE producer
Id: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: test2
```

**Figure 7.7 Dumping Log Segment**

Looking at Figure 7.7, you can see the payloads in text that is a little clearer to read than from previously trying to `cat` the log file directly. For example, you can see messages on the topic 'test', 'test1', and 'test2'. Hopefully you will have more valuable data in your payloads!

**TIP** **DumpLogSegments usage**

DumpLogSegments is a useful tool but can appear to give you less or incorrect details than you had hoped if you run it against the active segment index files. It might be helpful to run the `DumpLogSegments` command both against your active segment and any others that have been rolled to see the difference in how the file content might be different. The real way to troubleshoot index files is to use pass the option `--index-sanity-check` and/or `--verify-index-only` against the `log` file without passing the `--print-data-log` option. Also, when using the `DumpLogSegments` tool above, adding the `--deep-iteration` parameter will show information including messages compressed inside wrapper messages.

As for the large number name of the log file, is not random. Each segment is created with the offset of its first message as the file name. In our examples since we are just starting out with new topics, you will more than likely see them segments as all zeros.

Let’s try to make the current segment files roll and create new logs. One thing we can do for a quick test is to set how often our log segments should be rolled in milliseconds. Listing 7.8 shows how to alter the config of the topic named 'test'.

**Listing 7.8 Looking a dump of a segment**

```
kafka-configs.sh --zookeeper localhost:2181 --alter \   ❶
--entity-type topics \   ❷
--entity-name test  \   ❸
--add-config segment.ms=60000   ❹
```

❶ We are using the alter command to edit configuration since we are changing current default values

❷ Our enity type is topics a specific to your config change

❸ The entity name in this case is our topic name

❹ We are setting our segment files to rotate every minute

The `segment.ms` configuration value is being altered to roll our segment files after one minute. Using a time based approach helps us test the topic without having to rotate based off of message sizes. If you keep watching the partition directory, you should see a new group of segment files appear soon as you continue to send messages to your topic.

To stop this frequent rotation, you can use the `delete-config` option to remove the configuration that we added in our previous command. Listing 7.9 shows how to remove the configuration for `segment.ms` that we had just added.

**Listing 7.9 Looking a dump of a segment**

```
kafka-configs.sh --zookeeper localhost:2181 --alter \
--entity-type topics --entity-name test \
--delete-config segment.ms   ❶
```

❶ We are deleting the segment rotation configuration we just created for our testing

One of the impacts of being able to see this data is one thing to remember when you think about your overall data security. Do you have ways to limit access to the message brokers to only certain developers so that not just anyone can read the log files directly, i.e with tools such as `cat`? Do you require that your data-at-rest be encrypted? Since data security and access controls are common concerns with most data that holds value, we will look at ways you can secure Kafka and topics in Chapter 10.

Of course, facts about the segment log and index files are internal Kafka broker details that you hopefully would not rely on in your own applications. However, knowing how to look at these logs might be helpful when reasoning about and looking at your filesystem for troubleshooting. If you know where you log directory exists and you do not see any segments log growing when you know you should have producer traffic, it will be good to know the parts of the system should be updating and working as you narrow down your troubleshooting path.

## 7.3 More Topic and Partition Maintenance

It really helps to imagine Kafka as a living and complex system (it's distributed afterall) that might need some care and feeding from time to time. In this section we are going to tackle what to do if you really want to reassign partitions, do replica assignment changes, and preferred replica elections.

### 7.3.1 Replica Assignment Changes

As your topic starts to take on data over time, you might start to have events that can cause your topic's health to change over time. One reason to prompt for a change might be that a new broker is added to the cluster. Since that broker did not exist when your topic was created, Kafka did not even have the option to spread the load of the partitions to that server in your cluster. A removed broker (maybe due to hardware failure) can also cause an issue. If your topic had a replica on a broker that is now gone, it might be in a state of being under replicated. One method to fix these scenarios includes using the `kafka-reassign-partitions.sh` command.

This command requires a bit of work and a couple of steps to complete our intended data movement. Our first command helps us form a snapshot of our current assignments and also a plan for the future commands.

First create the topics.json file with a list of the topics you are interested in moving. I am using my topic 'test' for this example. Listing 7.10 shows an example topic file in json format for this single topic.

**Listing 7.10 Topics File for Reassigment Content**

```
{"topics":
  [{"topic": "test"}],    ❶
  "version":1
}
```

❶ Listing the topic name we care about only

Notice when running the reassign command that I have listed the broker ids for my setup (which is three brokers total). This parameter as well as your topics.json file content will likely be different. Listing 7.11 shows an example of using the json file we created listing our topic as input into `generate` option command.

**Listing 7.11 Generate a plan**

```
kafka-reassign-partitions.sh --zookeeper localhost:2181 --generate \   ❶
--topics-to-move-json-file topics.json \   ❷
--broker-list 0,1,2   ❸
```

❶ The generate command is like a dry-run option that does not alter our cluster

❷ topics.json is a local file that listed the config on the topic names in json we are interested in

❸ Broker id list of those in your specific cluster

This command generates output in two sections to standard output. The top section with the description: 'Current partition replica assignment' is data that you might want to hang onto in case our following changes need to be reverted. The second block of output is the proposed reassignment.

Now that we have a plan for the change, it is time to actually execute it. Up to this point, our commands did not change anything, we were just gathering and planning our next steps. For my command, I copy and paste the generated plan from the previous step into it's own file title plan.json. Using the `--execute` option is how this plan will actually start the reassignments. Listing 7.12 shows using our plan that was created in our previous step as input into the actual command that will change our cluster.

## Listing 7.12 Executing our plan

```
kafka-reassign-partitions.sh --zookeeper localhost:2181 --execute \   ❶
--reassignment-json-file plan.json   ❷
```

❶ Execute is what will actually alter our cluster

❷ plan.json holds the execution plan we generated in the previous plan

Keeping the specific reassignment-json-file that you used (in our case plan.json) for the 'execute' stage is handy to make sure that our execution is done or at least in process. We can verify our assigments by using the `--verify` command. Listing 7.13 lets us figure out the current status of our previous command.

## Listing 7.13 Verify our execution

```
kafka-reassign-partitions.sh --zookeeper localhost:2181 --verify \   ❶
--reassignment-json-file plan.json
```

❶ Verify helps us see the progress or status after the fact of how our assignment completed

## 7.3.2 Altering the Number of Replicas

One of the processes we talked about earlier in the chapter was how to add a partition to an existing topic using an alter command. However, we skipped over how to change the replication factor. Now that we have been introduced to the `kafka-reassign-partitions.sh` command, it will make more sense of how to change this number since we leverage the same command as we just did for reassignments as a whole. Listing 7.14 shows an example json file that can be used to start our plan.

**Listing 7.14 Create a Plan Adding a Replica**

```
{"partitions":
  [{"topic": "test", "partition": 0,    ❶
    "replicas": [    ❷
      0,
      1
     ]
   }
  ],
  "version":1
}
```

❶ We need to list the topic name and partition we are looking at a plan for

❷ 0 and 1 are an array of broker ids where replicas should exist in the future

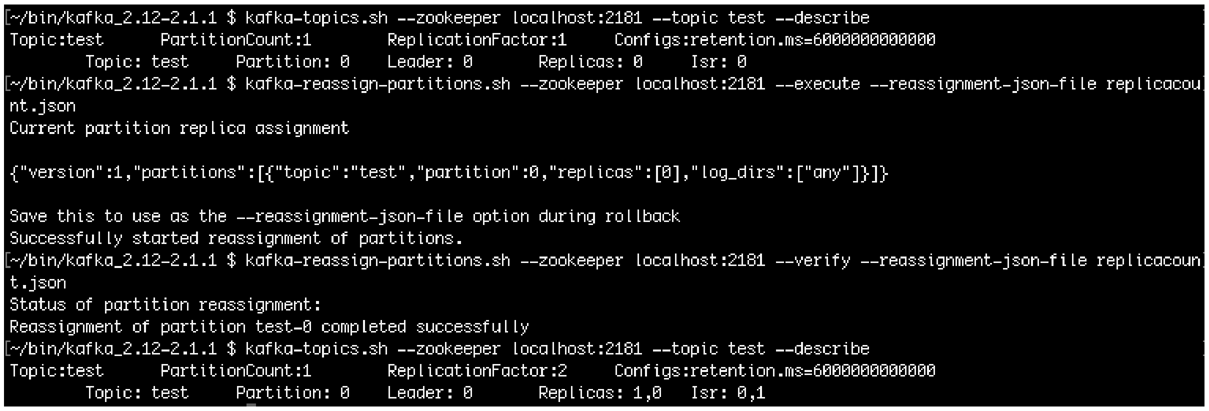
By adding the above to my local file replicacount.json, I should be able to have 2 replicas existing on broker id 0 and id 1 after this change. Listing 7.15 shows using the plan we created in order to apply the change to our cluster.

**Listing 7.15 Executing our plan to increase replica count**

```
kafka-reassign-partitions.sh --zookeeper localhost:2181 --execute \   ❶
--reassignment-json-file replicacount.json
```

❶ We run the execute to run the plan created in our previous command

Describing the topic before and after the plan was executed to show the change in replicas is a good idea to order to make sure that your change impacted your topic as you expected. After execution of the `kafka-reassign-partitions` command, you should be able to see a new replica present!



```
[~/bin/kafka_2.12-2.1.1 $ kafka-topics.sh --zookeeper localhost:2181 --topic test --describe
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:retention.ms=600000000000
        Topic: test     Partition: 0    Leader: 0       Replicas: 0     Isr: 0
[~/bin/kafka_2.12-2.1.1 $ kafka-reassign-partitions.sh --zookeeper localhost:2181 --execute --reassignment-json-file replicacou
nt.json
Current partition replica assignment

{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[0],"log_dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.
[~/bin/kafka_2.12-2.1.1 $ kafka-reassign-partitions.sh --zookeeper localhost:2181 --verify --reassignment-json-file replicacoun
t.json
Status of partition reassignment:
Reassignment of partition test-0 completed successfully
[~/bin/kafka_2.12-2.1.1 $ kafka-topics.sh --zookeeper localhost:2181 --topic test --describe
Topic:test      PartitionCount:1        ReplicationFactor:2     Configs:retention.ms=600000000000
        Topic: test     Partition: 0    Leader: 0       Replicas: 1,0   Isr: 0,1
```

**Figure 7.8 New Replica Process**

Notice in Figure 7.8 how when you describe the topic after adding the replica that the field ReplicationFactor is now 2, as well as two entries in the Replicas and Isr list.

## 7.3.3 Preferred Replica Elections

As a reminder from our partition discussion earlier, each partition has a leader that handles all client-related data needs. The other replicas are only followers that do not talk to clients. When topics are first created, the cluster tries to spread these leaders evenly among the brokers in order to split some of the leader workload evenly. The 'preferred' leader will not change for the topic as it will always be considered the replica that is first in the ISR list. However, after broker stops and starts for maintenance or updates, the preferred replica might not be the leader. The script `kafka-preferred-replica-election.sh` can be used in this scenario to help bring leadership back to replicas as they existed on creation of a topic. Listing 7.16 shows the simplest command syntax that will run against the entire cluster.

**Listing 7.16 Electing Preferred Leader**

```
kafka-preferred-replica-election.sh --zookeeper localhost:2181   ❶
```

❶ All of the topic partitions for our cluster will be impacted

If you cluster is large and you have many partitions to re-elect, you might see an error and will have to provide a json file to pass the `kafka-preferred-replica-election` called `path-to-json-file`. The format of that file will consist of a map of each topic to the partition that should be elected as the leader. Listing 7.17 shows an example of a json file that will help you limit your scope of changes you wish to make.

**Listing 7.17 Example Preferred Leader Json**

```
 {"partitions":
       [{"topic": "test", "partition": 1},     ❶
        {"topic": "my_test", "partition": 2}
       ]
}
```

1. Using this file with the path-to-json-file parameter can help you limit which topics and partitions you want to impact

If you are interested in looking at the success of this command and want to confirm the topic status, when describing a topic, you should see the leader of the partition broker id matching the first broker id in the list labeled Replicas.

### 7.3.4 Editing ZooKeeper Directly

While Kafka can be a robust distributed service, there are a couple of ways to get ourselves into trouble. One of the biggest things to think about is trying to modify ZooKeeper directly. Kafka uses ZooKeeper and without knowing all of the ways Kafka itself relies on and uses those pieces, it can be hard to know the impacts of you removing a znode. Being practical however, there may be a time when nothing else is working that you are willing to try to do what you need to get results. When we were looking at the data in ZooKeeper that showed the cluster broker that was the controller as we did in Chapter 6 Section 6.5, we found out it was located at the path /controller. If your existing cluster controller is just filling its logs with exceptions and errors or the JVM process is hung, deleting that ZooKeeper node in a manual fashion might be a valid way to get your cluster back in quicker fashion.

## 7.4 Topic Compaction

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

To make it clear, topic compaction is different from the retention of existing segment log files when compared to a regular topic. With compaction, the goal is not to expire messages but rather to make sure that the latest value of a key exists and is not maintaining any previous state. As just referenced, compaction depends on a key being part of your messages and not only being null. Otherwise, the whole concept fails if there is not 'key' to update a value for in practice.

`cleanup.policy=compact` is the topic level configuration option we use to create a topic as compacted. This is different from the default configuration value that was set to 'delete' before our override. Listing 7.18 shows adding the config option needed for this new compacted topic.

**Listing 7.18 Creating a compacted topic**

```
kafka-topics.sh --zookeeper localhost:2181 --create \   1
--topic compact_test_topic --replication-factor 2 --partitions 2 \
--config cleanup.policy=compact   2
```

1. We create a topic like any other topic
2. Setting the cleanup.policy config parameter makes this topic compacted

One of the easiest comparisons for how a compacted topic presents data can be seen in how a database table updates an existing field rather than appending more data. Let’s say that you want to keep a current member status for an online membership. A user can only be in one state at a time, either a 'Basic' or 'Gold' membership. At first, the user enrolled for the 'Basic' plan but over time ungraded to the 'Gold' plan for more features. While this is instead still an event that Kafka stores, in our case we only want the most recent membership level for a specific customer (our key). Figure 7.9 shows an example using 3 customers.

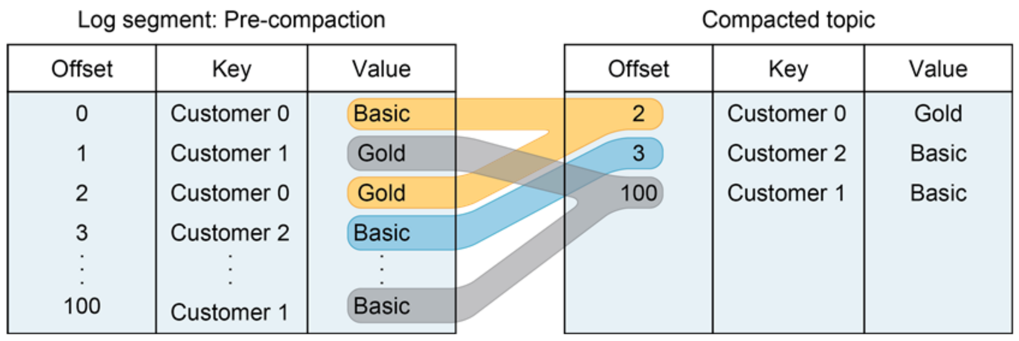


**Figure 7.9 Compaction in General**

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

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

## 7.4.1 Compaction Cleaning

When a topic is marked for compaction, a single log can be seen in a couple of different states: clean or dirty. Clean is the term for the messages that have been through compaction before. Duplicate values for each key should have been reduced to just one value. The dirty messages are those that have not yet been through compaction. Multiple values might still exist for this message for a specific key until all messages are cleaned. Figure 7.10 shows how a pointer is used to know which messages have been processed with compaction and which messages have yet to be visited.

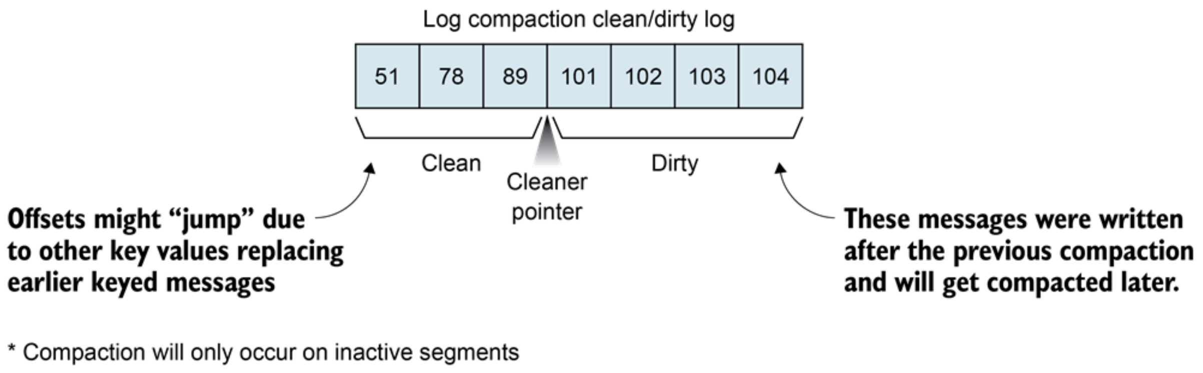

Log compaction clean/dirty log
51
78
89
101
102
103
104
Clean
Cleaner pointer
Dirty
Offsets might "jump" due to other key values replacing earlier keyed messages
These messages were written after the previous compaction and will get compacted later.
* Compaction will only occur on inactive segments

**Figure 7.10 Compaction Cleaning**

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

### *7.4.2 Can Compaction Cause 'Deletes'*

Even though we have the notion of an immutable stream of events, there is a time when updating a value for a customer might include a removal of that value. Following our subscriber level scenario above, what if the customer decides to remove their account? By sending an event with the customer key with a null message value, Kafka will keep this last message for a certain time period. The message is considered a 'tombstone'. If you have used other systems like HBase the notion is very similar. Besides being a part of an immutable log, why would Kafka still serve this message to consumer clients? One reason is that your consumers might want to know about this final state. In other scenario, the billing subscription team might want to know that a customer will not be billed further due to their current state (of removing their subcription). Figure 7.11 shows that the null value does not remove a message immediately but is served as any other message.

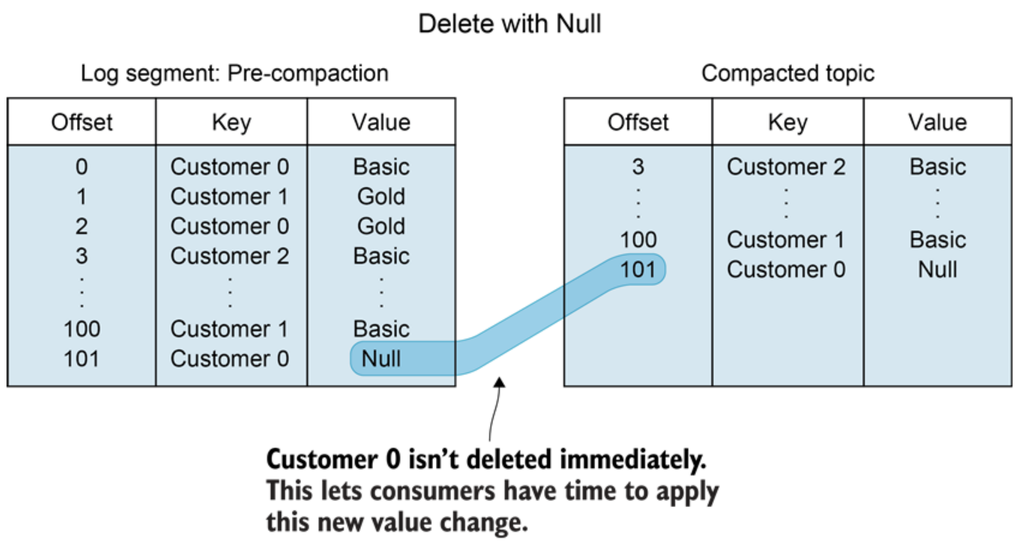


**Figure 7.11 Compaction For a Deleted Value**

The General Data Protection Regulation (GDPR) rules are one valid reason that you might want to make sure that some of your topics are setup for compaction depending on your customer's location. While these are external rules that an application may or may not have to deal with: Kafka can help you fulfill those data requirements by its core feature set. Our previous discussion of segments can help us further understand when the data is really removed from disk. Compaction only runs on inactive segments, not at with active segments. In other words, those segments being currently written to are not subject to the update fashion of compactions. This also means that client consumers can also see more than one message from a specific key! One side-effect then of how often data is enabled for specific processing is when that log segment is rolled. If your topic does not get a lot of data, you might want to force the rotation based on a period of time. `log.roll.hours=24` seems to be a popular configuration option in order to have segments rotated about once a day.

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

## 7.5 Summary

In this chapter you learned:

- Various topic creation parameters such as Replication Factor and Partitions
- How log file segments are written in partition directories
- A strategy to look at the log files on disk
- Techniques to reassign partitions and re-elect partition leaders
- What topic compaction is and why we would want to use it

# 8 Kafka storage

***This chapters covers:***

- Moving data and retention in Kafka
- Tools to help with data movement
- Data architectures Kafka enables

So far we have treated our data as moving into and out of Kafka for brief periods of time. Another decision to really analyze is where your data can live for longer terms. When we think of databases like MySQL or MongoDB, we do not always think if or how that data is expiring-we know it is for majority of your application's entire lifetime. In comparison, Kafka storage can logically sit in between the practical storage solutions of a database and a message broker in regards to data storage; especially if we think of message brokers holding onto messages until they are consumed by a client as it is often in other message brokers.

As in all of engineering, tradeoffs are involved in each decision, including your business needs and requirements. Let's look at a couple of options for storing and moving data in your environment.

## 8.1 How Long to Store Data

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

How do we configure brokers retention? The main focus is on the size of your logs and or the length of time your data exists. Table 8.1 shows some of the broker configuration options that are helpful for retention:

**Table 8.1 Broker Retention Configuration**

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

How do we disable log retention limites and allow them to stay 'forever'? By setting `log.rentention.bytes` and `log.rentention.ms` to -1 these values will have the effect of not using time or size of the log in order to delete log data.

Another option to think about is how you **may** get a similar rentention for latest values by using keyed events with a compacted topic. While data could still be removed during compaction cleaning, the most recent keyed messages will still be in your log. This may in fact be how you wish to retain data in use cases where you do not need every event (or a history) of how a key changed state to the current value.

What if you want your data to stick around for a while, but simply do not have the disk space to hold your data on brokers? Another option for long-term storage is to move the data outside of Kafka and not retain it internally to the Kafka brokers themselves. Before data would be removed by retention from Kafka, we could store the data into a database, HDFS, or upload our event messages into something like an S3 bucket in Amazon Web Services (AWS). All of these paths are valid options and can help us hold onto our data for longer periods than we might want to spend on disk space if our consumers are traditionally done in a specific time range.

## 8.2 Data Pipelines

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

### 8.2.1 Keeping the original event

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

### 8.2.2 Moving away from a batch mindset

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

## 8.3 Tools

Data movement is a key to many systems, Kafka included. While you can stay inside the open-source Kafka and Confluent offerings like Connect, which was discussed in Chapter 3, there are other tools that might fit your infrastructure or are already available in your tool suite. Depending on your specific data source or sinks, these options below might help you achieve your goals.

### 8.3.1 Apache Flume

If you were first introduced to Kafka through work in the Big Data space, it is a strong possibility that you might have used Flume in relation to your cluster. If you have ever heard the term "Flafka", you have definitely used this Kafka and Flume integration. Flume can provide an easier path for getting data into a cluster and would rather interact with configuration instead of custom code. If you wanted to ingest data into your Hadoop cluster and already have support from a vendor like Cloudera on these various pieces, Flume is a solid option to get data into your Kafka cluster.

Figure 8.1 shows an example of how a Flume agent is running on a node as its own process. It watches the files local to that server and then uses the configuration for the agent that you provided to send data to a sink destination.

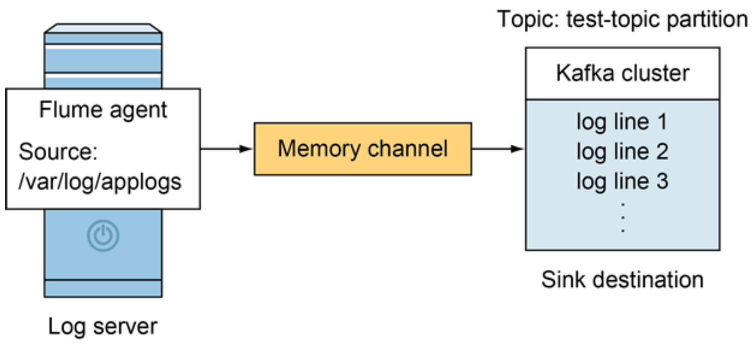


**Figure 8.1 Flume Agent**

Let's take a look again at integrating log files (our source of data) using a Flume agent into a Kafka topic (our data sink). Listing 8.1 shows a sample configuration file that would be used to set up a local Flume agent to watch a directory for changes. The changes would be placed in a Kafka topic titled `test-topic`. To imagine this example with a comparison, it is like using a `cat` command on a file in a directory to read a file and sending the result to a specific Kafka topic.

**Listing 8.1 Flume configuration for watching a directory**

```
ag.sources = logdir     ❶
ag.sinks = kafkasink
ag.channels = c1

#Configure the source directory to watch
ag.sources.logdir.type = spooldir   ❷
ag.sources.logdir.spoolDir = /var/log/applogs
ag.sources.logdir.fileHeader = true

# Describe the Kafka sink
ag.sinks.kafkasink.channel = c1   ❸
ag.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
ag.sinks.kafkasink.kafka.topic = test-topic
ag.sinks.kafkasink.kafka.bootstrap.servers = kafka-1:9092, kafka-2:9092
ag.sinks.kafkasink.kafka.flumeBatchSize = 10
ag.sinks.kafkasink.kafka.producer.acks = 1
ag.sinks.kafkasink.kafka.producer.linger.ms = 5
ag.sinks.kafkasink.kafka.producer.compression.type = snappy

# Memory channel configuration
ag.channels.c1.type = memory    ❹
ag.channels.c1.capacity = 1000
ag.channels.c1.transactionCapacity = 100

# Bind both the sink and source to the same channel
ag.sources.logdir.channels = c1    ❺
ag.sinks.kafkasink.channel = c1
```

❶ You can define your own names for the source, sink, and channel

❷ We are using the spooldir specific source type configuration that is needed to get Flume know which directory to watch for log entries

❸ This section helps define our topic and Kafka cluster information where we want our data to end up

❹ A channel is used as a transport method to get data from the source to the sink

❺ Like a pipe in Unix, the source is attached to the sink by the channel we defined in our configuration

Listing 8.1 shows how you could configure a Flume agent running on a server. At this point in your Kafka journey, you should notice the sink configuration looking a lot like the properties you have used before in your own Java client producer code. If you do not see your favorite property-it is important to note those in the example above with the prefix `kafka.producer`. The Flume agent will use those prefixed values with creating its internal Kafka producer client so you can use those other options if needed. On the flip side, if you are using Kafka as your source, you can use the prefix `kafka.consumer` to use any valid consumer property as well.

It is also interesting to note that Flume can use Kafka as not only a source or as a sink, but also as a channel. Since Kafka is seen as a more reliable channel for events, Flume can leverage Kafka to deliver messages between various sources and sinks. In our example for Listing 8.1, we are using a memory channel that could loss messages if the agent failed and only contained that information in RAM. Using a Kafka channel might be a better idea for higher availability. If you are reviewing Flume configuration and see Kafka mentioned, be sure to notice where and how it is actually used.

**Listing 8.2 Flume Kafka Channel configuration**

```
ag.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel    ❶
ag.channels.channel1.kafka.bootstrap.servers = kafka1:9092,kafka2:9092,kafka3:9092    ❷
ag.channels.channel1.kafka.topic = channel1-topic   ❸
ag.channels.channel1.kafka.consumer.group.id = flume-cgroup   ❹
```

❶ The Kafka channel type used in Flume is the builtin class KafkaChannel

❷ As any other Kafka cluster configuration, we provide our servers to connect to

❸ The topic that will hold the data between source and sink

❹ You can provide a consumer group to avoid collisions with your other consumers

Listing 8.2 shows the Flume agent configuration you could use to provide a reliable channel between various sources and sinks that Flume supports.

### 8.3.2 Debezium

Debezium describes itself as a distributed platform that helps turn databases into event streams. In other words, updates to your database can be treated as events! If you have a database background (or not!) you may have heard of the term change data capture (CDC). As the name implies, the changes to data can be tracked and used to react to that change. At the time of writing this chapter, MySQL, MongoDB, and PostgresSQL servers were supported with more systems slated to be added. Debezium uses connectors and Kafka Connect in order to record the events and your application would consume from Kafka as a normal client. Figure 8.2 shows an example of Debezium when it is registered as a connector in regards to Kafka Connect.

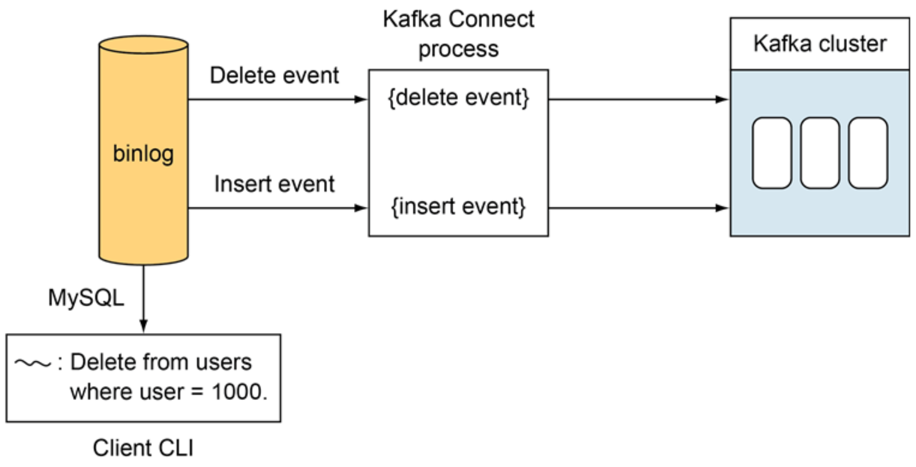


**Figure 8.2 Kafka Connect and Debezium used with a MySQL Database**

In our scenario, a developer was using a command line interface (CLI) and deleted a user against the MySQL database instance that was being monitored for changes. Debezium would capture that event being written to the databases internal log and that event would go through the connector service and feed into Kafka! If a second event such as a new user being inserted into the database, a new event would be captured.

An additional note, while not Kafka-specific, there are other examples of using techniques like CDC in order to provide timely events or changes to your data that might help you draw a parallel to what Debzium is aiming for overall. If you have ever used a platform like Meteor, you can see a couple of ways data changes can be captured and acted on in your application. MongoDB has the concept of an operations log (oplog). This specific log holds a continuous record of actions that have modified the data stored in your database instances. By watching this log for changes, you application or platform can propagate these events to other parts of your logic. This allows web UI pages to be updated in near real-time and seem as if they were updating without the need for a reload of the entire web page. In comparison, what if we did not have this log available for a specific database or for our instance? There is always the option of polling against a database. While not always the most efficient, it is a valid path to identify changes and differences between polls.

### 8.3.3 Secor

Secor is an interesting project that has been around since 2014 from Pinterest that aims to help persist Kafka log data to a variety of storage options including S3, Google Cloud Storage, Microsoft Azure Blob Storage, and OpenStack Swift. The options for output is also various including sequence files, ORC, Parquet files as well as other formats. As always, one major benefit of these projects having source code in a public repository is that fact that you can see how other teams have implemented requirements that might be similar to yours. Figure 8.3 shows how Secor would act as a consumer of your Kafka cluster, very much like any other application. Having a consumer being added to your cluster for data backup is not a big deal, it leverages the way Kafka has always handled multiple readers of the events!

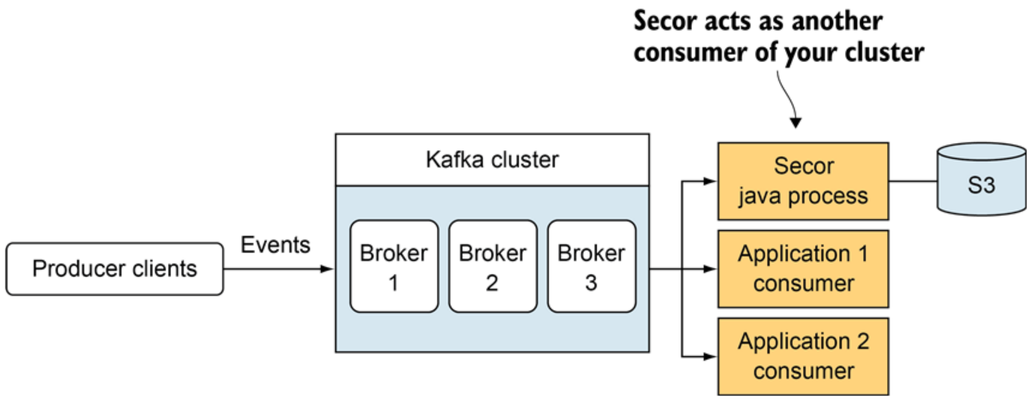


**Figure 8.3 Secor acting as a consumer and placing data into S3**

Secor runs as a Java process and can be feed your specific configuration. In effect, it acts as another consumer of your existing topic(s) to gather data to end up in your specific destination like an S3 bucket. Secor does not get in the way of your other consumers and allows you to have a copy of your events so that they are not lost once Kafka retention removes data from its own logs. Listing 8.3 shows an example of how you can start the Secor java application with parameters for your own usage.

**Listing 8.3 Running Secor Java application**

```
java -ea -Dlog4j.configuration=log4j.prod.properties \
  -Dconfig=secor.prod.backup.properties \   ❶
  -cp secor-0.1-SNAPSHOT.jar:lib/* \   ❷
  com.pinterest.secor.main.ConsumerMain   ❸
```

❶ You pass your configuration for the backup on the command line specific to your destination and Kafka cluster. The config file format can be seen by looking at the in src/main/config source code location

❷ The jar that contains the secor code

❸ ConsumerMain is the java file used to launch Secor

Invoking Secor should be familiar to those who are used to working with jars in a Java environment. Arguments can be passed to the Secor application with standard '-D' parameters. The most important file to update is the config option which is listed in our example as: `ecor.prod.backup.properties`. This file is where you can fill in the details about your specific S3 bucket, for example.

Let’s think about an example of how moving data out of Kafka could be used at a later time. First to clarify, we will break down our usage of the same data between two different areas. One area is working with the data as it comes into Kafka, in an operational manner. Operational data is your events that are produced by your day to day operations. I always think of the event to order an item from a website as a specific example. A 'purchase' event triggers your application into motion and does so in low latency way. The value of this data to your real-time applications might warrant keeping the data for a couple of days until the order is complete and mailed. After this timeframe, the value of your event may change to be more important for your analytical systems. Analytical data, while based on that same operational data, is usually used more to make business decisions. In traditional systems, this is where systems like a data warehouse, an Online Analytical Processing system (OLAP), and Hadoop shine. Those events can be data-mined using different combinations of fields in your events in different scenarios to find insights into sales data. If you notice that sales of cleaning supplies away spike before holiday, you might be able to use that data to generate better sale options for your business in the future.

## 8.4 Bringing data back into Kafka

One of the most important things to note is that just because your data has left Kafka once, it does not mean that you can't put it back in again! Figure 8.4 shows an example of data that lived out its normal lifespan in Kafka and was archived in S3. When a new application logic change required that older data be reprocessed, the application did not have to create a client to read from both S3 and Kafka. Rather, using a tool like Kafka Connect, we were able to load that data from S3 back into Kafka. The interface stayed the same from the point of view of your applications! While it might not seem obvious at first glance or why you would even want to do such a thing, let's think of a situation in which we might find value in moving our data back into Kafka after we processed it once already and the retention period of our normal workloads has passed.

Imagine a team working on trying to find patterns in data that they have collected throughout years of handling events. In our example, there are terabytes of data. To serve operational real-time data collection, this data has been moved from Kafka into HDFS after real-time consumers have dealt with the messages. So does your application logic now have to pull from HDFS directly? Why not just pull it back into Kafka and your application can process the data as it had before? Loading data into Kafka again is a valid way of reprocessing data that may have aged out of your system. Figure 8.4 shows another example of how we can move data back into Kafka.

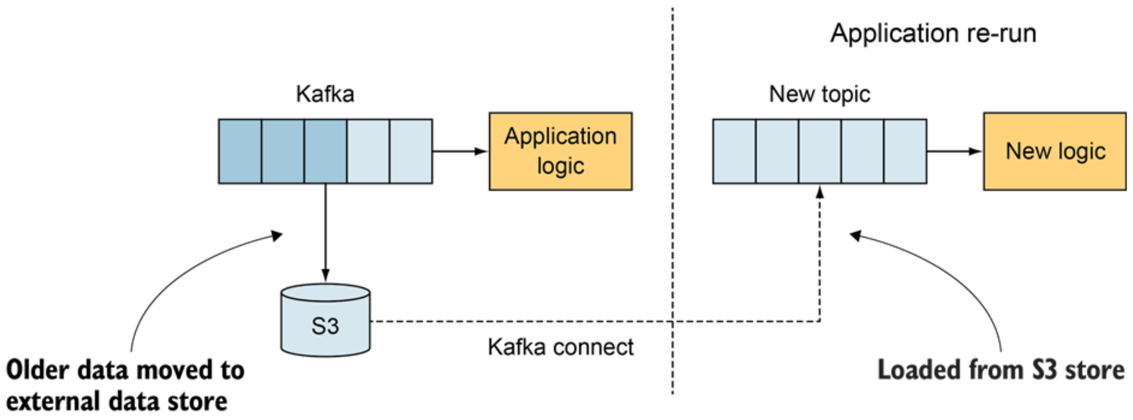


**Figure 8.4 Data Back into Kafka**

Figure 8.4 shows an application that pulls its data from Kafka. After some time, events are not available to the applications due to data retention within Kafka itself. However, we have a copy of all previous events in an S3 bucket. Let's say that we have a new version of our previous application and would prefer to go through all of the previous data events as our previous application. However, since they are not in Kafka, do we pull them from S3 now? Do you want your application logic to pull from various sources or just to have one interface (that being

Kafka)? For us, we can create a new topic in our existing Kafka cluster and load the data from S3 with Kafka Connect placing the data into a new Kafka topic. Our application can then run against Kafka, processing events without having to change its read/pulling processing logic.

The thought process is really to keep Kafka as the interface of your application and not have to create multiple ways to pull data into processing. Why create and maintain custom code to pull from HDFS, S3, etc when you can use one of the existing tools like Connect or Secor to move the data into and out of Kafka. Once you have your data in that one interface, you can process it the same. Keep in mind this technique only applies to data that has been removed from Kafka. If you still have the total timeline of data that you need in Kafka already, you can always seek to the earlier offsets!

## 8.5 Architectures with Kafka

While there are various architectural patterns to build your our products such as Model-View-Controller (MVC), Peer-to-Peer (P2P), or Service-oriented architecture (SOA) to name a few, Kafka and viewing your data as events can change the way you think about your entire architectural design. Let’s take a peek at a couple of architectures that could be powered by Kafka (and to be fair, other streaming platforms) and help us get a different perspective on what our systems can look like for our customers. It is interesting to note that the term 'Big Data' is used in reference to some of these discussions. I think it is important to note that the amount of data and the speed concerns of wishing to process all of that data were drivers that helped create some of these ideas around the system designs. However, these architectures are not limited to 'Fast Data' or 'Big Data' applications only. By hitting the limits of specific traditional database technologies, new views on data evolved.

### 8.5.1 Lambda Architecture

If you have ever researched or worked with data applications that has included needs for both batch processing and operational workloads, you might have seen references to another architecture called the Lambda Architecture. The implementation of this process can start with Kafka as well, but is a little more complex. The real-time view of the data is combined with a historical view to serve to end-users. The complexity of merging these two views of the data should not be ignored. From experience, it was a challenge to rebuild the serving table. Also, you are likely going to have different interfaces for your data to maintain as you the results from both systems. The book "Big Data" written by Nathan Marz with James Warren discuss the Lamba Architecture in detail and go in details about the Batch, Serving, and Speed layers. Figure 8.5 shows an example of how the batch and speed layers use different frameworks in order to expose their respective views to end-users.

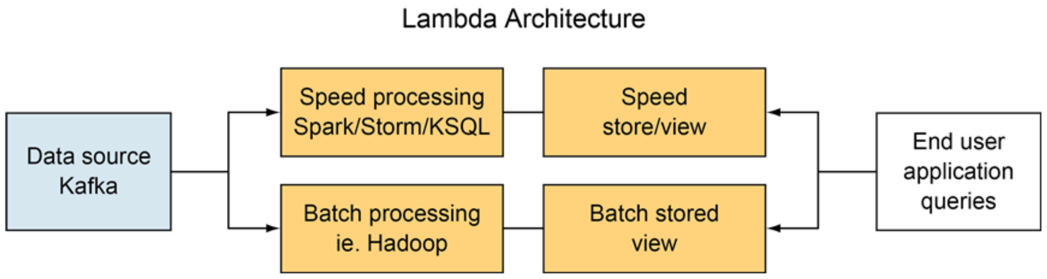

Lambda Architecture
Data source Kafka
Speed processing Spark/Storm/KSQL
Batch processing ie. Hadoop
Speed store/view
Batch stored view
End user application queries

**Figure 8.5 Lambda Architecture**

To get a feel for this architecture, let’s look at each layer at a high level.

- Batch: This layer can be thought of in a context of how batch processing with Map-Reduce occurs in a system like Hadoop. As new data is added to your data stores, the batch layer can continue to pre-compute your view of the data that has already arrived in your system.
- Speed: This layer is similar in concept to the batch layer accept it produces views from recent data.
- Serving: This layer updates the views it sends to consumers after each update to the batch views.

For the end-user, the Lambda architecture combines the results from the serving layer and the speed layer in order to answer requests with a complete view of all recent and past data. This real-time streaming layer is where Kafka fits in the most obvious way but can also feed the Batch layer as well.

### *8.5.2 Kappa Architecture*

Another architectural pattern that could also leverage the power of Kafka is the Kappa Architecture. This architecture was proposed by the co-creator of Kafka, Jay Kreps. Think about wanting to maintain a system that impacts your users without disruption. One way to do such a pattern is to switch out your updated views like in Lambda. Another way to do this is by running the current system in parallel to the new one and cutting over once the new version is ready to serve traffic. Part of this cut over is of course making sure that the data that is being served by the older version will be reflected correctly in the newer version.

You only regenerate the user-facing data when you need to-there is no need to merge old and new data which is an on-going process for some Lambda implementations. It does not have to be a continuous job, but rather invoked when you need an application logic change. Also, no need to change your interface to your data, Kafka can be leveraged by both your new and old application code at the same time. Figure 8.6 shows how a separate streaming process is started against the same source of data, in this case Kafka.

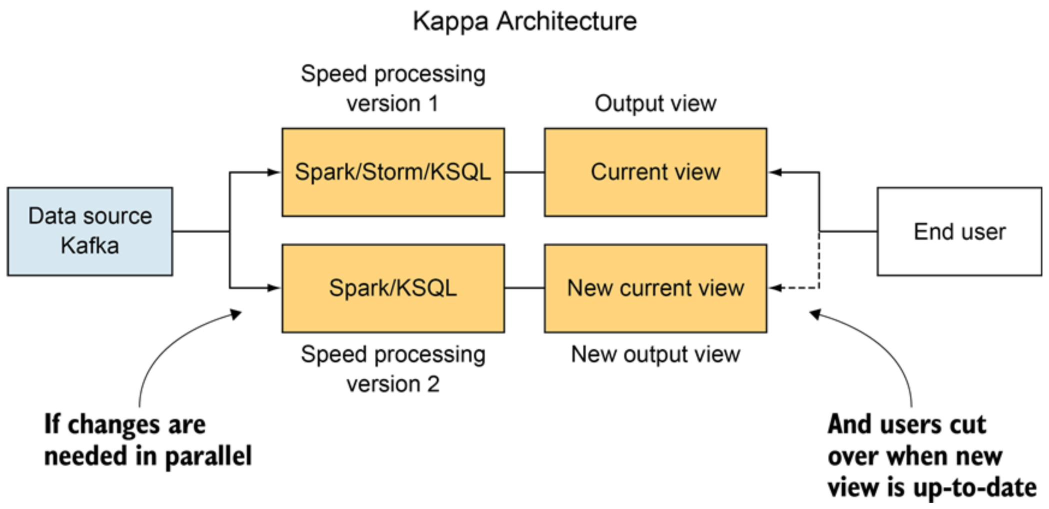

Kappa Architecture
Speed processing version 1
Output view
Data source Kafka
Spark/Storm/KSQL
Current view
End user
Spark/KSQL
New current view
Speed processing version 2
New output view
If changes are needed in parallel
And users cut over when new view is up-to-date

**Figure 8.6 Kappa Architecture**

The upper portion of Figure 8.6 shows the old application at version 1. It reads from Kafka and creates a view for the end-users. However, once a change is needed and a version 2 of the application is created and the lower portion of our flow kicks off in parallel to the existing running version 1. Version 2 reads from the same source of events as version 1, processes data and builds its view. Once version 2 has all of its events processed (in effect caught up with the events processed by version 1), the end-users can then be cut-over to read from the new view. Once version 1 is not needed for a backup or recovery, you should be able to delete the old view and version 1 code. There is no need to have a batch layer as there is only one set of logic used for making your end-user views!

## 8.6 Multicluster setups

Most of our topics and discussions so far have been from the viewpoint of our data in the context of one cluster. Kafka scales well and it is not unheard of to reach hundreds of brokers for a single cluster. However a one size cluster will not fit all infrastructures. One of the concerns we run into when talking about cluster storage is where you serve your data in relation to your end-user clients. In this section, we are going to talk about scaling by adding clusters rather than just by brokers alone.

### 8.6.1 Scaling by adding Clusters

Usually, the first thought to scale would be the resources inside your existing cluster. The number of brokers is usually the first option and makes a straight-forward path to growth. However, Netflix's MultiCluster strategy is a captivating take on how to scale Kafka clusters. Instead of using the broker number as the way to scale the cluster, they found they could scale by adding clusters themselves! They found that it might make sense for event producers to talk to one Kafka cluster and have those events moved to other Kafka clusters for consumption. In this setup, consumers would be reading messages from a different Kafka cluster that is entirely different from the ingestion cluster.

As per their scenario, the amount of data into the cluster was not a problem. However, once they had multiple consumers pulling that data out at once you can see a specific bottleneck for their network bandwidth. Even with 1 or 10 Gigabit Ethernet (GbE) data center networks, the rate of your data and the number of bytes out can add up quickly. Figure 8.7 shows an example of how having many consumers divided across clusters might help reduce the network impact of only one cluster being used. Note that this means each cluster would be on its own distinct network and not all in the same network or the problem would be likely worse.

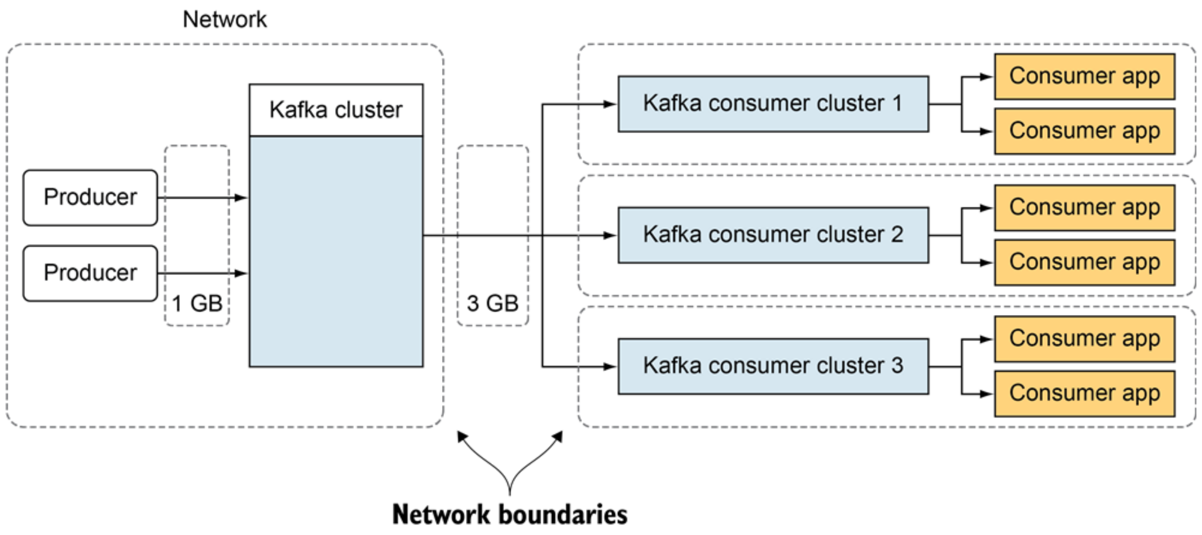


**Figure 8.7 Multiple Cluster Architecture**

Imagine the scenario, and with generous estimates on the network data amounts for ease of understanding, where we have producers that are sending about a gigabyte (GB) of data per second to one cluster. If you has six consumers, you would be sending 6 copies of that 1 GB of data for around 7 GB per second (againg let's not worry about exact numbers). You do not have much more room on a 10 GbE network. Figure 8.7 shows that if you have one cluster ingesting all of your events, you can narrow the amount of copies you send out in this specific network. The other clusters will help divide your total load.

This design brings to mind the idea of Command Query Responsibility Segregation (CQRS). Specifically, the idea of separating the load of reading data from that of writing data. Each action can scale in an independent manner without limiting each other. While CQRS is a pattern that can add complexity to your systems, it is interesting to note how this specific example helps handle the performance of a large cluster by moving the load of producers sending into Kafka from the much larger load of consumers reading the data.

Another benefit for this setup is the ability to do upgrades or fail-over in a non-rolling update fashion. Instead of rolling broker updates, the new cluster is bought up, sent traffic from producers and/or the old cluster, and then the older cluster is removed after the new one is ready to go.

One of the other distinct items to think about is the distance you have between your applications consumers and the data in your cluster. Data being local to your applications should help with the amount of time it would take to pull data from the other side of the country if you only had a single cluster as your source.

### 8.6.2 Hub and Spoke Integration

One of most common setups involved the hub and spoke model. As an aside, if you sort of squint at the Apache Kafka logo, you wouldn't be the first to see a hub and spoke diagram as your first reaction! Drawing a parallel with delivery logistics, a centralized hub (cluster in our case) receives messages from many different origin clusters with the spokes being different clusters that interact with local applications only.

### 8.6.3 Active-Active

Another common cluster setup example is the ability to have clusters in separate data centers in an Active-Active model. In these cases, each data-center has its own cluster. End-users of the clusters usually connect to the data center cluster that is closest to them unless there is an outage or failure scenario. The clusters have data replicated between them for both clusters to use no matter where the data originated from. Figure 8.8 shows how application in each data center would use the Kafka cluster that was closest to them.

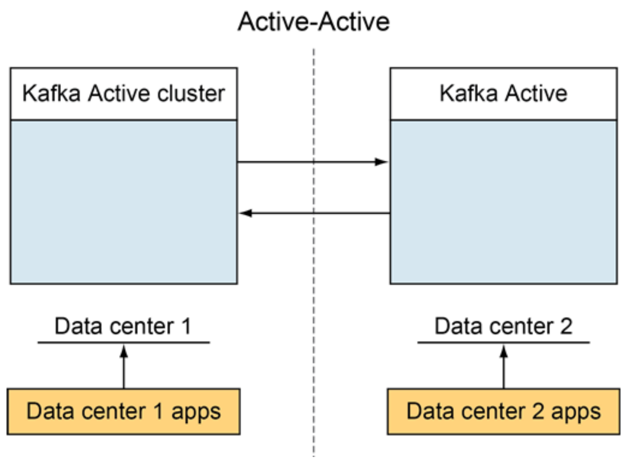


**Figure 8.8 Active-Active Clusters in Different Data Center Locations**

Figure 8.8 shows that in an ideal setup, you would have applications located in a data center use the cluster that was also present in that same data center. However, you would still allow consumers to see data from the other clusters once data was synched across the two data centers. One major concern for this setup is that data would have to be replicated across the clusters using MirrorMaker or something similar. Both clusters would be utilizing resources and serving traffic so if cost is an issue, you might need to determine if this setup is worth the cost. A service-level agreement (SLA) might also drive you to pursue this option. If you can not meet your client's needs in an agreed amount of time, you might have to make sure that you are using the location of your data to help make that possible.

### 8.6.4 Active-Passive

Another common setup is to have an active cluster in a data center with another sitting in another location. That passive cluster does not take any traffic from end-users, only data replicated from the other cluster. End-users would only utilize the passive cluster if the active cluster failed and passive would be made active for traffic from users. One of the biggest concerns with this setup is the lack of usage (hopefully!) of the passive cluster resources. The cost of data sitting idle in most cases is a consideration when you evaluate the value of your data. Is a failed cluster worth the overhead of a complete copy of the data? Figure 8.9 shows an example of how one data center would have a cluster that is idle in an Active-Passive setup.

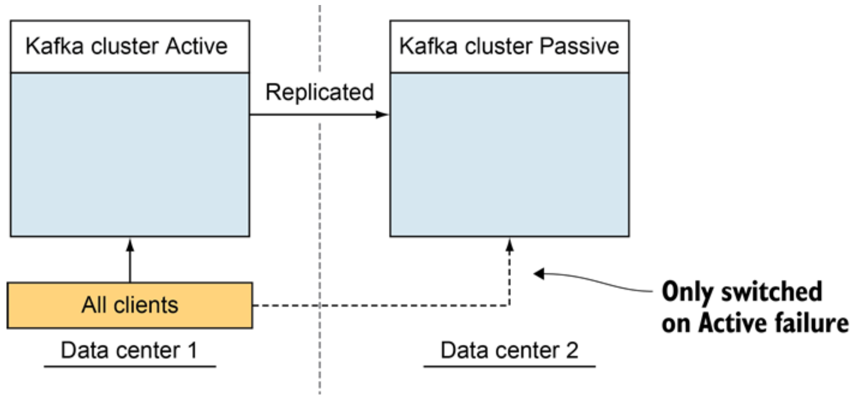


**Figure 8.9 Active-Passive Clusters in Different Data Center Locations**

Figure 8.9 is different from an Active-Active setup in that data in the active cluster in data center 1 would be replicated to the passive cluster. All traffic from clients go to the one active cluster during normal operation. In the event of a data center 1 failure, end-users would be routed to the cluster in data center 2. After a recovery of the failed data center, there are a couple of options. Data can be replicated from data center 2 back to data center 1 and end-users rerouted back to the previous data center. Or, the once passive cluster could be made the one active cluster and work in reverse.

## 8.7 Cloud and Container Based Storage Options

While we talked about not sharing drives between your operating system and your Kafka log directories in Chapter 6, we did not address the types of instances to use in environments that could be more ephemeral for storage. With AWS being a popular cloud deployment option, let's focus on a specific example even though these points would be similar for persistent disk solutions on other cloud providers.

### 8.7.1 Amazon Elastic Block Store

In regards to AWS, some users might be tempted to use local instance storage only when planning out their own cluster. Since Kafka was built to handle broker failures, the cluster more than likely would be able to recover on newly created broker instances as long as the cluster was otherwise healthy. This solution is straightforward for the most part, you start a new broker node and the cluster will work on gathering data for its partitions from the other brokers. This usually cheaper storage option might be worth the impact of the network traffic increase and any time delays for retrieving data vs other options.

However, Elastic Block Store (EBS) could be a safer option for your use-case. In case you are not familiar with the EBS service, it provides persistent block storage that helps data remain even in the case of a compute instance failure. Besides the fact that you can reassign the EBS volume to a new instance, the broker would only have to recover from messages that were missed during downtime vs the entire data partitions having to be replicated to that node. Amazon also offers instances that are optimized for writing Kafka logs: ie that are built for sequential I/O that is used in writing log files. Confluent shared a study on deplopyments with AWS considerations in which they mentioned a current image type of 'st1'. The benchmarked performance of the st1 EBS was better than instance storage.

Another concern to address is that your broker storage is spread across multiple availability zones (AVs). While this does not address a region failure, a failed AV will not cause your cluster to be unavailable.

As always, remember that Kafka itself keeps evolving and reacting to needs that users run into as daily challenges. KIP 392 shows an item that was under discussion at the time of writing that seeks to help address the issues of a Kafka cluster spanning data centers titled: 'Allow consumers to fetch from closest replica'. Be sure to check out recent KIPs from time to time to see how Kafka is evolving in exciting ways!

### 8.7.2 Kubernetes Clusters

Dealing with a containerized environment, you might run into similar challenges as you would in the cloud. If you hit a poorly configured memory limit on your broker, you might find yourself on an entirely new node without your data unless you have your data persisted correctly! Unless you are on a sandbox environment in which you can lose data, persistent volume claims will be needed by your brokers to make sure that your data will survive any restarts, failures, or moves. While the broker instance container might change, you should be able to claim the previous persistent volume (similar to how you can use EBS in AWS after a failure).

Kafka applications will likely use the StatefulSet api in order to maintain the identity of each broker across failures or pod moves. This static identity also helps us claim the same persistent volumes that were used before our pod was down.

There are already Helm charts to help you get started with a test setup if you are just exploring Kubernetes. Confluent does have plans to release a Confluent Operator to help as well. The scope of Kubernetes is relatively large to cover in our discussion, but the key concerns are present whatever environment you are in. Your brokers have an identity in the cluster and is tied to your data that each is related to. In order to keep your cluster healthy, you need to have those brokers maintain ways to identify their broker managed logs across failures, restarts, or upgrades.

## 8.8 Summary

In this chapter you learned:

- How to use retention in Kafka for your data
- Tools to leverage for help with data movement
- Data architectures that Kafka can enable

# 9 Administration: Cluster Tools, Logging, and Monitoring

***This chapters covers***

- Administration-client options
- Monitoring the cluster with JMX
- Logging management for our application logs
- Cluster management tool options
- Using interceptors with headers

We have spent time focused on brokers in depth in chapter 6 as well as other client concerns throughout the earlier chapters. But even using some of the best practices discussed, your environment might not always fit the ideal situation. The best way to keep your cluster moving along is to get to know your specific data as well as monitoring what is actually going on in your clusters. Even though running Kafka is not exactly like writing your own Java applications per se, it still requires monitoring log files and looking at what is happening at runtime with your workloads.

## 9.1 Administration clients

So far, we have taken almost all of the actions that have changed our cluster with the command line tools that are bundled with Kafka. Overall, you will likely need to be comfortable with a shell environment to set up and install Kafka. But there are some useful options that can help if you want to branch out from these provided scripts.

### 9.1.1 Administration in code with AdminClient

One useful client to look at is the AdminClient. Although we have done work on our brokers from the command line and used the shell scripts that have been provided in previous chapters, sometimes your automation tools can leverage this Java client instead of shelling out into a sub-command. If you already have the kafka-clients.jar that we used for Producer and Consumer Java clients, you already have this client as well! If you do not already have the jar, please use the pom.xml as part of your Maven project or look in the `share/` or `libs/` directory of the Kafka package you installed when you started running Kafka local on your machine.

Let's look how we could do a command we have completed before, but this time with the AdminClient to create a new topic. Listing 9.1 shows and replicates the command in chapter 2 that you would have invoked from the command line.

**Listing 9.1 Creating the selfserviceTopic Topic by command line**

```
bin/kafka-topics.sh --zookeeper localhost:2181 \ //        ❶
    --create --topic selfserviceTopic --partitions 2 \   ❷
    --replication-factor 2    ❷
```

❶ Using kafka-topic shell script in order to create a new topic

❷ Including our custom integers for the number of partitions and replicas for our topic

Though this command-line example should work, in this specific scenario, you want to support a self-service application written in Java. As part of this portal, you allow developers to create topics on a development cluster. Your form might take in the following set of information: a name for the topic, the number of partitions, and the number of replicas. Figure 9.1 shows an example of how this application might be set up for end-users. Once the user submits the webpage form, the admin-client Java code runs to create a new topic.

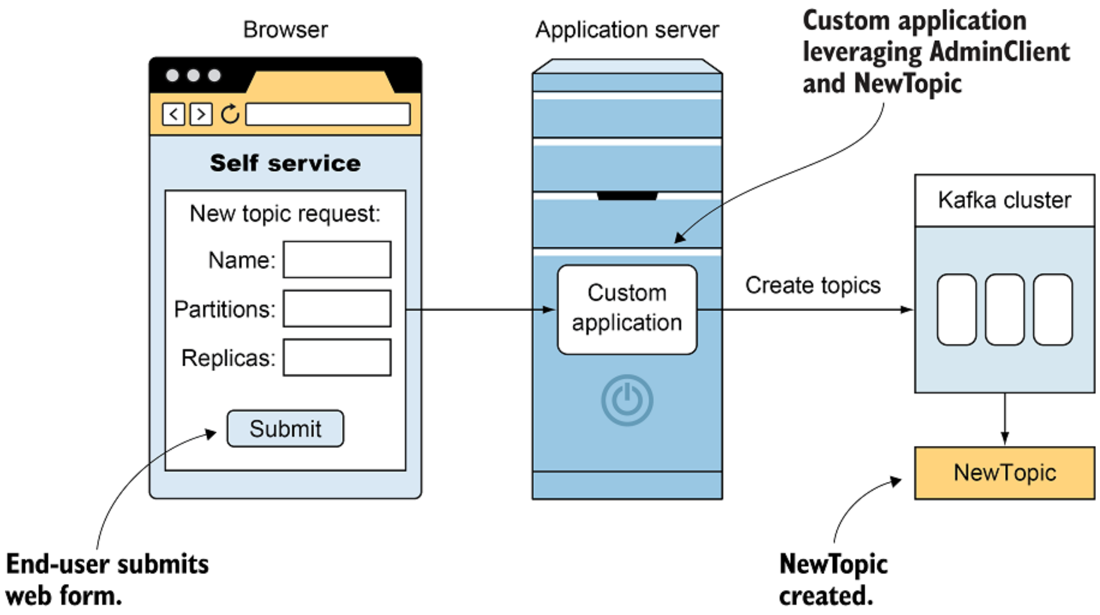


**Figure 9.1 Self Service Kafka Web Application**

In this example, you could add logic to make sure that naming conventions for topics fit a certain pattern if you had such a business requirement. This could be a way to maintain more control over your cluster rather than users working from the command-line tools. To start, you need to create a NewTopic object. This constructor takes three arguments: - the name of the topic - the number of partitions - the number of replicas Once you have this information, you have to leverage the AdminClient object to complete this work. The AdminClient takes a Properties object that uses names you are probably familiar with using any other client, like `bootstrap.servers` and `client.id`. Note that the class `AdminClientConfig` holds constants for configuration values such as `BOOTSTRAP_SERVERS_CONFIG` as a helper for those names. After calling the method createTopics on the client, notice that the result, `topicResult`, is actually a Future. Listing 9.2 shows how you leverage the `AdminClient` class to create a new topic called 'selfserviceTopic'.

## Listing 9.2 Using the AdminClient Create a Topic

```
NewTopic requestedTopic = new NewTopic("selfserviceTopic", 2,2);     ❶
AdminClient client = AdminClient.create(props); //                      ❷
CreateTopicsResult topicResult =
client.createTopics(Collections.singleton(requestedTopic)); //          ❸

// topicResult.values().get("selfserviceTopic").get(); //               ❹
```

❶ Create a NewTopic object with the topic name, two partitions, and two replicas

❷ Create an AdminClient, which is the client interface to the cluster

❸ Invoke createTopics method on the client which returns a Future object

❹ The commented out line shows how to get a specific Future for the topic 'selfserviceTopic' that we just created

At this time there is no synchronous API, so if you wish to block progress based on the result of that action, you will need to call the get() function. In our case, that would mean starting with the `topicResult` variable and evaluating the Future that was returned for that specific topic name.

Because this API is still evolving, the following list of tasks that you can complete with the AdminClient highlights only some of the common functions for this client that are available at the time of writing.

List of Client Admin Tasks

- alter config
- create/delete/list ACLS
- create partitions
- create/delete/list topics
- describe/list consumer groups
- describe the cluster

For example, where you do not have users familiar with the command line or a need for them to download the Kafka installation locally for access to scripts, using the AdminClient in your own user-facing applications might be a good alternative to let your users work on your cluster.

### 9.1.2 Kafkacat

Kafkacat is a handy tool to have on your workstation, especially if you usually have to connect remotely to your clusters. At this time, it focuses on being a producer and consumer client that can also give you metadata about your cluster. If you ever wanted to quickly work with a topic and didn't have the entire Kafka tools downloaded to your current machine, this executable will help you avoid the need to have those shell or bat scripts!

Listing 9.3 shows how you can quickly get data into a topic using kafkacat, compared with the `kafka-console-producer` script that you used in chapter 2.

**Listing 9.3 Kafkacat Producer usage**

```
kafkacat -P -b localhost:9092 -t selfserviceTopic      ❶

// vs the shell script we used before

bin/kafka-console-producer.sh --broker-list localhost:9092 \
  --topic selfserviceTopic //                               ❷
```

❶ By sending in a broker from your cluster as well as you topic, you are able to write messages from the 'selfserviceTopic' in our example with the -P flag

❷ A reminder of the same functionality with the console producer command

Notice that for the kafkacat command the `-P` argument is passed to set the command into producer mode, which is going to help us send messages to the cluster. The `-b` flag is used to pass in your broker-list and the `-t` is passed with the topic name that you wish to send messages to. Because you know that you also might want to test the consumption of these messages, let’s look at how you can also use kafkacat to consume messages (listing 9.4). As before, you can see the comparison between running the kafkacat command vs the `kafka-console-consumer` command. Notice that while the `-C` command is used to flag that the consumer mode is being used, the broker information is sent in with the same parameter as the producer mode. You might find that kafkacat has more consistent options between commands that might improve your workflow. For example, the producer and consumer client both take a parameter for a broker, one is named `broker-list` whereas the other is `bootstrap-server`.

**Listing 9.4 Kafkacat Consumer usage**

```
kafkacat -C -b localhost:9092 -t selfserviceTopic     ❶

// vs the shell script we used before

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic selfserviceTopic   ❷
```

❶ By sending in a broker from your cluster as well as you topic, you are able to read messages from the 'selfserviceTopic' in our example with the -C flag

❷ A reminder of the same functionality with the console consumer command

Having a quick way to test your topics and gather metadata on your cluster might make having this small utility a nice-to-have on your workstation. But by this point, you might be wondering if there are any other tools that you can use that are not command-line driven. And you would be right! For those of you that like REST, there is Confluent’s REST Proxy.

### 9.1.3 Confluent REST Proxy API

Sometimes the users of your cluster might prefer to use APIs that are RESTful, because it is a common way to work between applications, either due to preference or ease-of-use. Also, some companies with very strict firewall rules about ports might express caution with opening more ports like those we have used so far for the broker connections, for example, 9092. One such option is to use the Confluent REST Proxy API (figure 9.2). The functionality of this proxy is similar to kafkacat utility that we just discussed.

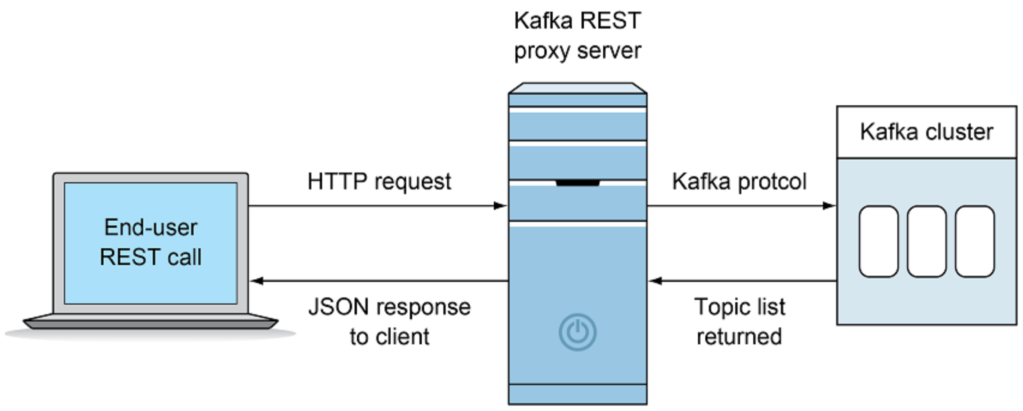


**Figure 9.2 Confluent REST Proxy**

At the time of this writing, the administration functions are limited more to querying the state of your cluster. The official documents list administration options as future supported features. This proxy is a separate part of Kafka that you would likely host on its own server if you stood it up for production usage.

To test out how to leverage the REST proxy, let’s start it up. For this to work, you will need to already have your ZooKeeper and Kafka instances running before you start the proxy, as shown in listing 9.5.

**Listing 9.5 REST Proxy startup**

```
./bin/kafka-rest-start ./etc/kafka-rest/kafka-rest.properties    ❶
```

❶ Running this command from your installed Kafka folder will start-up the REST endpoint for our testing.

Because you’re familiar with listing topics, let’s look at what your invocation looks like now, using a command like curl to hit an HTTP endpoint (listing 9.6).

**Listing 9.6 Curl call to REST Proxy for Topic list**

```
curl -X GET -H "Accept: application/vnd.kafka.v2+json" \ //                 ❶
 localhost:8082/topics //                                                    ❷


// Output:
["__confluent.support.metrics","_confluent-metrics","_schemas","alert"]   ❸
```

❶ Leveraging the Accept header helps specify the most specific format and version

❷ The endpoint /topics is our target to retrieve a list

❸ The list of topic output includes the alert topic we have been working with this chapter as well as the internal topics used by Kafka itself.

Because this is a 'GET' request, you can copy the following URL into your browser and see the result: localhost:8082/topics. But using a tool like curl allows you to control the header you send with your request. By using 'Accept', you are telling our Kafka cluster what format and version we are using. The `v2` is the API version we are using and `json` is the serialization format that pertain to our metadata requests.

Because this is an evolving API, please keep up with the API reference as newer versions will come out and possibly have new features.

## 9.2 Running Kafka as a systemd Service

One decision to make in regard to running Kafka is how you want to perform starts and restarts of your brokers. If you are used to managing your servers as a service on a Linux-base with a tool like Puppet, you may be familiar with installing service unit files and would likely leverage that knowledge to create your running instances. If you are not familiar with systemd, it initializes and maintains components throughout the system. One common way to define Zookeeper and Kafka are as unit files that are used by systemd.

Listing 9.7 shows an example service unit file that would start your ZooKeeper service when the server starts. It will also restart after an abnormal exit. In practice, this means something like a `kill -9` command against the pid of this process would trigger a restart of the process.

**Listing 9.7 Zookeeper Unit File**

```
[Unit]
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
ExecStart=/opt/kafka/bin/zookeeper-server-start.sh /opt/kafka/config/zookeeper.properties   ❶
ExecStop=/opt/kafka/bin/zookeeper-server-stop.sh   ❷
Restart=on-abnormal   ❸

[Install]
WantedBy=multi-user.target
```

❶ This section would make sure that your ZooKeeper service is running. This dependency makes one sense if both ZooKeeper and Kafka were on the same server.

❷ ExecStart captures the start command that will be run and should be similar to what you manually ran so far to start ZooKeeper

❸ ExecStop captures how to property shut down the ZooKeeper instance

\ <1> <2> <3> caused the process to fail

The unit file should look very familiar to how you have been starting ZooKeeper so far in our examples. Listing 9.8 shows an example unit file for a Kafka broker. This file adds a dependency on ZooKeeper being started before it would be started itself.

**Listing 9.8 Kafka Unit File**

```
[Unit]
Requires=zookeeper.service
After=zookeeper.service    ❶

[Service]
Type=simple
ExecStart=/opt/kafka/bin/kafka-server-start.sh \
/opt/kafka/bin/config/server.properties > /opt/kafka/broker.log 2>&1    ❷
ExecStop=/home/kafka/kafka/bin/kafka-server-stop.sh    ❸
Restart=on-abnormal    ❹

[Install]
WantedBy=multi-user.target
```

❶ This section would make sure that your ZooKeeper service is running. This dependency makes one sense if both ZooKeeper and Kafka were on the same server.

❷ ExecStart captures the start command that will be ran and should be similar to what you manually ran so far to start Kafka

❸ ExecStop captures how to property shutdown the Kafka instance

❹ The Restart condition would run ExecStart if an error condition caused the process to fail

The trick to this unit file is the fact that it is requiring the zookeeper.service you previously defined to be available and running before the Kafka service is started. This is a simplified example because your services are likely not going to be running on the same server. Another item to note is that you can add a `User` field and would likely want to, so that your application would run under that account and not as root. Though you can pick your own user name for the service, it is not uncommon to see the user as `kafka`.

Listing 9.9 shows that because your unit files are defined, you can how manage the services with systemctl commands.

**Listing 9.9 Kafka Startup with Systemctl**

```
sudo systemctl start zookeeper    ❶
sudo systemctl start kafka //     ❷
```

❶ Using systemctl command to start your ZooKeeper service

❷ Using systemctl command to start your Kafka service

If you are using the example files that come with downloading the Confluent bundle, once you unzip the folder, check inside the root folder `../lib/systemd/system` to see examples of other service files that you can leverage for other services, like Connect, Schema Registry, and the Rest API to name a few.

## 9.3 Logging

Besides Kafka's event logs that hold your event data, one of the other topics that you need to remember is the application logs, which Kafka produces as part of being a running program. The logs addressed in this section are not the events and messages from Kafka servers, but the output of the operation of Kafka itself. And we can not forget about Zookeeper either! Filling up a filesystem with logs that would not be helpful for timely troubleshooting or auditing is something to avoid.

### 9.3.1 Kafka application logs

While you might be used to one log file for an entire application, Kafka has multiple log files that you might be interested in or need for troubleshooting. Due to multiple files, you might have to look at modifying different log4j appenders to maintain a the view of your operations that you need.

**NOTE** **Which Kafka Appender**

The kafkaAppender name is not the same thing as the KafkaAppender itself. KafkaAppender. To use that as your appender you would need to update the following line as well as include dependencies for the clients and appender jars of the same versions:

```
log4j.appender.kafkaAppender=org.apache.kafka.log4jappender.KafkaLog4jAppender


<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-log4j-appender</artifactId>
    <version>2.2.1</version>
</dependency>
```

instead of the value `org.apache.log4j.ConsoleAppender` class. This is an interesting take on putting you logs directly into Kafka as some solutions parse the log files themselves and send to Kafka.

Be default, the server logs will continue being added to your directory as new logs are produced. No logs are removed - and this might be your preferred behavior if these files are needed for auditing or troubleshooting. However, if you do want to control the number and size, the easiest way is to update the file `log4j.properties` before you start your broker server. Listing 9.10 sets two important properties for the kafkaAppender: MaxFileSize and MaxBackupIndex. It is recommended that you do have a specific plan on evaluating your needs and to make sure that you do not fill your disk space on logfiles alone!

**Listing 9.10 Kafka server log retention**

```
log4j.appender.kafkaAppender.MaxFileSize=100KB        ❶
log4j.appender.kafkaAppender.MaxBackupIndex=10        ❷
```

❶ This settings helps define the size of the file in order to determine when to create a new log file.

❷ This value helps set the number of older files to keep. This might be helpful if you want more than just the current log if needed for troubleshooting.

Note that modifying the `kafkaAppender` changes only how the server.log file is treated. If you want to apply different file sizes and backup files numbers for various Kafka-related files, you can use the appender to log file name table to know which appenders to update. Note that the appender name in the left column will be the logging key which will effect how the log files on the right will be stored on the brokers.

**Table 9.1 Appender to Log Pattern**

| Appender Name | Log File Name |
|---|---|
| kafkaAppender | server.log |
| stateChangeAppender | state-change.log |
| requestAppender | kafka-request.log |
| cleanerAppender | log-cleaner.log |
| controllerAppender | controller.log |
| authorizerAppender | kafka-authorizer.log |

Changes to the `log4j.properties` file will require a restart of the broker, so it is best to determine your logging requirements before starting your brokers for the first time if possible. You could also change the value with JMX but the value would not be persistent across restarts of your broker.

As you look at managing your logs, it is important to consider where you are sending your logs. Feeding error logs from Kafka into the same Kafka instance is something you want to avoid. While this advice might seem simple, it is amazing to see how many configuration values in other tools, like Flume, might be pointing to brokers in the same cluster. Evaluate where you errors are shipped to other tools and if that transport relies on the same cluster you are trying to monitor itself.

While we have focused on Kafka itself, we need to address our ZooKeeper logs as well. Since ZooKeeper is running and logging data just like our brokers, you will need to be mindful of logging output for those servers as well.

### 9.3.2 Zookeeper logs

Depending on your install of ZooKeeper, you also want to make sure to look at the configuration of your installation: default configuration for ZooKeeper does not remove log files. However, your Kafka install may have added that for you. Either way, it is a good idea to make sure that the retention of the following application logs are what you would need for troubleshooting.

- `autopurge.snapRetainCount` This is the number of recent snapshots and the related transaction logs in the dataDir and dataLogDir locations. Once the number is exceeded, those log files are deleted. Depending on your needs, you might want to keep more or less based on your troubleshooting needs or if used in audit scenarios.
- `autopurge.purgeInterval` For cleanup to occur, you need to set this value to an integer greater than the default of 0. This will be the number of hours for how often the purge will take place.
- `snapCount` ZooKeeper logs its transactions to a transaction log. Setting this value will determine the amount of transactions that are logged to the one file. If you see issues with your file size, you might want to set this number less than the default (10,000).
- `preAllocSize` Your mileage might vary with this setting, but you can also modify the block size to allocate a smaller size in comparison to the default of 64 MB.

In general, it is also always a good idea to have a strategy to monitor your free disk space. In most cases, filling a filesystem can be avoided. There is nothing more frustrating that having a server go down because of something that can be automated before it becomes an issue. In general, you might already have you preferred solution to log rotation and cleanup that might be something more than log4j. `logrotate` is a helpful tool that you can look at that enables options such as log rotation and compression of logs files.

Log file maintenance is a core administration duty. However, there is also another task that might require your attention as you start to rollout your Kafka cluster in existing or isolated environments: making sure your clients can connect to your brokers.

## 9.4 Firewall

Depending on your network, you might serve clients that exist inside your network or those out of the network where your Kafka brokers are set up.

Kafka brokers can serve multiple ports. For example, the default examples for a plain text port is 9092. An SSL port can be set up on that same host at 9093. Both of these ports might need to be open depending on which ports your clients are using for connecting to your brokers.

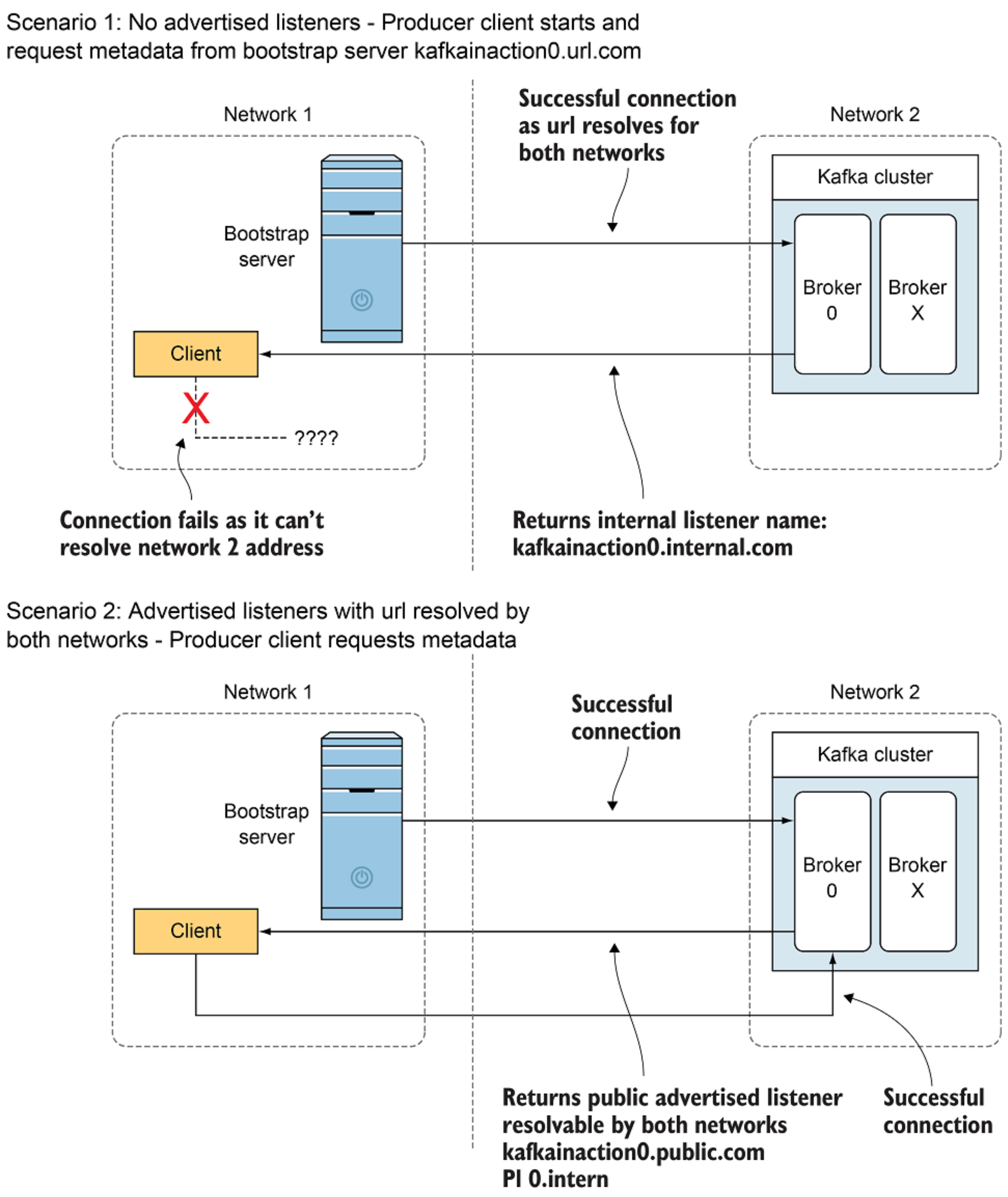
In addition, Zookeeper ports include 2181 for client connections. Kafka depends on Zookeeper as well as some of your admin command line tools, so it might be good to open the ports to clients as well as between the servers themselves. Port 2888 is used by follower Zookeeper nodes to connect to the leader Zookeeper node and the port 3888 is also used between Zookeeper nodes to communicate.

Also, if connecting remotely for JMX or other Kafka services (such as the REST Proxy), remember to account for any exposure of that port to other environments or users. In general, if you use any command line tools and include a port on the end of your hostname to ZooKeeper or Kafka bootstrap servers, you want to make sure you can test if the connection is allowed, especially if a firewall is in your environment.

### 9.4.1 Advertised listeners

One error when connecting that often appears like a firewall issue is when you are using the listeners and advertised.listeners properties. Your clients will have to use that hostname to connect if given-so it will need to be reachable however you setup your rules. For example, let’s look at an example `listeners` vs `advertised.listeners` where your values might not be the same.

Let’s imagine you are connecting to a broker and can get a connection when the client starts up, but not when it attempts to consume messages? How could you have this behavior that appears inconsistent? Remember that whenever a client starts up, it connects to any broker to get metadata about which broker to connect to. The first initial connection from the client uses the information that is located in the Kafka `listeners` configuration. What it gives back to the client to connect to next is the data in Kafka `advertised.listeners`. This makes it so your client might be connecting to a different host. Figure 9.3 shows how the first connection attempt uses one hostname at first but then uses a different hostname on its second connection. This second hostname was actually given to the client from its first initial call as the new location to connect to.



Scenario 1: No advertised listeners - Producer client starts and request metadata from bootstrap server kafkainaction0.url.com
Network 1
Successful connection as url resolves for both networks
Network 2
Kafka cluster
Bootstrap server
Broker 0
Broker X
Client
????
Connection fails as it can't resolve network 2 address
Returns internal listener name: kafkainaction0.internal.com
Scenario 2: Advertised listeners with url resolved by both networks - Producer client requests metadata
Network 1
Successful connection
Network 2
Kafka cluster
Bootstrap server
Broker 0
Broker X
Client
Returns public advertised listener resolvable by both networks kafkainaction0.public.com PI 0.intern
Successful connection

**Figure 9.3 Advertised listeners difference from listener**

`inter.broker.listener.name` is also an important setting to look at that will determine how the brokers connect across the cluster to each other. If the brokers cannot reach each other, replicas will fail and your cluster will not be in a good state, to say the least! Another excellent explanation of advertised listeners was written in an article by Robin Moffatt if you wish to dig into more details.

## 9.5 Metrics

In chapter 6, you looked at an example of setting up a way to see some JMX metrics from your application. The ability to see those metrics is the first step. The harder part is picking out which are important to your cluster. Let's take a peek at some that are likely to highlight areas of concerns.

### 9.5.1 JMX Console

To be able to browse the metrics that are available, it is possible to use a GUI to look around at the exposed metrics. In my example, I am using VisualVM. Other tools are available that will help as well, like JConsole—another popular tool. Looking around at what JMX metrics can help you discover points of interest that you might choose to add alerting. If you install VisualVM, make sure that you go through the additional step of installing the MBeans browser. You can do this by going to the menu bar Tools → Plugins.

As noted in chapter 6, make sure that you have a `JMX_PORT` defined for each broker you want to connect to. This can be done with an environment variable in the terminal like so: `export JMX_PORT=1099`. It is interesting to note that if you have ZooKeeper running on the same server as your broker, its configuration might rely on this same property. Make sure that you scope it correctly to be separate for each broker as well as each ZooKeeper node.

`KAFKA_JMX_OPTS` is also another option to look at if you want to connect remotely to your brokers. Make sure that you note the correct port and hostname. The following example uses port 1099 and the localhost as the hostname. Note that the other parameters allow you to connect without SSL and not have to authenticate. Listing 9.11 shows an example setting `KAFKA_JMX_OPTS` with various arguments.

**Listing 9.11 Kafka JMX Options**

```
KAFKA_JMX_OPTS="-Djava.rmi.server.hostname=127.0.0.1    ❶
-Dcom.sun.management.jmxremote.local.only=false   //       ❷
-Dcom.sun.management.jmxremote.rmi.port=1099  //           ❸
-Dcom.sun.management.jmxremote.authenticate=false  //      ❹
-Dcom.sun.management.jmxremote.ssl=false" //               ❹
```

❶ Setting the hostname for the rmi server to be localhost

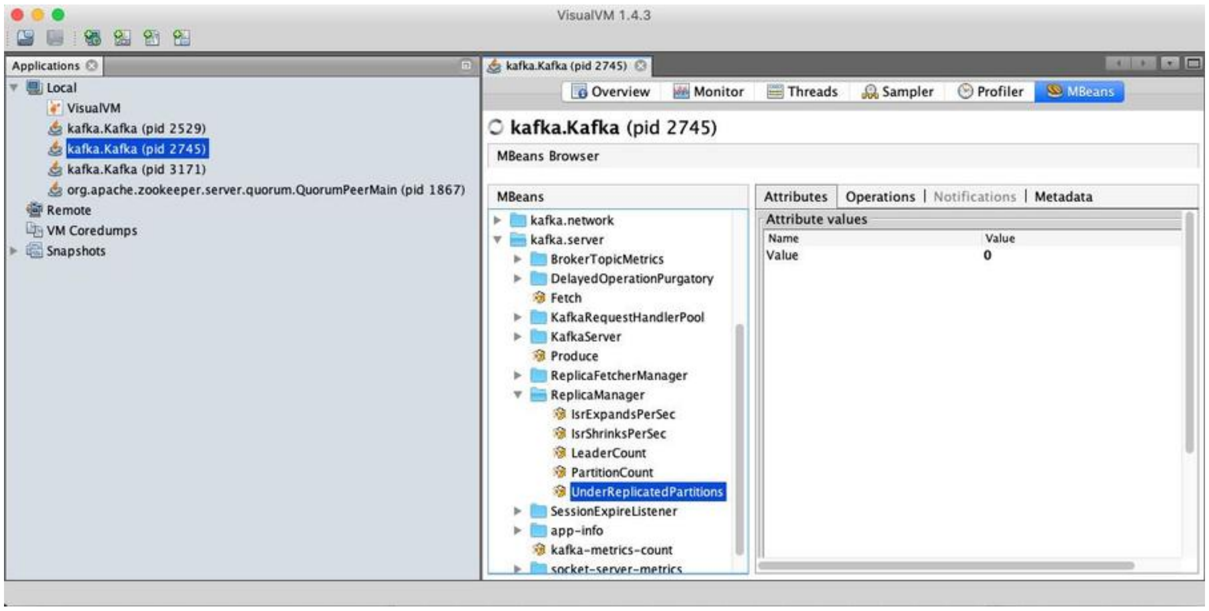
❷ Allowing remote connections

❸ The port you are exposing for JMX

❹ Turning off authenticate and SSL checks

Let's take a look at one of the metrics that is a key broker metric and how it looks in VisualVM (figure 9.4), which shows using the MBeans browser tab to see the value of

UnderReplicatedPartitions. Using the names in this book such as `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`: you drill down what looks like the folder structure starting with kafka.server.



| Name | Value |
|---|---|
| Value | 0 |

Figure 9.4 UnderReplicatedPartitions in VisualVM

Continuing on, you will then find the ReplicaManager with the name 'UnderReplicatedPartitions'. Now that you have seen the how to browse to specific values, let’s go into detail about some of the most important things to look at on your servers.

## 9.5.2 Important broker metrics

Because Kafka brokers are built as part of a cluster, losing a broker might not be the most important metric to keep an eye on, as startling as it may seem at first. Let’s look at some metrics that might mean action may be needed to help restore your cluster.

- `kafka.controller:type=KafkaController,name=OfflinePartitionsCount` Because clients read and write to a leader partition on a broker-there always needs to be a leader for your clients to be successful. If the number of offline partitions is greater than 0, your ability to read and write to that partition is compromised. This situation can happen when all of the partitions that existed are on brokers that are down.
- `kafka.controller:type=KafkaController,name=ActiveControllerCount` For a single cluster, the value that you want to see for this metric is 1. There should only be one controller at a time. If you had two different stop lights controlling a busy intersection, you would not know which one to follow.
- `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions` For a specific partition, this metric being greather than 0 might mean trouble. Not only are you not getting the number of replica copies that you expect, which means you can handle fewer failure scenarios, but you might also be stopping your clients from producing new

messages to your topic. For example, if you have three replicas and you need acknowledgement from all before your produce request is satisfied, you will not have messages be placed in your topic.

- `kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs`
  This metric is non-zero when there are likely broker failures. When leaders are being elected after server startup or new topic creation, you might want to look and see why leaders would be changing. One concern with frequent leader elections is why would your leadership be rotating so much? Do you have a network connection issue that ZooKeeper decides to remove your broker from the group? Clients have to wait for leadership ownership to be resolved, so it is important to make sure you know why that rotation keeps moving.

There are many other metrics, as you can see in your MBeans browser. As you gain experience, running your cluster and running into your own issues, you will likely add to the preceding list of items that are important flags in your infrastructure that something might need alerted on and action taken.

## 9.5.3 Important producer metrics

Using the Java producer client (which include your code examples in this book) also means that you will have access to JMX metrics. But remember, the clients are their own applications. They will not provide an end-to-end look at the cluster health alone. Another important thing to note is that if producers are not running actively against the cluster, you will not get those metrics. Having a time-series database of past metrics (like Prometheus) might help you capture more than a point-in-time view of what your producers are doing. Let’s dig into some important producer metrics to keep in mind.

- `kafka.producer:type=producer-metrics,client-id=<YOUR_ID_HERE>,name=record-error-ra`
  The ideal value of this attribute is zero. This is an average for the number of messages that failed per second. The producer client has already exhausted its retry logic and no more attempts are made. These are failures and are something you will want to look into why-even if every message doesn’t matter. On a side note, you can also get this metric at the topic level as well.
- `kafka.producer:type=producer-metrics,client-id=<YOUR_ID_HERE>,name=request-latency`
  This value is again an average of how long a request takes in milliseconds. If your requests get longer and longer, you might be able to see issues start to come up in your clients.

### 9.5.4 Important consumer metrics

- `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<YOUR_ID_HERE>,name=f`
  As the name might suggest, this attribute is similar in purpose as the same named attribute for the producer. The average value will let you know how long messages are taking to get to your consumer client. If this number starts to get larger and larger, you will want to look into the cause of your numbers. For example, you might be able to change the amount of data you are pulling with each fetch request or it could indeed be a network issue that is causing a random spike in time.
- `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<YOUR_ID_HERE>,name=r`
  This value is the maximum number of records for a given partition in which the consumer is behind the producer. If you notice an increasing value over time, you might consider alerting on this. Despite its name, this metric might not do exactly want you want. Let's dig into more details about *consumer lag* and also at a different monitoring option.

### 9.5.5 Burrow for Consumer Lag Logic

Although you usually rely on Kafka being fast for event delivery, there can be times when a specific consumer client becomes behind in pulling current messages. Even if a client does not stop or fail on its own, sometimes it might just have more messages than normal to process than its normal workload. The number of messages from the last offset in Kafka for your partition and the offset that your consumer client is currently processing is called consumer lag.

Burrow is a tool that provides consumer-lag checking and seems to have been developed with a thoughtful mindset toward its subject. Burrow addresses the shortcomings of JMX metrics such as `records-lag-max` that would be produced by a client. I say thoughtful because Burrow's code shows the marks of experienced operators that have worked with Kafka enough to be able to automate more meaning behind this metric. One such issue that is addressed is that the lag metric itself would be driven by and impacted by the consumer client that was doing its own reporting. If a failure or delay occurred, you might see invalid data reported or no new JMX messages! No clients running would report no metrics and might not show your monitoring the real scenario.

Also, you might see that thresholds are hard to set for a topic if you do not know the normal lag of your consumers. My first attempt at a script for checking lag monitoring was a script similar to one shown in listing 9.12.

**Listing 9.12 Kafka Consumer Group Command to find lag usage**

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
--describe --group test-consumer   ❶

#Truncated example output //          ❷
TOPIC   PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
alert    0          3               3               0
```

❶ By doing a describe argument with my group-I will find a number for lag.

❷ Truncated example output showing the lag value for one consumer group

Per the output, you see the column heading 'Lag' and could choose to parse this output for your own monitoring. But after finding Burrow I left this type of script in the dust for lag monitoring. The ability to avoid setting a specific or hard-coded threshold number helped avoid managing various consumer-group-to-threshold values that I had no idea about regarding the normal rate of consumption. Your mileage may vary with Burrow, but it is an interesting tool to monitor your lag that avoids pitfalls of the JMX metrics themselves.

## *9.6 Tracing messages across a cluster*

The built-in metrics that you looked at so far can give you a great snapshot of current health, but what if you wanted to trace a single message through the system? What would you leverage to see a produced message and see its consumed status?

Let’s talk about a simple but straightforward model that might work for your requirements. Let’s say that you have a producer in which each event has a unique ID. Because each message is important, you do not want to miss any of these events. With one client, the business logic runs as normal and consumes the messages from the topic. It makes sure to log in a database or flat file the ID of the event that was processed. A separate consumer, let’s call it an auditing consumer in this instance, would fetch data from the same topic and make sure that there were no IDs missing from the processed entries of the first application. Though this process can work well, it might be something that you wish to avoid, which is adding logic to your business applications for this process to provide reconciliation of what was processed with what was in the topic. Figure 9.5 shows a different approach using Kafka interceptors.

In practice, the interceptor that we define is a way to add logic to the producer, consumer, or both by hooking into the normal flow of our clients, intercepting the record, and adding our custom data before it moves along its normal path. Our changes to the clients is configuration-driven and helps keep our specific logic out of our clients for the most part.

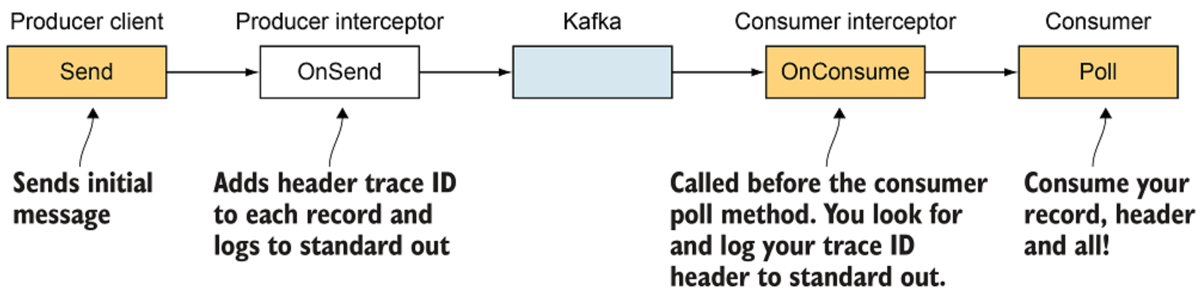

Producer client
Producer interceptor
Kafka
Consumer interceptor
Consumer
Send
OnSend
OnConsume
Poll
Sends initial message
Adds header trace ID to each record and logs to standard out
Called before the consumer poll method. You look for and log your trace ID header to standard out.
Consume your record, header and all!

**Figure 9.5 Interceptors for Tracing**

Let’s revisit the concept of interceptors that we touched on briefly in chapter 4 in introducing what producer interceptors could do for our messages. By adding an interceptor on both the producer and consumer clients that you are using, you can remove the monitoring logic from the application logic itself. The crosscutting concern of monitoring can hopefully be more encapsulated by this approach.

### *9.6.1 Producer interceptor*

It is also interesting to note that you can have more than one interceptor so you don’t have to include all of your logic in one class: you can add and remove others later. The order that you list the classes is important as that is the order in which logic will run. The first interceptor gets the record from the producer client. If the interceptor modified that record, note that it might not be the exact same as the first interceptor got.

Let’s start with looking at the Java interface `ProducerInterceptor`. You’ll add this new interceptor to your Alert producer that you used in previous examples in chapter 4. You will create a new class called `AlertProducerMetricsInterceptor` to add logic around alerts being produced, as in Listing 9.13. Implementing the interface `ProducerInterceptor` allows you to hook into the producers interceptor life-cycle. The logic in the `onSend` method will be called by the send from the normal producer client you have used so far. For your example, you are going to add a header that you will call a 'traceId'. Using a unique ID will help you confirm on the consumption side that you are seeing the same message on the end of your journey that you are producing in this step.

**Listing 9.13 AlertProducerMetricsInterceptor Example**

```
public class AlertProducerMetricsInterceptor
  implements ProducerInterceptor<Alert, String> { //                          ❶


    public ProducerRecord<Alert, String> onSend(ProducerRecord<Alert, String> record) {   ❷
        Headers headers = record.headers();
        String traceId = UUID.randomUUID().toString();
        headers.add("traceId", traceId.getBytes()); //                        ❸
        System.out.println("Created traceId: " + traceId);
        return record; //                                                     ❹
    }

    public void onAcknowledgement(RecordMetadata metadata, Exception exception) { //  ❺
        if (exception != null) {
            System.out.println("producer send exception " + exception.getMessage());
        } else {
            System.out.println(String.format("ack'ed topic=%s, partition=%d, offset=%d\n",
                    metadata.topic(), metadata.partition(), metadata.offset()));
        }

    }

 ...

}
```

❶ Implement ProducerInterceptor to hook into the interceptor life-cycle

❷ The onSend method is called by the producer client send method

❸ Add a custom header to the record in order to carry the generated ID across Kafka

❹ Return the record that you modified. This will include your new header

❺ onAcknowledgement will be called when a record is acknowledged or an error occurs

Implementing the interface `ProducerInterceptor` allows you to hook into the producers interceptor life-cycle. The logic in the `onSend` method will be called by the send from the normal producer client you have used so far. For this example, you are going to add a header that you'll call a 'traceId'. Using a unique ID will help confirm on the consumption side that you are seeing the same message on the end of your journey that you are producing in this step.

You also have to modify your existing `AlertProducer` class to register your new interceptor. The property name `interceptor.classes` will need to be added to the producer configuration with a value of our new class: `com.kafkainaction.producer.AlertProducerMetricsInterceptor`. Although I used the property name for clarity, remember that you can use the constant provided by the `ProducerConfig` class. In this case you would use `ProducerConfig.INTERCEPTOR_CLASSES_CONFIG`. Listing 9.14 shows this required modification.

**Listing 9.14 AlertProducer with Interceptor Configuration**

```
Properties props = new Properties();
...
props.put("interceptor.classes",    ❶
  "com.kafkainaction.producer.AlertProducerMetricsInterceptor");   ❷

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

❶ Use the property name interceptor.classes to set your interceptors

❷ The value can be one as in our case or else a comma-separated list

In this example overall, you have one interceptor that logs out a unique ID for your produced message. You will add this ID as a header to the record, so that when a consume sees this message, a corresponding consumer interceptor will log this ID that it has processed. The goal is to provide your own sort of end-to-end monitoring that is outside of Kafka. By parsing the application logs, you will see messages like the following that came from your `AlertProducerMetricsInterceptor` class as shown, in Listing 9.15.

**Listing 9.15 Alert Interceptor Output**

```
Created traceId: 603a8922-9fb5-442a-a1fa-403f2a6a875d    ❶
ack'ed topic=alert, partition=0, offset=1
```

❶ An example of our logged value added by the producer interceptor

Because you have completed your look at sending a message, you also need to see how to implement similar logic on the consumer end of your system as well. You want to validate that you can see that same header value that you added with the producer interceptor on the consumption end. Listing 9.16 shows implementation of the `ConsumerInterceptor` class to help retrieve this header.

### 9.6.2 Consumer Interceptor

**Listing 9.16 AlertConsumerMetricsInterceptor Example**

```
public class AlertConsumerMetricsInterceptor
  implements ConsumerInterceptor<Alert, String> { //          ❶

    public ConsumerRecords<Alert, String> onConsume(ConsumerRecords<Alert, String> records) {
        if (records.isEmpty()) {
            return records;
        } else {
            for (ConsumerRecord<Alert, String> record : records) {
                Headers headers = record.headers(); //         ❷
                for (Header header : headers) {
                    if ("traceId".equals(header.key())) {   ❸
                        System.out.println("TraceId is: " + new String(header.value()));
                    }
                }
            }
        }

        return records; //                                     ❹
    }
...

}
```

❶ Implementing ConsumerInterceptor is needed to allow Kafka to recognize our interceptor

❷ We are looping through each record's headers

❸ Once we find our custom header we added from the producer interceptor, we will log it to standard output

❹ We return the records to continue on the path of callers from our interceptor

In a similar fashion as your producer, you use a consumer-specific interface, `ConsumerInterceptor`, to make your new interceptor. You are looping through all of the records and their headers to find any that have your custom 'traceId' as the key and sending this to standard output. You also have to modify your existing `AlertConsumer` class to register your new interceptor. The property name `interceptor.classes` will need to be added to the consumer configuration with a value of our new class: `com.kafkainaction.consumer.AlertConsumerMetricsInterceptor`.

Listing 9.17 shows this required step.

**Listing 9.17 AlertConsumer with Interceptor Configuration**

```
public class AlertConsumer {

Properties props = new Properties();
...
props.put("group.id", "alertinterceptor");  //                     ❶
props.put("interceptor.classes", //                                ❷
  "com.kafkainaction.consumer.AlertConsumerMetricsInterceptor");  ❸

}
```

❶ Using a new group.id for this example to make sure we are starting with our current offsets (and not one from a previous group.id)

❷ The property name `interceptor.classes` is needed to add our custom interceptor

❸ Our custom class is added as the value. We can include a comma-separated list if we have more than one class we need to use.

Although I used the property name for clarity, remember that you can use the constant provided by the `ConsumerConfig` class. In this case you would use `ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG`. While we can seen the usage of interceptor on both ends of our flow, there is also another way to add functionality to client code: overriding clients.

### 9.6.3 Overriding Clients

If you control the source code clients that your other developers will use, you can also subclass an existing client or create your own that implements the Kafka Producer/Consumer interfaces. The Brave project has an example of one such client that adds tracing data. If you are not familiar with Brave, it is a library meant to help add instrumentation for distributed tracing. It has the ability to send this data to something like a Zipkin server, which can handle the collection and search of this data.

Both the producer and consumer clients are decorated to enable tracing, but you'll focus on the consumer client in the following example. As in the example with adding a traceId to records with a header, the code also leverages the usage of headers. The code in listing 9.18 is a mixed section of code examples from the `TracingConsumer` class. Developers wanting to consume messages with tracing will use an instance of `TracingConsumer` which takes a regular consumer client named `delegate` in this example. Following the Decorator design pattern, the tracing functionality will be added by the object named `kafkaTracing` and provide our needed behavior. This code is a real-world example of how tracing is accomplished in yet another way for Kafka messages.

**Listing 9.18 Brave Consumer Client Tracing**

```
final class TracingConsumer<K, V> implements Consumer<K, V> {

...
  final Consumer<K, V> delegate;  //                             ❶
  final KafkaTracing kafkaTracing; //                            ❷
...
  TracingConsumer(Consumer<K, V> delegate, KafkaTracing kafkaTracing) {  ❸
    this.delegate = delegate;
    this.kafkaTracing = kafkaTracing;
    this.tracing = kafkaTracing.tracing;
    this.injector = kafkaTracing.injector;
    this.remoteServiceName = kafkaTracing.remoteServiceName;
  }

}
```

❶ The Kafka consumer client is being used as a delegate

❷ The additional tracing object is added to our consumer. This would not be part of the normal default client.

❸ The constructor takes both arguments in order to make this instance of the consumer client your developers would leverage

This added behavior is not getting in the way of the normal client, it is adding the ability to add tracing information.

## 9.7 General monitoring tools

Because Kafka is a Scala application, it has the ability to leverage JMX and uses the Yammer Metrics library. This library was used to provide JMX metrics on various parts of the application, and you have seen some of the options you can evaluate already. But as Kafka usage has expanded, there are some tools out there that leverage the availability of using JMX metrics as well as administration-related commands to provide easier-to-manage clusters. Let's take a look at a few options.

### 9.7.1 Kafka Manager

Kafka Manager is an interesting project that focuses on managing Kafka as well as being a UI for various administration activities. One key feature is the ability to manage multiple clusters. While we look at the commands that we have run so far, we have been looking at one cluster. These commands can be harder to scale the more clusters you are running at once. Other features include inspection of your overall cluster state and the ability to generate and run partition reassignment. The tool can also deal with authenticating users with LDAP, which might be needed in your enterprise usage.

### 9.7.2 Cruise Control

Cruise Control was created by the developers at LinkedIn. Because they have thousands of brokers across their clusters, they have experience running Kafka clusters and have helped codify and automate some of their pain points over the years. A REST API can be used as well as a UI, so you have a couple of ways to interact with this tooling. Some of the most interesting features are how Cruise Control watches your cluster and can generate suggestions on rebalances based on workloads.

### 9.7.3 Confluent Control Center

Confluent Control Center is another web-based tool that can help you monitor and manage your clusters. But one item to note is that it is a commercial feature that would need an enterprise license for a production setup. If you already have a subscription to the Confluent Platform, there is no reason not to check it out. This tool uses dashboards and can help you identify message failures, network latency, and other external connectors.

### 9.7.4 General OS monitoring needs

Not all of your monitoring will be focused on the Kafka application itself. A couple of core operating system (OS) monitoring needs appear in order to provide a solid foundation for Kafka to run on. Let's look at a few specifics, which I'll touch on briefly.

- Disk Percentage Free: As discussed about in the logging section, having a filesystem fill up without notice is not a good surprise for your users. Alerting at less than 85 percent of disk space free in a good idea in most cases.
- File Handles: Another thing to be aware of is the number of open file handles that Kafka has on your system. There will be a file handle for each log segment file and also for network connections on your broker. In other words, the number can add up quickly. As you installed Kafka, hopefully you made sure you adjusted your file descriptors limits. While entering a large number like 128000 is not unheard of, it is a good idea to make sure that you have alerting on when that number is almost exhausted. If not, you will get a generic-looking error like the following: `java.io.IOException: Too many open files`.
- Swappiness: You will want to make sure that you do not have unnecessary swapping of RAM. Kafka likes to be fast! Though you can alert when swap usage is greater than 128MB, you might have to adjust your own levels based on any performance impacts in your cluster.

Tools like Nagios might already be in your enterprise and leveraged for some of these monitoring needs already.

Overall, Kafka provides you with many options to not only manage but to monitor your cluster. As your experience level grows with Kafka, you will add your own metrics to monitoring that you find important as well as you own ideas to avoid any unexpected outages. Distributed systems are difficult and the more experience you gain, the more your monitoring skills and practices will also likely improve.

## 9.8 Summary

In this chapter you learned:

- Administration client options to use
- Important monitoring JMX metrics
- How to manage our application logs
- About various cluster management tool options

# 10 Protecting Kafka

***This chapters covers:***

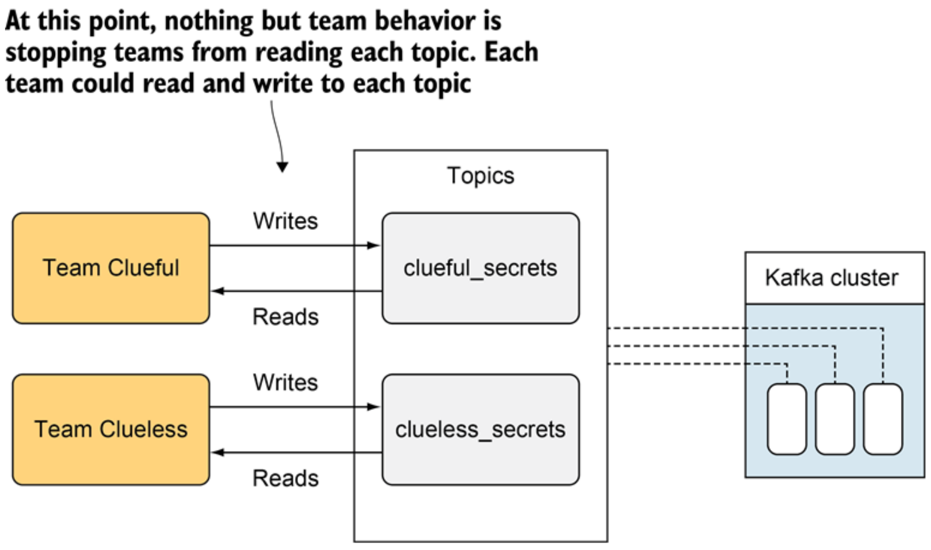
- Setting up SSL between our cluster and our clients
- Looking at SASL with Kerberos
- Using quotas
- Configuring Access Control Lists (ACLs)

This chapter will focus on keeping our data locked down to only those that need to read or write to it. Since security is a huge area to cover, we will talk about some basic concepts to get a general background on the options we have in Kafka. This will not be an exhaustive guide of security in general, but with that foundation, we will try to discuss some of the practical actions you would take in your setup. We will have client impacts as well as brokers and ZooKeeper to make our cluster more secure.

Of course, your data might not need some of the protections we discuss. If you are not tracking anything of a secure nature, i.e. personal information or financial data, then you might not even need this protections. Knowing your data is key to deciding if you need the tradeoffs of managing access. For a quick example, if you are handling any thing related to personal information like date of birth or credit card numbers, you will mostly like want to look at most of the security options discussed in this chapter. However, if you are only handling generic information such as the number of web clicks on a series of marketing campaigns, your cluster might not need to introduce features like SSL that might slow down your throughput while at the same time increasing you CPU needs.

Let's start with an example of data that we would want to protect. That would be data around solving the puzzle book Masquerade published in 1979. If you have not heard of this book, let's set the context of our task. By looking at 16 paintings, readers are meant to find clues to a treasure hunt. The reward: the location of a golden hare. As a competition-wide exercise, we have broken into teams and do not want other co-teams to access our own team's work. Spoiler alert: those of you who are familiar with this work in particular know that the location was indeed already found. But we will pretend like its still undiscovered in our use-case!

Starting out, each team picks their own topic names and shares that name with their own team members only. Without knowing which topic name to write and read from, your data is out of sight of the other teams. Each team starts out sending in their clues to what they assume is their 'private' topics. Over time, members of various teams start to wonder on the progress of other teams and if they have any clues that they don't. This is when the trouble starts.


At this point, nothing but team behavior is stopping teams from reading each topic. Each team could read and write to each topic
Topics
Writes
Team Clueful
clueful_secrets
Reads
Kafka cluster
Writes
Team Clueless
clueless_secrets
Reads

One tech savvy competitor who has used Kafka before reaches for his command line tools to find out the name of the other topics. After doing a list of topics, the user knows the topics of his rivals. For example, let's say that this one team member of Team Clueless looks at the topic of Team Clueful: topic clueful_secrets. With great happiness, all it takes is a consumer console command to be able to list all of the data that Team Clueful has been working on in the competition so far! But the bad actor did not stop there. In order to throw Team Clueful off of the trail, the actor also wrote false information into the channel. Now Team Clueful has bad data in their topic that is hindering their clue-solving progress! Since they are not sure who really wrote the message on the topic, the team has to go determine the false messages and lose valuable time they could be working on figuring out the grand-prize location.

How could we avoid this situation? Is there a way that only those clients that have permission would be able to read from or write to a topic? There really are two parts to our needs. The first is how to find out who a person is in our system. Not only who they are, but also making sure that the claimed identity of the user can be verified. The second part is that once we verify a user, what are they able to do in our system? We will dive into those topics deeper as we look at some of solutions provided with Kafka.

## 10.1 Security Basics

In regards to computer application security, you will likely discuss encryption, authentication and authorization at some point in your journey.

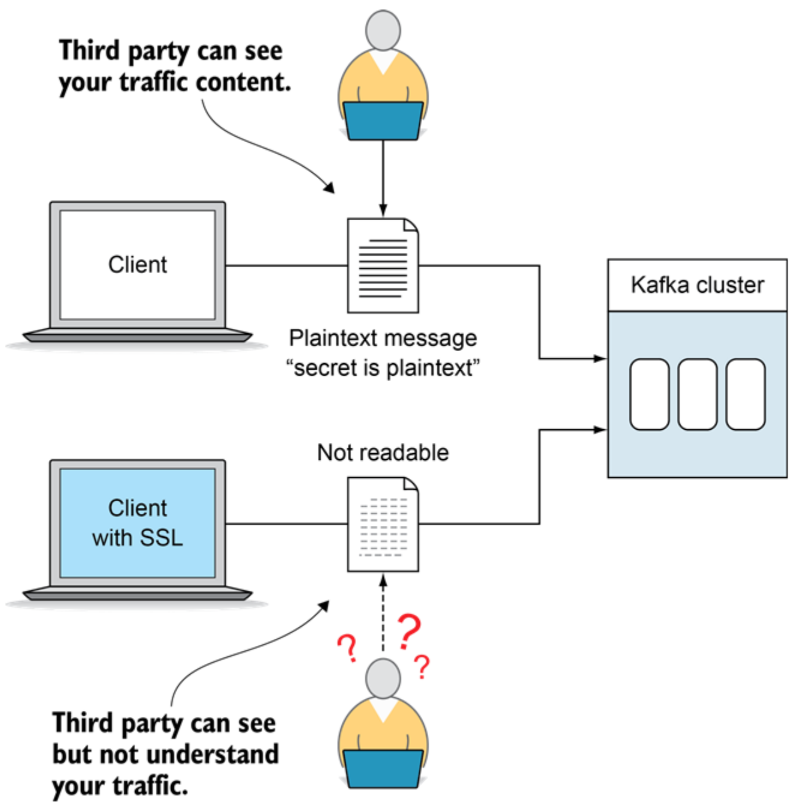
Encryption is a way of encoding information in a way that only authorized parties can access the content. It is important to note that encryption does not mean that others might not see your messages, but if they do, they will likely not be able to understand how to derive the original content that you are protecting. Many people will think of how they are encouraged to use a site that is HTTPS for online shopping on a WiFi network. Later, we are going to enable SSL for our communication not between a website and our computer, but between our clients and brokers.

Moving forward, let's talk about authentication first in a real-world scenario. In order to verify the identity of a user or application, we need to have a way to authenticate that user: a process to prove that they are who they claim they are. If you went to sign up for a library card, does the library issue the card to everyone without making sure the user is who they say they are? In most cases, the library would confirm the person's name and address with something like a government issued id and utility bill. This process is intended to make sure that others cannot claim another identity to use for their own purposes in an easy manner. If someone had claimed your identity in order to loan books and never return them, sending the fines your way, we can easily see a drawback of not confirming the user's claim.

Authorization, on the other hand, instead focuses on what the user can do. Continuing with our library example above, our card as an adult might give us different permissions than if it was for a user considered as a child. Access for online publications might be limited for that user to only terminals inside the library.

### 10.1.1 Encryption with SSL

So far all of brokers have supported plain text. In effect, there has been no authentication or encryption over the network. Knowing these facts, it might make more sense when reviewing one of the broker server configuration values. If you pick at the current server.properties file, you will find an entry like the following: `listeners=PLAINTEXT:localhost//:9092`. That listener is in effect providing a mapping of a protocol to a specific port on the broker. Since the brokers support multiple ports, this will allow us to keep the PLAINTEXT port up and running as we test adding SSL, or other protocols, on a new port. Having both ports will help make our transition smoother as we shift over a period of time away from PLAINTEXT.


Third party can see your traffic content.
Client
Plaintext message "secret is plaintext"
Kafka cluster
Not readable
Client with SSL
Third party can see but not understand your traffic.

At this point, we are starting with a cluster in which we do not have any security really baked into our cluster. Luckily, we can add various pieces to our cluster as we harden it against other teams. Setting up SSL between the brokers in our cluster and our clients is one place to start. No extra servers are directories are needed. No client coding changes either will be required as the changes will be configuration driven. We don't know how advanced other teams are with listening to our traffic on the same WiFi network with WireShark, so we know we might not want to send PLAINTEXT from our brokers to our clients. While this following setup is needed for Kafka security, those that have set up SSL or HTTPS in the past (and especially with Java) will find an approach that is a similar pattern to other client/server trust setups.

**NOTE** One item to note is that while Transport Layer Security (TLS) is what we will actually be dealing with, you will see most of the reference in your configuration as Secure Sockets Layer (SSL). While TLS is the successor of SSL, the label SSL has been maintained as the property name you will see in our work below. To avoid confusion between explanations and the actual values used, we will keep the term SSL through this material.

### *10.1.2 SSL Between Brokers and Clients*

In our previous examples on writing our clients and connecting to Kafka, we have not used SSL in order to connect. However, now we are going to look at turning it on for the communication between our clients and our cluster to encrypt our network traffic with TLS.

Let’s walk-through the process of what we are going to need to accomplish in order to get our cluster updated with this feature.

- Generate a key and a certificate for each broker
- Use a certificate authority (CA) to sign each broker’s certificate
- Add the CA to the client truststore
- Sign each cluster certificate and import that back into its keystore with the CA cert

One of our first steps is to generate a key and certificate for each of our brokers. Since you should already have Java on your machine-one option is to use the `keytool` utility that is part of a Java installation. The `keytool` tool is used in order to manage a keystore of keys and trusted certificates. The important part to note is the storage part as it might be good to think of it as a database that your JVM programs can lookup up this information for your processes when it’s needed. At this point, we are also going to leverage the ability to generate a key for our brokers.

**Listing 10.1 SSL Key Generation for a Broker**

```
keytool -genkey -noprompt \
    -alias localhost \
    -dname "CN=kia.manning.com, OU=TEST, O=MASQUERADE, L=Portland, S=Or, C=US" \
    -keystore kafka.broker0.keystore.jks \   ❶
    -keyalg RSA \
    -storepass masquerade \   ❷
    -keypass masquerade \
    -validity 365
```

❶ This will be the name of the keystore file that will hold our newly generated key

❷ We have used a password so that the store cannot be changed without it

After running this command, we have created a new key and stored it in the keystore file: kafka.broker0.keystore.jks.

Since we have a key that in a way identifies our broker, we need something to signal that we don’t just have any certificate issued by a random user. One way to verify our certificates is by signing them with a CA. You might have heard of CAs such as Let’s Encrypt, Entrust, Symantec, or GoDaddy to name a few. The role of a CA is to act as a trusted authority that certifies the ownership and identity of the public key. In our examples, we are going to be our own CA to avoid any needs of verifying our identity to a third-party. Our next step is to create our own CA.

**Listing 10.2 Creating our own Certificate Authority**

```
openssl req -new -x509 -keyout ca.key -out ca.crt \   ❶
  -days 999 -subj '/CN=localhost/OU=TEST/O=MASQUERADE/L=Portland/S=Or/C=US' \
  -passin pass:masquerade -passout pass:masquerade
```

❶ We create a new CA and want a key file as well as the cert file produced as files.

After we have created our own CA, we will use it to sign out clients' truststore and brokers' truststore. This generated CA is now something that we want to let our clients know that they should trust. Similar to the term keystore, we will use a truststore to hold this new information. We will use `openssl` as a tool to help us act as this CA. Now that we have generated our CA, we will use it to sign our certificates for our brokers that we have already made. First, we will export the cert that we generated earlier for each broker from the keystore, sign that with our CA, and then import both the CA cert and newly signed cert back into the keystore.

**Listing 10.3 Importing the CA cert into our truststores and signing our keystore**

```
keytool -keystore kafka.client.truststore.jks \   ❶
-alias CARoot -import -file ca.crt  -keypass masquerade -storepass masquerade

keytool -keystore kafka.server.truststore.jks \   ❷
-alias CARoot -import -file ca.crt  -keypass masquerade -storepass masquerade

keytool -keystore kafka.broker0.keystore.jks \   ❸
-alias localhost -certreq -file cert-file -storepass masquerade -noprompt

openssl x509 -req -CA ca.crt -CAkey ca.key \   ❹
-in cert-file -out cert-signed -days 365 \
-CAcreateserial -passin pass:masquerade

keytool -keystore kafka.broker0.keystore.jks \   ❺
-alias CARoot -import -file ca.crt -storepass masquerade -noprompt

keytool -keystore kafka.broker0.keystore.jks \   ❻
-alias localhost -import -file cert-signed -storepass masquerade -noprompt
```

❶ Adding our CA to the trusted client truststore

❷ Adding our CA to the trusted server truststore

❸ We extract the key from the broker0 keystore we created earlier

❹ We use the CA to sign this broker0 key

❺ We place import the CA certificate into our broker0 keystore

❻ We place the signed certificate into the broker broker0 keystore as well

These changes might seem involved, but after running through this process, it is something that you can automate when you are adding more brokers in the future.

**NOTE** While running the above commands, if your operating system or tool version has a different prompt that passed in above, note that it will likely have a user prompt appear after run your command. Our examples try to avoid these prompts but might be a way to enter in your passwords without having them remain in your terminal's history.

As part of our changes, we need to update the server.properties configuration file on the each broker as well.

## Listing 10.4 Broker server properties changes

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093   ❶
ssl.truststore.location=/var/ssl/private/kafka.server.truststore.jks   ❷
ssl.truststore.password=masquerade
ssl.keystore.location=/var/ssl/private/kafka.broker0.keystore.jks   ❸
ssl.keystore.password=masquerade
ssl.key.password=masquerade
```

❶ We have added the SSL broker port and left the older PLAINTEXT port

❷ We need to provide the truststore location for our broker as well as the password

❸ We need to provide the keystore location for our broker as well as the password

Changes are also needed for our client configurations.

## Listing 10.5 SSL Configuration for Clients

```
security.protocol=SSL   ❶
ssl.truststore.location=/var/private/ssl/client.truststore.jks   ❷
ssl.truststore.password=masquerade   ❸
```

❶ Make our protocol specific to use SSL

❷ The is the file path to the truststore we worked on creating for our client

❸ Since we used a password for our truststore, we referene it here

While testing your changes, you can also have multiple listeners setup for your broker. This also helps clients migrate over time as both ports can serve traffic before you drop the older PLAINTEXT port option for clients. One of the simplest ways to check our SSL setup is to quickly use the consoles producers and consumer clients in order to connect to a topic on our SSL port 9093.

## Listing 10.6 Using SSL Configuration for Command line Clients

```
kafka-console-producer.sh --broker-list localhost:9093 --topic testSSL \
 --producer.config client-ssl.properties   ❶
kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic testSSL \
 --consumer.config client-ssl.properties   ❷
```

❶ Use the producer.config option to let our producer know about the SSL details

❷ Use the consumer.config option to leverage our SSL config for consumers

One of the nicest features is that you can use the same configuration for both producers and consumers. As you look at the contents of this configuration, one issue that might spring into mind is the use of passwords in these files. The most straight forward option is to make sure that you are aware of the permissions around this file. Limiting the ability to read, as well as the ownership of the file, would be important to note before placing this configuration.

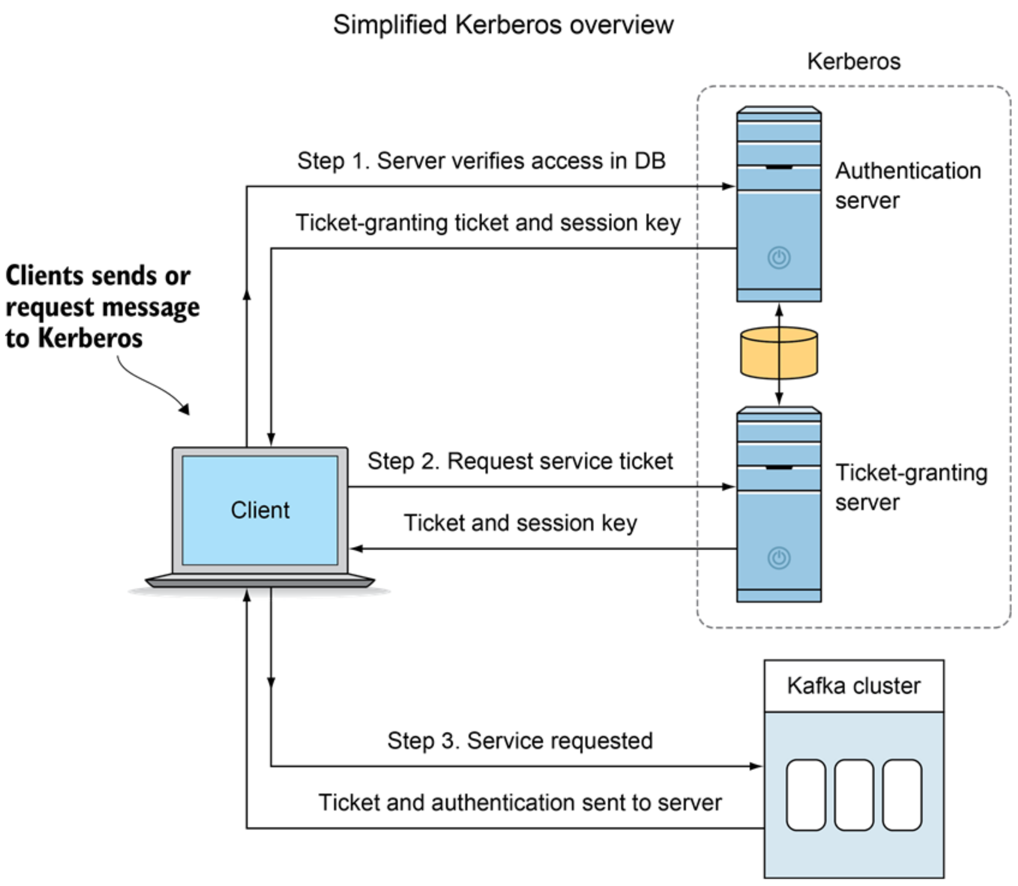
### 10.1.3 SSL Between Brokers

Since you also have your brokers talking to each other, you might want to determine if you also want to use SSL for those interactions. `security.inter.broker.protocol = SSL` should be used if you do not want to continue using plaintext for your communication between brokers.

## 10.2 Simple Authentication and Security Layer (SASL)

### 10.2.1 Kerberos

If you have a security team that already has a Kerberos server, your life likely got a little easier! When I first started working with Kafka, it was part of a suite of Big Data tools that mostly all leveraged Kerberos. Kerberos is often found in larger organizations as a method to provide single sign-on (SSO) that is secure. If you have a Kerberos server setup already, you will need to work with a user with access to that Kerberos environment to create a principal for each broker and also for each user (or application id) that will access our cluster.


Simplified Kerberos overview
Kerberos
Step 1. Server verifies access in DB
Authentication server
Ticket-granting ticket and session key
Clients sends or request message to Kerberos
Step 2. Request service ticket
Client
Ticket-granting server
Ticket and session key
Kafka cluster
Step 3. Service requested
Ticket and authentication sent to server

One common file that we will talk about for the brokers and clients are the idea of Java Authentication and Authorization Service (JAAS) files. This file will help us provide to Kafka the principal we will use as well as credentials like a keytab file. A keytab will likely be a separate file that has principles and encrypted keys. This keytab file will be used to authenticate to the Kafka brokers without requiring a password. However, it is important to note that you need to treat your keytab file with the same security and care that you would for any credential.

To get our brokers setup, let's look in detail at the server property changes needed as well as the JAAS configuration as well. To start, it is important to note that each broker will need their own keytab file. Our JAAS file will help our brokers find the keytab's location on our server as well as our principal to use.

## Listing 10.7 Broker SASL JAAS File

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab=”/var/kafka_server0.keytab”
    principal=”kafka/kafka0.kia.manning.com@EXAMPLE.COM”;
};

// Zookeeper client authentication - Kafka being the client
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab=”/var/kafka_server0.keytab”
    principal=”kafka/kafka0.kia.manning.com@EXAMPLE.COM”;
};


# Java parameter for Kafka server startup to point to the above file content
 -Djava.security.auth.login.config=/var/broker0_jaas.conf
 -Djava.security.krb5.conf=/var/krb.conf
 -Dsun.security.krb5.debug=true
```

We are going to add another port as well in order to test out SASL_SSL together before we remove the older ports.

## Listing 10.8 Broker SASL properties changes

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093,SASL_SSL://localhost:9094   ❶
sasl.kerberos.service.name=kafka   ❷
```

❶ We have added the SASL_SSL broker port and left the older ports

❷ We will use the same primary name for all brokers

The setup from a client is very similar. We have to add a JAAS file as well as a parameter on start-up.

**Listing 10.9 Client SASL JAAS File**

```
    KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/car/kafkaclient.keytab"
    principal="kafkaclient@EXAMPLE.COM";
};

# Java parameter for Kafka server startup to point to the above file content
-Djava.security.auth.login.config=/var/kafkaclient_jaas.conf
```

We also need to add the configuration for the values if not included already for SASL.

**Listing 10.10 Client SASL JAAS File**

```
security.protocol=SASL_SSL
sasl.kerberos.service.name=kafka
ssl.truststore.location=/var/kafka.client.truststore.jks
ssl.truststore.password=masquerade
ssl.key.password=masquerade
```

After testing that things are not broke on port 9092 or 9093, we can attempt to use our new configuration by validating the same result as before when we use our new SASL_SSL protocol.

### 10.2.2 HTTP Basic Auth

One common method for developers that use REST APIs or web services in general, is HTTP basic authentication. The reason why we do not discuss this option in-depth is that it is limited to the following components:

- KSQL
- Schema Registry
- Connect REST API
- REST Proxy
- Control Center REST API

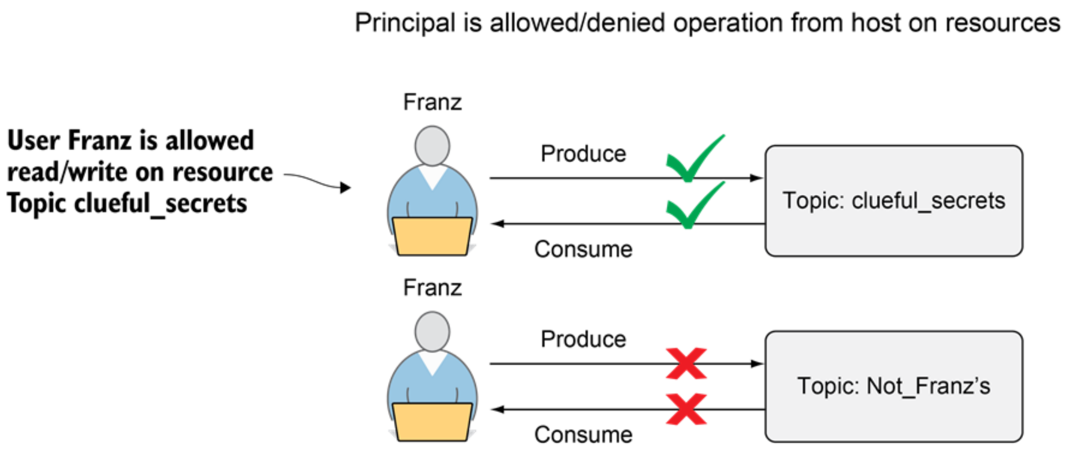
Most of these options rely on Java Authentication and Authorization Service (JAAS) configuration files and creation of a password properties file. Overall, if you have Kerberos already in-use, that is probably your most supported option at the current time of publication.

## 10.3 Authorization in Kafka

Since we have looked at how to use authentication with Kafka, let's take a look at how we can start using that information about the user in order to apply access.

### 10.3.1 Access Control Lists

As we quickly reviewed, authorization is the process that controls what a user can do. One way to do that is with Access Control Lists (ACLs). While most Linux users are familiar with permissions on a file such as read, write and execute, one drawback is that they permissions are set for an owner, one group, and all others. ACLs can provide permissions for multiple individuals and groups as well as more types of permissions. ACLs are often used when different levels of access for a shared folder is needed. One example is a permission to let a user edit a file but not allow that same user to delete it (delete is a separate permission altogether).


Principal is allowed/denied operation from host on resources
User Franz is allowed read/write on resource Topic clueful_secrets
Franz
Produce
Consume
Topic: clueful_secrets
Franz
Produce
Consume
Topic: Not_Franz's

Kafka designed their authorizer to be pluggable to allow users to make their own logic if desired. Kafka does have a class `SimpleAclAuthorizer` that we will use in our example. With this authorizer, ACLs are stored in ZooKeeper. Brokers asynchronously get this information and cache this metadata in order to make following processing quicker.

**Listing 10.11 ACL Authorizer and Super Users**

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer    ❶
super.users=User:Franz   ❷
```

❶ Every broker configuration should include the SimpleAclAuthorizer

❷ We add a super user that will have access to all resources with or without ACLs

The Kafka Authorizer CLI tool is included with the other Kafka scripts in our install. `kafka-acls.sh` allows us to add, delete, or even list current ACLs. An important item to note is that once we configure an Authorizer, ACLs will need to be set or only those considered super users will have access.

Let's figure out how to make only Team Clueful have access to produce and consume from their own topic: clueful_secrets.

For brevity, we will have 2 users in our example team, Franz and Hemingway. Since we already have the keytabs of the users created, we know the principal information that we need.

**Listing 10.12 Kafka ACLs To Read and Write to a Topic**

```
bin/kafka-acls --authorizer-properties \
  zookeeper.connect=localhost:2181 --add \
 --allow-principal User:Franz --allow-principal User:Hemingway \   ❶
 --operation Read --operation Write --topic clueful_secrets   ❷
```

❶ We are identifying our 2 users we want to grant permissions

❷ We allow the named principals to both Read and Write to the specific topic mentioned

As you may have noticed in Figure 10.X, the operation granted READ is allowing consumers the ability to get data from the topic. The operation WRITE on the other lets the same principal produce data into the topic.

While a commercial component of the Confluent Platform, the Confluent LDAP Authorizer is one option to look into if you want to leverage existing Group and/or User principal types.

## 10.3.2 Role-based access control

Role-based access control (RBAC) is a newer option that Confluent Platform supports. RBAC is a way to control access based on roles. Users are then assigned to their role according to their needs such as a job duty. Instead of granting each and every user permissions, with RBAC, you manage the privileges assigned to predefined roles.

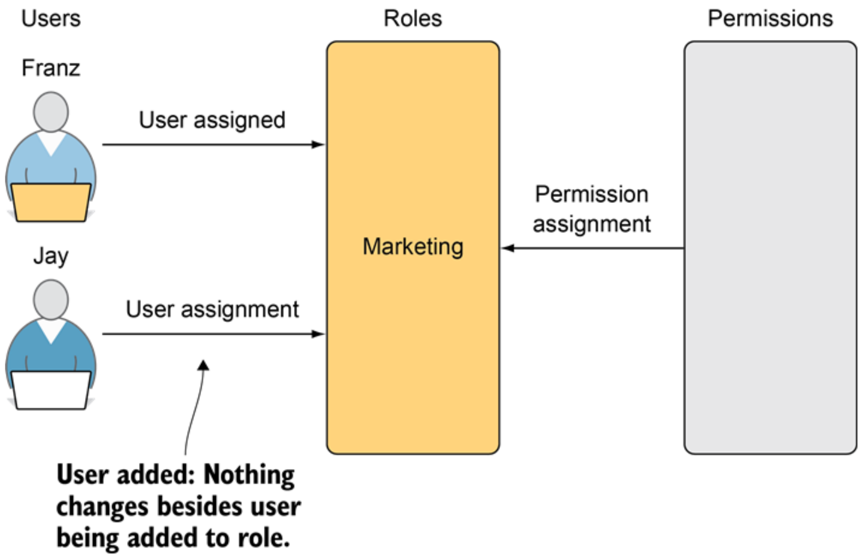


**Figure 10.1 Role-based access control**

It can be used as a layer on top of ACLs that we discussed earlier. One of the biggest reason to consider both RBAC and ACLS is if you need more granular access control in certain areas. For example, while a user might be able to access a resource like a topic due to RBAC, you could still use an ACL to deny access to specific members of that group.

For our treasure hunting teams, it might make sense to have a specific Role per team. This might mirror how a team in a Marketing area would have a role vs a team from Accounting. If some user moved departments, they would be reassigned their Role and not individual permissions. Since this is a newer option that might change as it matures and is limited to a Confluent Platform environment, the option is mentioned for awareness even though we will not dig further in this material.

## 10.4 ZooKeeper

Part of securing Kafka is also the need to look at how you secure all parts of your cluster: including ZooKeeper. Not only can someone really miss up your cluster by modifying broker data stored in ZooKeeper, the other concern is also the ACLs themselves. If you protect the brokers but not the system that holds that security-related data, it would be possible for those with knowledge to update those accesses without much effort. While the data in ZooKeeper is not sensitive and znodes can be read by anyone, we are trying to stop changing the znodes. We hope to stop cluster changes that might hurt the health of the cluster.

One first step is to setup the Client section of the JAAS login file we used for the brokers.

We also need to set the value `zookeeper.set.acl` in each Kafka broker to true.

**Listing 10.13 ACL ZooKeeper**

```
zookeeper.set.acl=true    ❶
```

❶ Every broker configuration should include this ZooKeeper related value

### 10.4.1 Kerberos Setup

Making sure that ZooKeeper works with Kerberos requires a couple of configuration changes.

In the zookeeper.properties configuration we would place to add values that let ZooKeeper know that SASL should be used for clients and which provider should be used.

**Listing 10.14 ZooKeeper properties updates for Kerberos**

```
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
requireClientAuthScheme=sasl
jaasLoginRenew=3600000
```

On start-up of the ZooKeeper server, we will have to add a parameter to point to a JAAS file that helps us define our keytab file location on the server as well as our principal.

**Listing 10.15 ZooKeeper JAAS File Example**

```
Server {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab=”/var/kafka.keytab”   ❶
storeKey=true
useTicketCache=false
principal=”kafka/kia.manning.com”;   ❷
};

# Java parameter for ZooKeeper server startup
-Djava.security.auth.login.config=/var/zk_jaas.conf   ❸
```

❶ The JAAS file will need the location of the keytab present

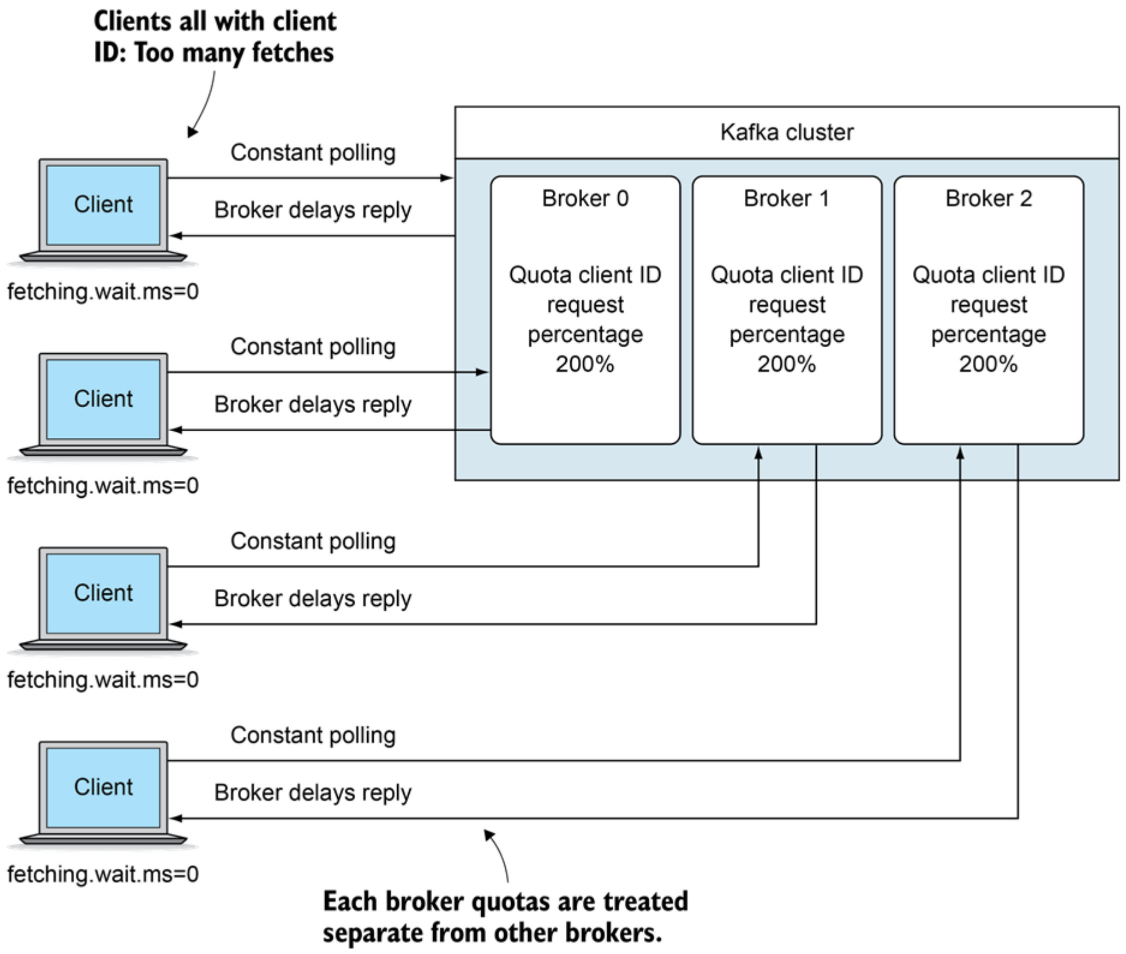
❷ The principal used will also been to be declared

❸ The ZooKeeper server will need to be restarted to set the location of the new jaas conf file

## 10.5 Quotas

Let’s say that users of our web application start to notice that they don’t have any issues with trying over and over to request data. While this is often a good thing for end-users who want to be able to use a service as much as they want without their progress being limited, the cluster might need some protection from users who might use that to their advantage. In our instance, since we made it so the data was locked down to members of our team only, some users have thought of a way to try to avoid others from using the system. In effect, they are trying to use a distributed denial-of-service (DDoS) attack against our system. If you are not familiar with what a DDoS is, it is a malicious attempt to disrupt normal traffic of a service and network. A targeted attack against our cluster can overwhelm our brokers and their surrounding infrastructure. In practice, the other teams are requesting reads from their topics over and over while reading from the beginning. One detail to know is that quotas is defined on a per-broker basis. The cluster does not look across each broker to calculate a total, so a per-broker definition is needed.

Our quota metadata is written to ZooKeeper which allows the brokers to get this updated information quickly. This update path is also a benefit in that we do not have to have a restart of our brokers in order for this new configuration to take effect. The ability to react quickly to our quotas changes is a plus if we are addressing clients that are suddenly a problem in our ecosystem. Also, by default, quotas are unlimited. While there is a way to set producer and consumer defaults with the keys: `quota.producer.default` and `quota.consumer.default` this technique is noted in the project documentation as being deprecated.



To set our own custom quotas, we need to know a couple of items including: how to identify the "who" to limit and then the limit itself. Whether you have security impacts what options we have for defining the who we are limiting. Without security, you are able to use the `client.id` property. With security enabled, we can also add the `user`, and thus, any `user` and `client.id` combinations as well.

There are a couple of types of quotas that we can look at defining for our clients: network bandwidth and request rate quotas. Let's take a look at the network bandwidth option first.

## 10.5.1 Network Bandwidth Quota

Network bandwidth is measured for the number of bytes per second. In our example, we want to make sure that each client is respecting the network and not flooding it to prevent others from using it as well. Each team in our competition uses the same client id specific to their team for any producer or consumer requests from their clients. In our example, we are going to limit the clients using the client id `clueful` by setting a `producer_byte_rate` and the `consumer_byte_rate`.

**Listing 10.16 Creating a Network Bandwidth Quota for Client Clueful**

```
kafka-configs.sh  --zookeeper localhost:2181 --alter \
--add-config 'producer_byte_rate=1048576,consumer_byte_rate=10485760' \   ❶
--entity-type clients --entity-name clueful   ❷
```

❶ We are allowing producers 1 MB/sec and Consumers 10 MB/sec

❷ The entity-name for a client type is our client.id clueful

As is often the case, we might need to list our current quotas as well as delete them if no longer needed. All of these commands can be completed with sending different arguments to the `kafka-configs.sh` script.

**Listing 10.17 Listing and Deleting a Quota for Client Clueful**

```
kafka-configs.sh  --zookeeper localhost:2181 \
---describe \   ❶
--entity-type clients --entity-name clueful

kafka-configs.sh  --zookeeper localhost:2181 --alter \
--delete-config 'producer_byte_rate,consumer_byte_rate' \   ❷
--entity-type clients --entity-name client1
```

❶ We are listing the existing configuration of our client.id

❷ We are using the delete-config option to remove those we just added

As you start to add quotas and might have more than one quotas apply to a client, you will want to make sure of the precedence of how various quotas are applied. While it might seem like the most restrictive (lowest bytes allowed) of the quotas would be applied, it is not always the case. The following is the order in which quotas are applied with the most highest precedence listed at the top:

- User and client.id provided quotas
- User quotas
- Client.id quotas

For example, if a user named Franz had a user quota limit of 10 MB and a client.id limit of 1 MB, the consumer he ran would be allowed 10 MB per second due to the User-defined quota having higher precedence.

### 10.5.2 Request Rate Quotas

The other quota to examine is for request rate. Why the need for a second quota? While a DDoS attack is offer thought of as a network issue, clients making lots of connections could still overwhelm the broker by making CPU-intensive requests (anything related to SSL or message compressed/decompression are good examples). Consumer clients that poll continuously with a setting of `fetch.max.wait.ms=0` also a concern that can be addressed with this quota.

The simplest way to think about this quota is that it represents the total percentage of CPU that a group of related clients is allowed to use. To set this quota, we use the same entity types and add-config options as we did with our other quotas. The biggest difference is setting the configuration for `request_percentage`. For a formula that you can use is the number of I/O threads (num.io.threads) + the number of network threads (num.network.threads) * 100%. For our example, we are setting a request percentage of 100. This is setting each thread quota at .5.

**Listing 10.18 Creating a Network Bandwidth Quota for Client Clueful**

```
kafka-configs.sh  --zookeeper localhost:2181 --alter \
--add-config 'request_percentage=100' \   ❶
--entity-type clients --entity-name clueful   ❷
```

❶ We are allowing producers 100 request percentage. (.5 + .5) * 100 = 100

❷ The entity-name for a client type is our client.id clueful

Using quotas are a good way to protect your cluster as well as react to clients that suddenly might start putting a strain on your brokers.

## 10.6 Data at Rest

One question is whether you need to encrypt the data that Kafka writes to disk. By default, Kafka does not encrypt the events it adds to the log. There have been a couple of Kafka Improvement Proposals that have looked at this feature, but at the time of publication you will still need to make sure your own strategy for your data has that need accounted for. One of the first attempts at this feature could be volume encryption. However, for those that have access to the broker file system, like OS administrators, they might then have access to all of the topic data. Depending on your business needs as well, you might want to actually only encrypt specific topics or even specific topics with unique keys.

### 10.6.1 Managed Options

Of course, if you are using a managed option for your cluster, it might be good to check out what features the service provides. Amazon's Managed Streaming for Apache Kafka (MSK) is one example of a cloud provider that handles a large part of your cluster management including some security pieces as well. Having your brokers and ZooKeeper nodes updated with automatically deployed hardware patches and related updates addresses one major source of keeping issues at bay. The other benefit with these updates is that you are not providing access to even more administrators to your cluster. Amazon MSK also provides encryption for your data at rest using either the AWS Key Management Service (KMS) Customer Master Key (CMK) or your own CMK. Additional features that we covered in our examples are the ability to use SSL between your clients and cluster and ACLs.

Confluent Cloud also is an option that can be deployed across various public cloud offerings. Support for Data encryption at rest & in-motion as well as ACL support matches options that you should be aware of when matching your security requirements to the actual provider.

Also sticking with the Confluent stack, Confluent Platform 5.3 has a commercial feature called Secret Protection. When we looked at our SSl configuration files earlier, we had clear text passwords in certain files. However, Secret Protection is meant to address that issue by encrypting the secrets in the file and keeping those exposed values out of log files as well. Since this is a commercial offering, I will not go into depth on how it works, but just be aware there are options to dig into.

## 10.7 Summary

In this chapter we learned:

- How to set up SSL between our cluster and our clients
- Kerberos use for security
- The use of quotas to limit client impact
- To utilize Access Control Lists (ACLs) for our resources

# A 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.

## A.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.

## A.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.

## A.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.

## A.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 A.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 A.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 A.3 Creating Multiple Kafka Broker Configs**

```
cp config/server.properties config/server0.properties   ❶
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   ❷
broker.id=0
listeners=PLAINTEXT://localhost:9092
log.dir=/tmp/kafka-logs-0

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

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

❶ Make three copies of the default server properties files.

❷ Update id, port, and log directory for broker id 0

❸ Update id, port, and log directory for broker id 1

❹ Update id, port, and log directory for broker id 2

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

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

```
bin/kafka-server-start.sh config/server0.properties   ❶
bin/kafka-server-start.sh config/server1.properties
bin/kafka-server-start.sh config/server2.properties
```

❶ 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 A.5 jps output for Zookeeper (QuorumPeerMain) and 3 Brokers**

```
2532 Kafka              ❶
2745 Kafka              ❷
2318 Kafka              ❶
2085 QuorumPeerMain     ❷
```

❶ The three Kafka jvm process label and id for each broker

❷ ZooKeeper jvm process label and id

## A.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.

## Notes

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

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