1 Introduction to Kafka

This chapter covers

- Why you might want to use Kafka
- Common myths of big data and message systems
- Real-world use cases to help power messaging, streaming, and IoT data processing

As many developers are facing a world full of data 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 is Apache Kafka®.[1] Kafka is changing the standards for data platforms. It is leading the way to move from extract, transform, load (ETL) and batch workflows (in which work was often held and processed in bulk at one predefined time) to near-real-time data feeds [1]. Batch processing, which was once the standard workhorse of enterprise data processing, might not be something to turn back to after seeing the powerful feature set that Kafka provides. In

[1] Apache, Apache Kafka, and Kafka are trademarks of the Apache Software Foundation.

fact, you might not be able to handle the growing snowball of data rolling toward enterprises of all sizes unless something new is approached.

With so much data, systems can get easily overloaded. 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 or evolving data, processing this information as it happens is a way to stay up to date and current on the system's state.

Kafka touches many of the newest and the most practical trends in today's IT fields and makes its easier for daily work. For example, Kafka has already made its way into microservice designs and the Internet of Things (IoT). As a de facto technology for more and more companies, Kafka is not only for super geeks or alpha-chasers. Let's start by looking at Kafka's features, introducing Kafka itself, and understanding more about the face of modern-day streaming platforms.

1.1 What is Kafka?

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

- Reading and writing records like a message queue
- Storing records with fault tolerance
- Processing streams as they occur [2]

Readers who are not as familiar with queues or message brokers in their daily work might need help when discussing the general purpose and flow of such a system. As a generalization, a core piece of Kafka can be thought of as providing the IT equivalent of a receiver that sits in a home entertainment system. Figure 1.1 shows the data flow between receivers and end users.

As figure 1.1 shows, digital satellite, cable, and Blu-ray™ players can connect to a central receiver. You can think of those individual pieces as regularly sending data in a format that they know about. That flow of data can be thought of as nearly constant while a movie or CD is playing. The receiver deals with this constant stream of data and converts it into a usable format for the external devices attached to the other end (the receiver sends the video to your television and the audio to a decoder as well as to the speakers). So what does this have to do with Kafka exactly? Let's look at the same relationship from Kafka's perspective in figure 1.2.

Kafka includes clients to interface with other systems. One such client type is called a *producer*, which sends multiple data streams to the Kafka brokers. The brokers serve a similar function as the receiver in figure 1.1. Kafka also includes *consumers*, clients that can read data from the brokers and process it. Data does not have to be limited to only a single destination. The producers and consumers are completely decoupled, allowing each client to work independently. We'll dig into the details of how this is done in later chapters.

As do other messaging platforms, Kafka acts (in reductionist terms) like a middleman for data coming into the system (from producers) and out of the system (for consumers or end users). The loose coupling can be achieved by allowing this separation

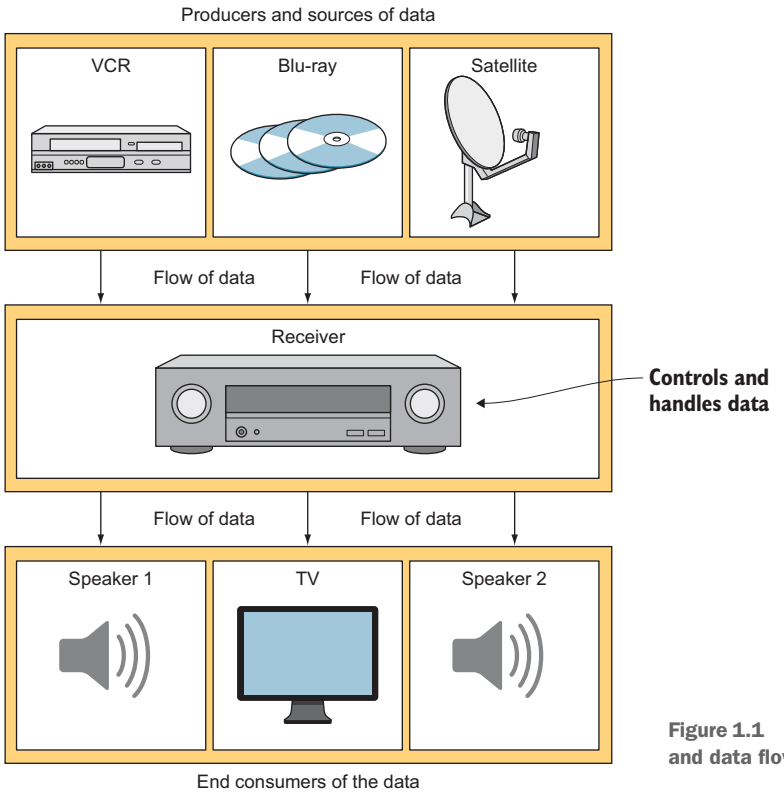

Figure 1.1 Producers, receivers, and data flow overview

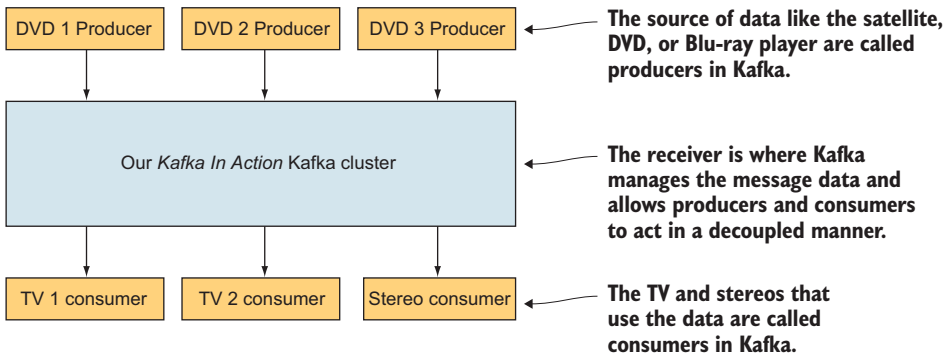

Figure 1.2 Kafka's flow from producers to consumers

between the producer and the end user of the message. The producer can send whatever message it wants and still have no clue about if anyone is subscribed. Further, Kafka has various ways that it can deliver messages to fit your business case. Kafka's message delivery can take at least the following three delivery methods [3]:

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

Let's dig into what those messaging options mean. Let's look at *at-least-once* semantics (figure 1.3). In this case, Kafka can be configured to allow a producer of messages to send the same message more than once and have it written to the brokers. If a message does not receive a guarantee that it was written to the broker, the producer can resend the message [3]. 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 it is one of the safest delivery methods.

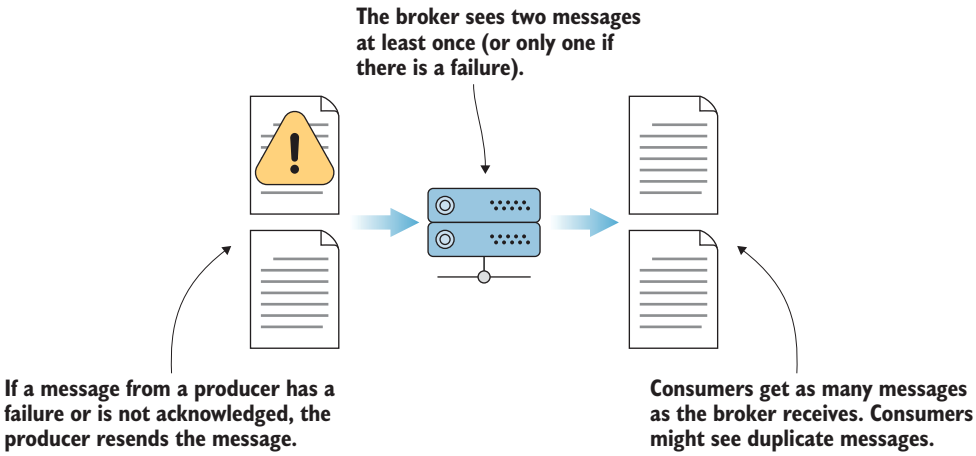

Figure 1.3 At-least-once message flow

At-most-once semantics (figure 1.4) is when a producer of messages might send a message once and never retry. In the event of a failure, the producer moves on and doesn't attempt to send it again [3]. Why would someone be okay with losing a message? If a popular website is tracking page views for visitors, it might be okay with missing a few page view events out of the millions it processes each day. Keeping the system performing and not waiting on acknowledgments might outweigh any cost of lost data.

Kafka added the *exactly-once* semantics, also known as EOS, to its feature set in version 0.11.0. EOS generated a lot of mixed discussion with its release [3]. On the one hand, exactly-once semantics (figure 1.5) are ideal for a lot of use cases. This seemed like a logical guarantee for removing duplicate messages, making them a thing of the past. But most developers appreciate sending one message and receiving that same message on the consuming side as well.

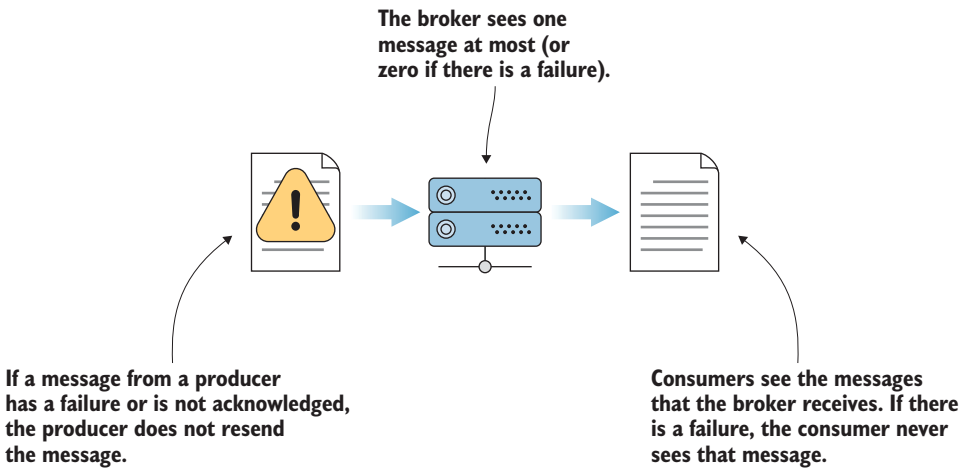

Figure 1.4 At-most-once message flow

Another discussion that followed the release of EOS was a debate on if exactly once was even possible. Although this goes into deeper computer science theory, it is helpful to be aware of how Kafka defines their EOS feature [4]. If a producer sends a message more than once, it will still be delivered only once to the end consumer. EOS has touchpoints at all Kafka layers—producers, topics, brokers, and consumers—and will be briefly tackled later in this book as we move along in our discussion for now.

Besides various delivery options, another common message broker benefit is that if the consuming application is down due to errors or maintenance, the producer does

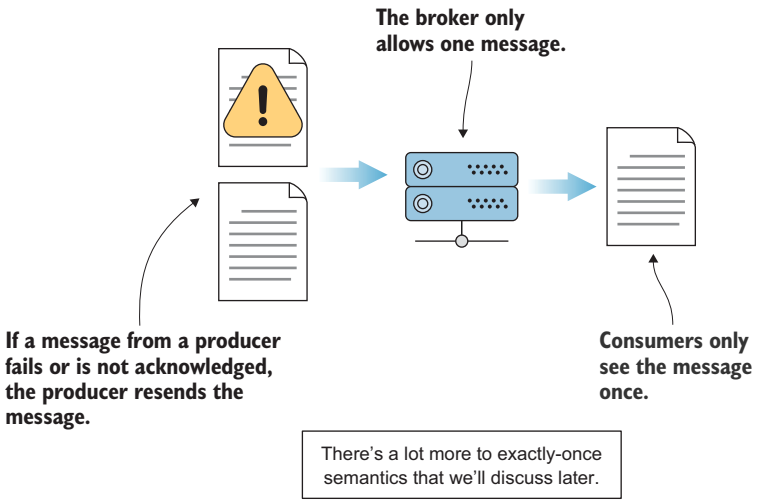

Figure 1.5 Exactly-once message flow

not need to wait on the consumer to handle the message. When consumers start to come back online and process data, they should be able to pick up where they left off and not drop any messages.

1.2 Kafka usage

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

With Kafka, enormous data processing needs are handled with Kafka in production [5]. All this with a tool that was not at its 1.0 version release until 2017! However, besides these headline-grabbing facts, why would users want to start looking at Kafka? Let's look at that answer next.

1.2.1 Kafka for the developer

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

One of the various on-ramps for newer developers to Kafka is to apply things they know to help them with the unknown. For example, Java® developers can use Spring® concepts, and Dependency Injection (DI) Spring for Kafka (https://projects.spring.io/spring-kafka) has already been through a couple of major release versions. Supporting projects, as well as Kafka itself, have a growing tool ecosystem all their own.

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

Take, for example, an HR system that employees would use to submit paid vacation leaves. If you are used to a create, read, update, and delete (CRUD) system, the submission of time off would likely be processed by not only payroll but also project burndown charts for forecasting work. Do you tie the two applications together? What if the payroll system goes down? Should that impact the availability of the forecasting tooling?

With Kafka, we will see the benefits of being able to decouple some of the applications that we have tied together in older designs. (We will look more in-depth at maturing our data model in chapter 11.) Kafka, however, can be put into the middle of the workflow [7]. Your interface to data becomes Kafka instead of numerous APIs and databases.

Some say that there are better and simpler solutions. What about using ETL to at least load the data into databases for each application? That would only be one interface per application and easy, right? But what if the initial source of data is corrupted

or outdated? 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 again 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 "dogfoods" itself. For example, when we dig into consumers in chapter 5, we will see how Kafka uses topics internally to manage consumers' offsets. After the release of v0.11, exactly-once semantics for Kafka also uses internal topics. The ability to have many data consumers using the same message allows many possible outcomes.

Another developer question might be, why not learn Kafka Streams, ksqlDB, Apache Spark™ Streaming, or other platforms and skip learning about core Kafka? The number of applications that use Kafka internally is indeed impressive. Although abstraction layers are often nice to have (and sometimes close to being required with so many moving parts), we believe that Kafka itself is worth learning.

There is a difference in knowing that Kafka is a channel option for Apache Flume™ and understanding what all of the config options mean. Although Kafka Streams can simplify examples you might see in this book, it is interesting to note how successful Kafka was before Kafka Streams was even introduced. Kafka's base is fundamental and will, hopefully, help you see why it is used in some applications and what happens internally. If you want to become an expert in streaming, it is important to know the underlying distributed parts of your applications and all the knobs you can turn to fine-tune your applications. From a purely technical viewpoint, there are exciting computer science topics applied in practical ways. Perhaps the most talked about is the notion of distributed commit logs, which we will discuss in depth in chapter 2, and a personal favorite, hierarchical timing wheels [8]. These examples show you how Kafka handles an issue of scale by applying an interesting data structure to solve a practical problem.

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

1.2.2 Explaining Kafka to your manager

As is often the case, sometimes members of the C-suite will hear the word *Kafka* and be more confused by the name than care about what it does. It might be nice to explain the value found in this product. Also, it is good to step back and look at the larger picture of what the real added value is for this tool.

One of Kafka's most important features is the ability to take volumes of data and make it available for use by various business units. Such a data backbone that makes information coming into the enterprise available to all the multiple business areas allows for flexibility and openness on a company-wide scale. Nothing is prescribed, but increased access to data is a potential outcome. Most executives also know that

with more data than ever flooding in, the company wants insights as fast as possible. Rather than pay for data to get old on disk, its value can be derived as it arrives. Kafka is one way to move away from a daily batch job that limits how quickly that data can be turned into value. *Fast data* seems to be the newer term, hinting that real value focuses on something different from the promises of big data alone.

Running on a Java virtual machine JVM® should be a familiar and comfortable place for many enterprise development shops. The ability to run on premises is a crucial driver for some whose data requires on-site oversight. And the cloud and managed platforms are options as well. Kafka can scale horizontally, and not depend on vertical scaling alone, which might eventually reach an expensive peak.

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

1.3 Kafka myths

When you start to learn any new technology, it is often natural to try to map existing knowledge to new concepts. Although that technique can be used in learning Kafka, we wanted to note some of the most common misconceptions that we have run into in our work so far. We'll cover those in the next sections.

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 for users as a tool as part of a Cloudera™ or Hortonworks™ suite. It isn't uncommon to hear the myth that Kafka only works with Hadoop. What could cause this confusion? One of the causes is likely the various tools that use Kafka as part of their products. Spark Streaming and Flume are examples of tools that use Kafka (or did at one point) and could be used with Hadoop as well. The dependency (depending on the version of Kafka) on Apache ZooKeeper™ is also a tool that is often found in Hadoop clusters and might tie Kafka further to this myth.

One other fundamental myth that often appears is that Kafka requires the Hadoop Distributed Filesystem (HDFS). That is not the case. Once we start to dig into how Kafka works, we will see that Kafka's speed and techniques used to process events would likely be much slower with a NodeManager in the middle of the process. Also, the block replication, usually a part of HDFS, is not done in the same way. One such example is that in Kafka, replicas are not recovered by default. Whereas both products use replication in different ways, the durability that is marketed for Kafka might be easy to group under the Hadoop theme of expecting failure as a default (and thus planning for overcoming it) and is a similar overall goal between Hadoop and Kafka.

1.3.2 Kafka is the same as other message brokers

Another big myth is that Kafka is just another message broker. Direct comparisons of the features of various tools (such as Pivotal's RabbitMQ™ or IBM's MQSeries®) to Kafka often have asterisks (or fine print) attached and are not always fair to the best use cases of each. Some tools over time have gained or will gain new features just as Kafka has added the exactly-once semantics. And default configurations can be changed to mirror features closer to other tools in the same space. In general, the following lists some of the most exciting and standout features that we will dig into in a bit:

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

Kafka was designed to have multiple consumers. What that means is that one application reading a message from the message broker doesn't remove it from other applications that might want to consume it as well. One effect of this is that a consumer who has already seen the message can again choose to read it (and other messages as well). With some architecture models like lambda (discussed in chapter 8), programmer mistakes are expected just as much as hardware failures. Imagine consuming millions of messages, and you forget to use a specific field from the original message. In some queues, that message is removed or sent to a duplicate or replay location. However, Kafka provides a way for consumers to seek specific points and read messages again (with a few constraints) by just seeking an earlier position on the topic.

As touched on briefly, Kafka allows for parallel processing of data and can have multiple consumers on the same topic. Kafka also has the concept of consumers being part of a consumer group, which will be covered in depth in chapter 5. Membership in a group determines which consumers get which messages and what work has been done across that group of consumers. Consumer groups act independently of any other group and allow for multiple applications to consume messages at their own pace with as many consumers as they require working together. Processing can happen in various ways: consumption by many consumers working on one application or consumption by many applications. No matter what other message brokers support, let's now focus on the robust use cases that have made Kafka one of the options developers turn to for getting work done.

1.4 Kafka in the real world

Applying Kafka to practical use is the core aim of this book. One of the things to note about Kafka is that it's hard to say it does one specific function well; it excels in many specific uses. Although 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 in real-world use cases. The Apache Kafka site lists general areas where Kafka is used in the real world that we explore in the book. [9].

1.4.1 Early examples

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

For an end user developing with a Kafka client, most of the details are in the configuration, and the logic becomes relatively straightforward (for example, "I want to place a message on this topic"). Having a durable channel for sending messages is also why Kafka is used.

Oftentimes, memory storage of data in RAM will not be enough to protect your data; if that server dies, the messages are not persisted across a reboot. High availability and persistent storage are built into Kafka from the start. Apache Flume provides a Kafka channel option because the replication and availability allow Flume events to be made immediately available to other sinks if a Flume agent (or the server it is running on) crashes [10]. Kafka enables robust applications to be built and helps handle the expected failures that distributed applications are bound to run into at some point.

Log aggregation (figure 1.6) is useful in many situations, including when trying to gather application events that were written in distributed applications. In the figure,

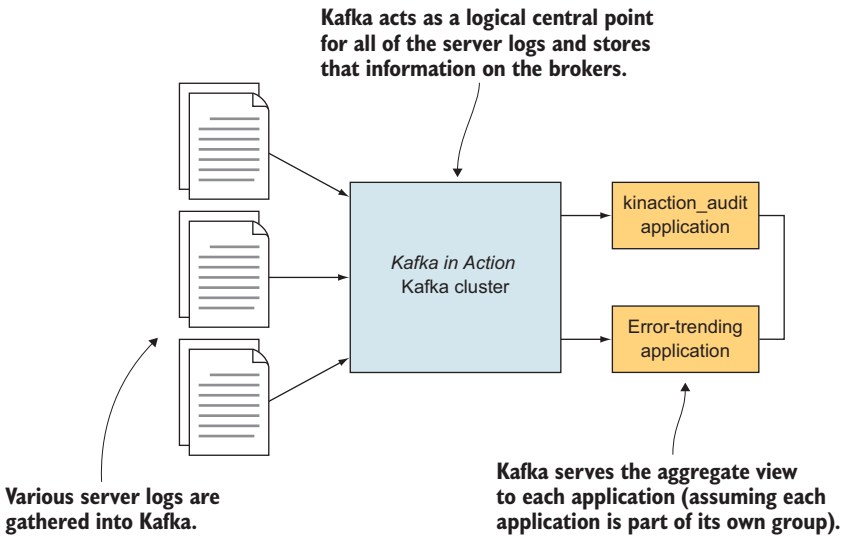

Figure 1.6 Kafka log aggregation

the log files are sent as messages into Kafka, and then different applications have a single logical topic to consume that information. With Kafka's ability to handle large amounts of data, collecting events from various servers or sources is a key feature. Depending on the contents of the log event itself, some organizations use it for auditing and failure-detection trending. Kafka is also used in various logging tools (or as an input option).

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

1.4.2 Later examples

Microservices used to talk to each other with APIs like REST, but they can now leverage Kafka to communicate between asynchronous services with events [11]. Microservices can use Kafka as the interface for their interactions rather than specific API calls. Kafka has placed itself as a fundamental piece for allowing developers to get data quickly. Although 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 provides a higher-level view of working with your data as an unbounded stream.

In the Kafka 0.11 release, exactly-once semantics was introduced. We will cover what that means in practice later, once we get a more solid foundation. However, users running end-to-end workloads on top of Kafka with the Streams API may benefit from hardened delivery guarantees. Streams make this use case easier than it has ever been to complete a flow without the overhead of any custom application logic, ensuring that a message was only processed once from the beginning to the end of the transaction.

The number of devices for the Internet of Things (figure 1.7) will only increase with time. With all of those devices sending messages, sometimes in bursts when they get a Wi-Fi or cellular connection, something needs to be able to handle that data effectively. As you may have gathered, massive quantities of data are one of the critical areas where Kafka shines. As we discussed previously, small messages are not a problem for Kafka. Beacons, cars, phones, homes, etc.—all will be sending data, and something needs to handle the fire hose of data and make it available for action [12].

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

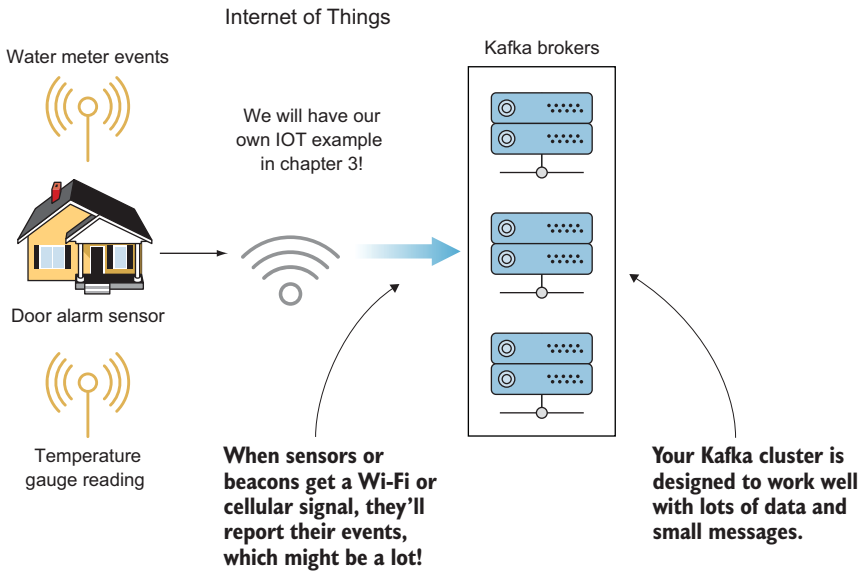

Figure 1.7 The Internet of Things (IoT)

1.4.3 When Kafka might not be the right fit

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

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

If your main access pattern for data is a mostly random lookup of data, Kafka might not be your best option. Linear read and writes are 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 from which indexes are built.

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

One of the other practical items that come to mind is that large messages are an exciting challenge. The default message size is about 1 MB [13]. With larger messages, you start to see memory pressure increase. In other words, the lower number of messages you can store in page cache could become a concern. If you are planning on sending huge archives around, you might want to see if there is a better way to manage those messages. Keep in mind that although you can probably achieve your end goal with Kafka in the previous situations (it's always possible), it might not be the first choice to reach for in your toolbox.

1.5 Online resources to get started

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

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

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

Summary

- Apache Kafka is a streaming platform that you can leverage to process large numbers of events quickly.
- Although Kafka can be used as a message bus, using it only as that ignores the capabilities that provide real-time data processing.
- Kafka may have been associated with other big data solutions in the past, but Kafka stands on its own to provide a scalable and durable system. Because it uses the same fault tolerant and distributed system techniques, Kafka fills the needs of a modern data infrastructure's core with its own clustering capabilities.
- In instances of streaming a large number of events like IoT data, Kafka handles data fast. As more information is available for your applications, Kafka provides results quickly for your data that was once processed offline in batch mode.

References

1 R. Moffatt. "The Changing Face of ETL." Confluent blog (September 17, 2018). https://www.confluent.io/blog/changing-face-etl/ (accessed May 10, 2019).

2 "Introduction." Apache Software Foundation (n.d.). https://kafka.apache.org/intro (accessed May 30, 2019).

3 Documentation. Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#semantics (accessed May 30, 2020).

4 N. Narkhede. "Exactly-once Semantics Are Possible: Here's How Apache Kafka Does It." Confluent blog (June 30, 2017). https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it (accessed December 27, 2017).

5 N. Narkhede. "Apache Kafka Hits 1.1 Trillion Messages Per Day – Joins the 4 Comma Club." Confluent blog (September 1, 2015). https://www.confluent.io/blog/apache-kafka-hits-1-1-trillion-messages-per-day-joins-the-4-comma-club/ (accessed October 20, 2019).

6 L. Dauber. "The 2017 Apache Kafka Survey: Streaming Data on the Rise." Confluent blog (May 4, 2017). https://www.confluent.io/blog/2017-apache-kafka-survey-streaming-data-on-the-rise/ (accessed December 23, 2017).

7 K. Waehner. "How to Build and Deploy Scalable Machine Learning in Production with Apache Kafka." Confluent blog (September 29, 2017) https://www.confluent.io/blog/build-deploy-scalable-machine-learning-production-apache-kafka/ (accessed December 11, 2018).

8 Y. Matsuda. "Apache Kafka, Purgatory, and Hierarchical Timing Wheels." Confluent blog (October 28, 2015). https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels (accessed December 20, 2018).

9 "Use cases." Apache Software Foundation (n.d.). https://kafka.apache.org/uses (accessed May 30, 2017).

10 "Flume 1.9.0 User Guide." Apache Software Foundation (n.d.). https://flume.apache.org/FlumeUserGuide.html (accessed May 27, 2017).

11 B. Stopford. "Building a Microservices Ecosystem with Kafka Streams and KSQL." Confluent blog (November 9, 2017). https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/ (accessed May 1, 2020).

12 "Real-Time IoT Data Solution with Confluent." Confluent documentation. (n.d.). https://www.confluent.io/use-case/internet-of-things-iot/ (accessed May 1, 2020).

13 Documentation. Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#brokerconfigs_message.max.bytes (accessed May 30, 2020).