O'REILLY®

Second Edition

Kafka

The Definitive Guide

Real-Time Data and Stream Processing at Scale

Compliments of

CONFLUENT

Gwen Shapira, Todd Palino,
Rajini Sivaram & Krit Petty

O'REILLY®

Kafka: The Definitive Guide

Every enterprise application creates data, whether it consists of log messages, metrics, user activity, or outgoing messages. Moving all this data is just as important as the data itself. With this updated edition, application architects, developers, and production engineers new to the Kafka streaming platform will learn how to handle data in motion. Additional chapters cover Kafka's AdminClient API, transactions, new security features, and tooling changes.

Engineers from Confluent and LinkedIn responsible for developing Kafka explain how to deploy production Kafka clusters, write reliable event-driven microservices, and build scalable stream processing applications with this platform. Through detailed examples, you'll learn Kafka's design principles, reliability guarantees, key APIs, and architecture details, including the replication protocol, the controller, and the storage layer.

You'll examine:

- Best practices for deploying and configuring Kafka
- Kafka producers and consumers for writing and reading messages
- Patterns and use-case requirements to ensure reliable data delivery
- Best practices for building data pipelines and applications with Kafka
- How to perform monitoring, tuning, and maintenance tasks with Kafka in production
- The most critical metrics among Kafka's operational measurements
- Kafka's delivery capabilities for stream processing systems

"A must-have for developers and operators alike. You need this book if you're using or running Kafka."

—**Chris Riccomini**
Software Engineer, Startup Advisor, and Coauthor of *The Missing README*

Gwen Shapira is an engineering leader at Confluent and manages the cloud native Kafka team, which is responsible for Kafka performance, elasticity, and multitenancy.

Todd Palino, principal staff engineer in site reliability at LinkedIn, is responsible for capacity and efficiency planning.

Rajini Sivaram is a principal engineer at Confluent, designing and developing cross-cluster replication and security features for Kafka.

Krit Petty is the site reliability engineering manager for Kafka at LinkedIn.

DATA | DATABASES

Praise for *Kafka: The Definitive Guide*

Kafka: The Definitive Guide has everything you need to know to get the most from Kafka, whether in the cloud or on-prem. A must-have for developers and operators alike. Gwen, Todd, Rajini, and Krit jam years of wisdom into one concise book. You need this book if you're using or running Kafka.

—Chris Riccomini, software engineer, startup advisor, and coauthor of The Missing README

A comprehensive guide to the fundamentals of Kafka and how to operationalize it.

—Sumant Tambe, senior software engineer at Linkedin

This book is an essential read for any Kafka developer or administrator. Read it cover to cover to immerse yourself in its details, or keep it on hand for quick reference. Either way, its clarity of writing and technical accuracy is superb.

—Robin Moffatt, staff developer advocate at Confluent

This is foundational literature for all engineers interested in Kafka. It was critical in helping Robinhood navigate the scaling, upgrading, and tuning of Kafka to support our rapid user growth.

—Jaren M. Glover, early engineer at Robinhood, angel investor

A must-read for everyone who works with Apache Kafka: developer or admin, beginner or expert, user or contributor.

—Matthias J. Sax, software engineer at Confluent and Apache Kafka PMC member

Great guidance for any team seriously using Apache Kafka in production, and engineers working on distributed systems in general. This book goes far beyond the usual introductory-level coverage and into how Kafka actually works, how it should be used, and where the pitfalls lie. For every great Kafka feature, the authors clearly list the caveats you'd only hear about from grizzled Kafka veterans. This information is not easily available in one place anywhere else. The clarity and depth of explanations is such that I would even recommend it to engineers who do not use Kafka: learning about the principles, design choices, and operational gotchas will help them make better decisions when creating other systems.

—Dmitriy Ryaboy, VP of software engineering at Zymergen

SECOND EDITION

Kafka: The Definitive Guide

Real-Time Data and Stream Processing at Scale

Gwen Shapira, Todd Palino,
Rajini Sivaram, and Krit Petty

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kafka: The Definitive Guide
by Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Gary O'Brien
Production Editor: Kate Galloway
Copyeditor: Sonia Saruba
Proofreader: Piper Editorial Consulting, LLC

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2017: First Edition
November 2021: Second Edition

Revision History for the Second Edition
2021-11-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492043089* for release details.

978-1-492-08736-6

[LSI]

Table of Contents

Foreword to the Second Edition

The first edition of *Kafka: The Definitive Guide* was published five years ago. At the time, we estimated that Apache Kafka was used in 30% of Fortune 500 companies. Today, over 70% of Fortune 500 companies are using Apache Kafka. It is still one of the most popular open source projects in the world and is at the center of a huge ecosystem.

Why all the excitement? I think it is because there has been a huge gap in our infrastructure for data. Traditionally, data management was all about storage—the file stores and databases that keep our data safe and let us look up the right bit at the right time. Huge amounts of intellectual energy and commercial investment have been poured into these systems. But a modern company isn't just one piece of software with one database. A modern company is an incredibly complex system built out of hundreds or even thousands of custom applications, microservices, databases, SaaS layers, and analytics platforms. And increasingly, the problem we face is how to connect all this up into one company and make it all work together in real time.

This problem isn't about managing data at rest—it is about managing data in motion. And right at the heart of that movement is Apache Kafka, which has become the de facto foundation to any platform for data in motion.

Through this journey, Kafka hasn't remained static. What started as a bare-bones commit log has evolved as well: adding connectors and stream processing capabilities, and reinventing its own architecture along the way. The community not only evolved existing APIs, configuration options, metrics, and tools to improve Kafka's usability and reliability, but we've also introduced a new programmatic administration API, the next generation of global replication and DR with MirrorMaker 2.0, a new Raft-based consensus protocol that allows for running Kafka in a single executable, and true elasticity with tiered storage support. Perhaps most importantly, we've made Kafka a no-brainer in critical enterprise use cases by adding support for advanced security options—authentication, authorization, and encryption.

As Kafka evolves, we see the use cases evolve as well. When the first edition was published, most Kafka installations were still in traditional on-prem data centers using traditional deployment scripts. The most popular use cases were ETL and messaging; stream processing use cases were still taking their first steps. Five years later, most Kafka installations are in the cloud, and many are running on Kubernetes. ETL and messaging are still popular, but they are joined by event-driven microservices, real-time stream processing, IoT, machine learning pipelines, and hundreds of industry-specific use cases and patterns that range from claims processing in insurance companies to trading systems in banks to helping power real-time game play and personalization in video games and streaming services.

Even as Kafka expands to new environments and use cases, writing applications that use Kafka well and deploy it confidently in production requires acclimating to Kafka's unique way of thinking. This book covers everything developers and SREs need to use Kafka to its full potential, from the most basic APIs and configuration to the latest and most cutting-edge capabilities. It covers not just what you can do with Kafka and how to do it, but also what not to do and antipatterns to avoid. This book can be a trusted guide to the world of Kafka for both new users and experienced practitioners.

— Jay Kreps
Cofounder and CEO at Confluent

Foreword to the First Edition

It's an exciting time for Apache Kafka. Kafka is being used by tens of thousands of organizations, including over a third of the Fortune 500 companies. It's among the fastest-growing open source projects and has spawned an immense ecosystem around it. It's at the heart of a movement toward managing and processing streams of data.

So where did Kafka come from? Why did we build it? And what exactly is it?

Kafka got its start as an internal infrastructure system we built at LinkedIn. Our observation was really simple: there were lots of databases and other systems built to *store* data, but what was missing in our architecture was something that would help us to handle the continuous *flow* of data. Prior to building Kafka, we experimented with all kinds of off-the-shelf options, from messaging systems to log aggregation and ETL tools, but none of them gave us what we wanted.

We eventually decided to build something from scratch. Our idea was that instead of focusing on holding piles of data like our relational databases, key-value stores, search indexes, or caches, we would focus on treating data as a continually evolving and ever-growing stream and build a data system—and indeed a data architecture—oriented around that idea.

This idea turned out to be even more broadly applicable than we expected. Though Kafka got its start powering real-time applications and data flow behind the scenes of a social network, you can now see it at the heart of next-generation architectures in every industry imaginable. Big retailers are reworking their fundamental business processes around continuous data streams, car companies are collecting and processing real-time data streams from internet-connected cars, and banks are rethinking their fundamental processes and systems around Kafka as well.

So what is this Kafka thing all about? How does it compare to the systems you already know and use?

We've come to think of Kafka as a *streaming platform*: a system that lets you publish and subscribe to streams of data, store them, and process them, and that is exactly what Apache Kafka is built to be. Getting used to this way of thinking about data might be a little different than what you're used to, but it turns out to be an incredibly powerful abstraction for building applications and architectures. Kafka is often compared to a couple of existing technology categories: enterprise messaging systems, big data systems like Hadoop, and data integration or ETL tools. Each of these comparisons has some validity but also falls a little short.

Kafka is like a messaging system in that it lets you publish and subscribe to streams of messages. In this way, it is similar to products like ActiveMQ, RabbitMQ, IBM's MQSeries, and other products. But even with these similarities, Kafka has a number of core differences from traditional messaging systems that make it another kind of animal entirely. Here are the big three differences: first, it works as a modern distributed system that runs as a cluster and can scale to handle all the applications in even the most massive of companies. Rather than running dozens of individual messaging brokers, hand wired to different apps, this lets you have a central platform that can scale elastically to handle all the streams of data in a company. Second, Kafka is a true storage system built to store data for as long as you might like. This has huge advantages in using it as a connecting layer as it provides real delivery guarantees—its data is replicated, persistent, and can be kept around as long as you like. Finally, the world of stream processing raises the level of abstraction quite significantly. Messaging systems mostly just hand out messages. The stream processing capabilities in Kafka let you compute derived streams and datasets dynamically off of your streams with far less code. These differences make Kafka enough of its own thing that it doesn't really make sense to think of it as "yet another queue."

Another view on Kafka—and one of our motivating lenses in designing and building it—was to think of it as a kind of real-time version of Hadoop. Hadoop lets you store and periodically process file data at a very large scale. Kafka lets you store and continuously process streams of data, also at a large scale. At a technical level, there are definitely similarities, and many people see the emerging area of stream processing as a superset of the kind of batch processing people have done with Hadoop and its various processing layers. What this comparison misses is that the use cases that continuous, low-latency processing opens up are quite different from those that naturally fall on a batch processing system. Whereas Hadoop and big data targeted analytics applications, often in the data warehousing space, the low-latency nature of Kafka makes it applicable for the kind of core applications that directly power a business. This makes sense: events in a business are happening all the time, and the ability to react to them as they occur makes it much easier to build services that directly power the operation of the business, feed back into customer experiences, and so on.

The final area Kafka gets compared to is ETL or data integration tools. After all, these tools move data around, and Kafka moves data around. There is some validity to this as well, but I think the core difference is that Kafka has inverted the problem. Rather than a tool for scraping data out of one system and inserting it into another, Kafka is a platform oriented around real-time streams of events. This means that not only can it connect off-the-shelf applications and data systems, it can also power custom applications built to trigger off of these same data streams. We think this architecture centered around streams of events is a really important thing. In some ways these flows of data are the most central aspect of a modern digital company, as important as the cash flows you'd see in a financial statement.

The ability to combine these three areas—to bring all the streams of data together across all the use cases—is what makes the idea of a streaming platform so appealing to people.

Still, all of this is a bit different, and learning how to think and build applications oriented around continuous streams of data is quite a mindshift if you are coming from the world of request/response-style applications and relational databases. This book is absolutely the best way to learn about Kafka, from internals to APIs, written by some of the people who know it best. I hope you enjoy reading it as much as I have!

— Jay Kreps
Cofounder and CEO at Confluent

Preface

The greatest compliment you can give an author of a technical book is "This is the book I wish I had when I got started with this subject." This is the goal we set for ourselves when we started writing this book. We looked back at our experience writing Kafka, running Kafka in production, and helping many companies use Kafka to build software architectures and manage their data pipelines, and we asked ourselves, "What are the most useful things we can share with new users to take them from beginner to expert?" This book is a reflection of the work we do every day: run Apache Kafka and help others use it in the best ways.

We included what we believe you need to know in order to successfully run Apache Kafka in production and build robust and performant applications on top of it. We highlighted the popular use cases: message buses for event-driven microservices, stream-processing applications, and large-scale data pipelines. We also focused on making the book general and comprehensive enough so it will be useful to anyone using Kafka, no matter the use case or architecture. We cover practical matters such as how to install and configure Kafka and how to use the Kafka APIs, and we also dedicate space to Kafka's design principles and reliability guarantees, and explore several of Kafka's delightful architecture details: the replication protocol, controller, and storage layer. We believe that knowledge of Kafka's design and internals is not only a fun read for those interested in distributed systems but is also incredibly useful for those who are seeking to make informed decisions when they deploy Kafka in production and design applications that use Kafka. The better you understand how Kafka works, the more you can make informed decisions regarding the many tradeoffs that are involved in engineering.

One of the problems in software engineering is that there is always more than one way to do anything. Platforms such as Apache Kafka provide plenty of flexibility, which is great for experts but makes for a steep learning curve for beginners. Very often, Apache Kafka tells you how to use a feature but not why you should or shouldn't use it. Whenever possible, we try to clarify the existing choices, the

tradeoffs involved, and when you should and shouldn't use the different options presented by Apache Kafka.

Who Should Read This Book

Kafka: The Definitive Guide was written for software engineers who develop applications that use Kafka's APIs, and for production engineers (also called SREs, DevOps, or sysadmins) who install, configure, tune, and monitor Kafka in production. We also wrote the book with data architects and data engineers in mind—those responsible for designing and building an organization's entire data infrastructure. Some of the chapters, especially Chapters 3, 4, and 14, are geared toward Java developers. Those chapters assume that the reader is familiar with the basics of the Java programming language, including topics such as exception handling and concurrency. Other chapters, especially Chapters 2, 10, 12, and 13, assume the reader has some experience running Linux and some familiarity with storage and network configuration in Linux. The rest of the book discusses Kafka and software architectures in more general terms and does not assume special knowledge.

Another category of people who may find this book interesting are the managers and architects who don't work directly with Kafka but work with the people who do. It is just as important that they understand the guarantees that Kafka provides and the trade-offs that their employees and coworkers will need to make while building Kafka-based systems. The book can provide ammunition to managers who would like to get their staff trained in Apache Kafka or ensure that their teams know what they need to know.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Kafka: The Definitive Guide* by Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty (O'Reilly). Copyright 2021 Chen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty, 978-1-491-93616-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/kafka-tdg2*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

We would like to thank the many contributors to Apache Kafka and its ecosystem. Without their work, this book would not exist. Special thanks to Jay Kreps, Neha Narkhede, and Jun Rao, as well as their colleagues and the leadership at LinkedIn, for cocreating Kafka and contributing it to the Apache Software Foundation.

Many people provided valuable feedback on early versions of the book, and we appreciate their time and expertise: Apurva Mehta, Arseniy Tashoyan, Dylan Scott, Ewen Cheslack-Postava, Grant Henke, Ismael Juma, James Cheng, Jason Gustafson, Jeff Holoman, Joel Koshy, Jonathan Seidman, Jun Rao, Matthias Sax, Michael Noll, Paolo Castagna, and Jesse Anderson. We also want to thank the many readers who left comments and feedback via the rough-cuts feedback site.

Many reviewers helped us out and greatly improved the quality of this book, so any mistakes left are our own.

We'd like to thank our O'Reilly first-edition editor, Shannon Cutt, for her encouragement and patience, and for being far more on top of things than we were. Our second-edition editors, Jess Haberman and Gary O'Brien, kept us on track through global challenges. Working with O'Reilly is a great experience for an author—the support they provide, from tools to book signings, is unparalleled. We are grateful to everyone involved in making this happen, and we appreciate their choice to work with us.

And we'd like to thank our managers and colleagues for enabling and encouraging us while writing the book.

Gwen wants to thank her husband, Omer Shapira, for his support and patience during the many months spent writing yet another book; her cats, Luke and Lea, for being cuddly; and her dad, Lior Shapira, for teaching her to always say yes to opportunities, even when it seems daunting.

Todd would be nowhere without his wife, Marcy, and daughters, Bella and Kaylee, behind him all the way. Their support for all the extra time writing, and long hours running to clear his head, keeps him going.

Rajini would like to thank her husband, Manjunath, and son, Tarun, for their unwavering support and encouragement, for spending weekends reviewing the early drafts, and for always being there for her.

Krit shares his love and gratitude with his wife, Cecilia, and two children, Lucas and Lizabeth. Their love and support make every day a joy, and he wouldn't be able to pursue his passions without them. He also wants to thank his mom, Cindy Petty, for instilling in Krit a desire to always be the best version of himself.

CHAPTER 1
Meet Kafka

Every enterprise is powered by data. We take information in, analyze it, manipulate it, and create more as output. Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data has a story to tell, something of importance that will inform the next thing to be done. In order to know what that is, we need to get the data from where it is created to where it can be analyzed. We see this every day on websites like Amazon, where our clicks on items of interest to us are turned into recommendations that are shown to us a little later.

The faster we can do this, the more agile and responsive our organizations can be. The less effort we spend on moving data around, the more we can focus on the core business at hand. This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

> Any time scientists disagree, it's because we have insufficient data. Then we can agree on what kind of data to get; we get the data; and the data solves the problem. Either I'm right, or you're right, or we're both wrong. And we move on.
>
> —Neil deGrasse Tyson

Publish/Subscribe Messaging

Before discussing the specifics of Apache Kafka, it is important for us to understand the concept of publish/subscribe messaging and why it is a critical component of data-driven applications. *Publish/subscribe (pub/sub) messaging* is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub

systems often have a broker, a central point where messages are published, to facilitate this pattern.

How It Starts

Many use cases for publish/subscribe start out the same way: with a simple message queue or interprocess communication channel. For example, you create an application that needs to send monitoring information somewhere, so you open a direct connection from your application to an app that displays your metrics on a dashboard, and push metrics over that connection, as seen in Figure 1-1.

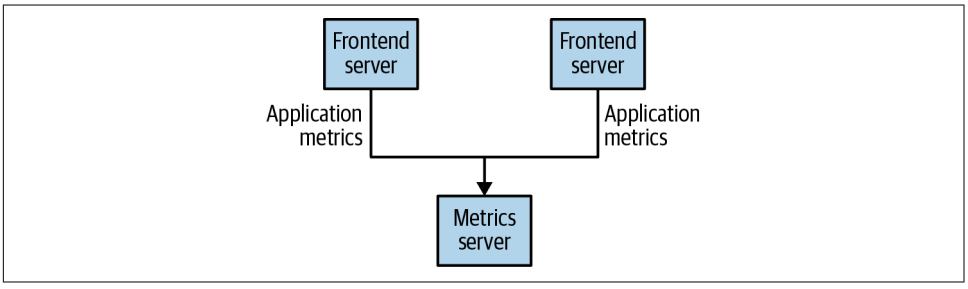

Figure 1-1. A single, direct metrics publisher

This is a simple solution to a simple problem that works when you are getting started with monitoring. Before long, you decide you would like to analyze your metrics over a longer term, and that doesn't work well in the dashboard. You start a new service that can receive metrics, store them, and analyze them. In order to support this, you modify your application to write metrics to both systems. By now you have three more applications that are generating metrics, and they all make the same connections to these two services. Your coworker thinks it would be a good idea to do active polling of the services for alerting as well, so you add a server on each of the applications to provide metrics on request. After a while, you have more applications that are using those servers to get individual metrics and use them for various purposes. This architecture can look much like Figure 1-2, with connections that are even harder to trace.

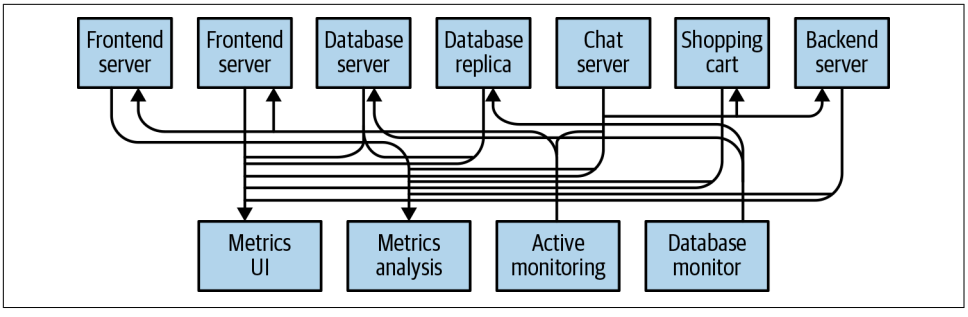

Figure 1-2. Many metrics publishers, using direct connections

The technical debt built up here is obvious, so you decide to pay some of it back. You set up a single application that receives metrics from all the applications out there, and provide a server to query those metrics for any system that needs them. This reduces the complexity of the architecture to something similar to Figure 1-3. Congratulations, you have built a publish/subscribe messaging system!

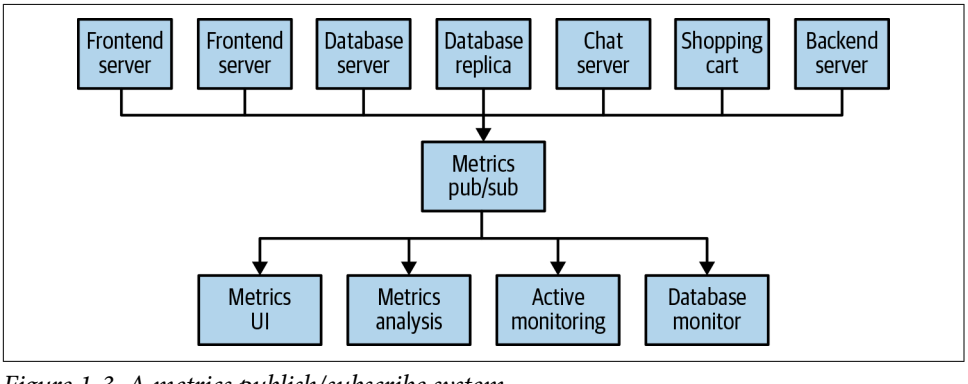

Figure 1-3. A metrics publish/subscribe system

Individual Queue Systems

At the same time that you have been waging this war with metrics, one of your coworkers has been doing similar work with log messages. Another has been working on tracking user behavior on the frontend website and providing that information to developers who are working on machine learning, as well as creating some reports for management. You have all followed a similar path of building out systems that decouple the publishers of the information from the subscribers to that information. Figure 1-4 shows such an infrastructure, with three separate pub/sub systems.

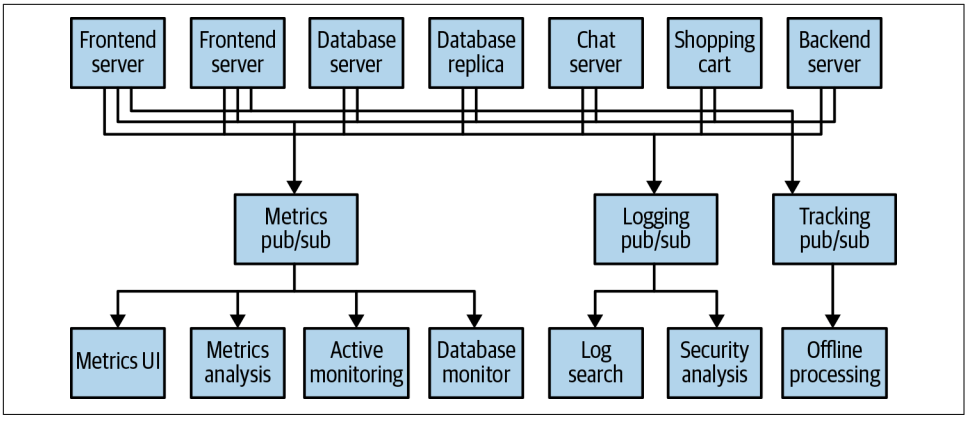

Figure 1-4. Multiple publish/subscribe systems

This is certainly a lot better than utilizing point-to-point connections (as in Figure 1-2), but there is a lot of duplication. Your company is maintaining multiple systems for queuing data, all of which have their own individual bugs and limitations. You also know that there will be more use cases for messaging coming soon. What you would like to have is a single centralized system that allows for publishing generic types of data, which will grow as your business grows.

Enter Kafka

Apache Kafka was developed as a publish/subscribe messaging system designed to solve this problem. It is often described as a "distributed commit log" or more recently as a "distributing streaming platform." A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages and Batches

The unit of data within Kafka is called a *message*. If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*. A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. A message can have an optional piece of metadata, which is referred to as a *key*. The key is also a byte array and, as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key and then select the partition number for that

message by taking the result of the hash modulo the total number of partitions in the topic. This ensures that messages with the same key are always written to the same partition (provided that the partition count does not change).

For efficiency, messages are written into Kafka in batches. A *batch* is just a collection of messages, all of which are being produced to the same topic and partition. An individual round trip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. Of course, this is a trade-off between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power. Both keys and batches are discussed in more detail in Chapter 3.

Schemas

While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood. There are many options available for message *schema*, depending on your application's individual needs. Simplistic systems, such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML), are easy to use and human readable. However, they lack features such as robust type handling and compatibility between schema versions. Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro provides a compact serialization format, schemas that are separate from the message payloads and that do not require code to be generated when they change, and strong data typing and schema evolution, with both backward and forward compatibility.

A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled. When these tasks are tightly coupled, applications that subscribe to messages must be updated to handle the new data format, in parallel with the old format. Only then can the applications that publish the messages be updated to utilize the new format. By using well-defined schemas and storing them in a common repository, the messages in Kafka can be understood without coordination. Schemas and serialization are covered in more detail in Chapter 3.

Topics and Partitions

Messages in Kafka are categorized into *topics*. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*. Going back to the "commit log" description, a partition is a single log. Messages are written to it in an append-only fashion and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message ordering across the entire topic, just within a single

partition. Figure 1-5 shows a topic with four partitions, with writes being appended to the end of each one. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server. Additionally, partitions can be replicated, such that different servers will store a copy of the same partition in case one server fails.

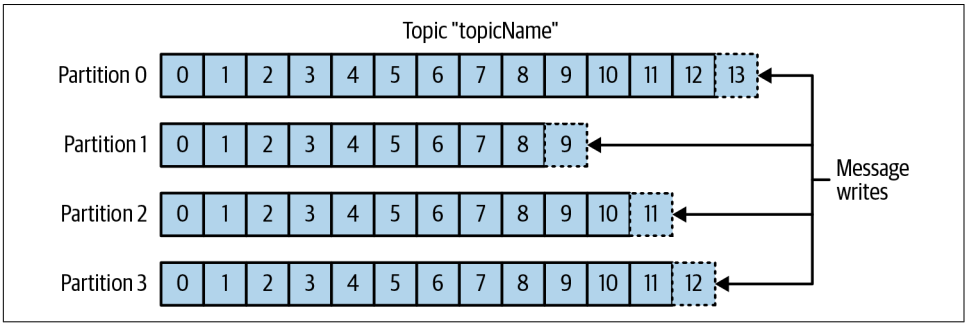

Figure 1-5. Representation of a topic with multiple partitions

The term *stream* is often used when discussing data within systems like Kafka. Most often, a stream is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers. This way of referring to messages is most common when discussing stream processing, which is when frameworks—some of which are Kafka Streams, Apache Samza, and Storm—operate on the messages in real time. This method of operation can be compared to the way offline frameworks, namely Hadoop, are designed to work on bulk data at a later time. An overview of stream processing is provided in Chapter 14.

Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. There are also advanced client APIs—Kafka Connect API for data integration and Kafka Streams for stream processing. The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Producers create new messages. In other publish/subscribe systems, these may be called *publishers* or *writers*. A message will be produced to a specific topic. By default, the producer will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This ensures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that

follows other business rules for mapping messages to partitions. Producers are covered in more detail in Chapter 3.

Consumers read messages. In other publish/subscribe systems, these clients may be called *subscribers* or *readers*. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced to each partition. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The *offset*—an integer value that continually increases—is another piece of metadata that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset, and the following message has a greater offset (though not necessarily monotonically greater). By storing the next possible offset for each partition, typically in Kafka itself, a consumer can stop and restart without losing its place.

Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic. The group ensures that each partition is only consumed by one member. In Figure 1-6, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer.

In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will reassign the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 4.

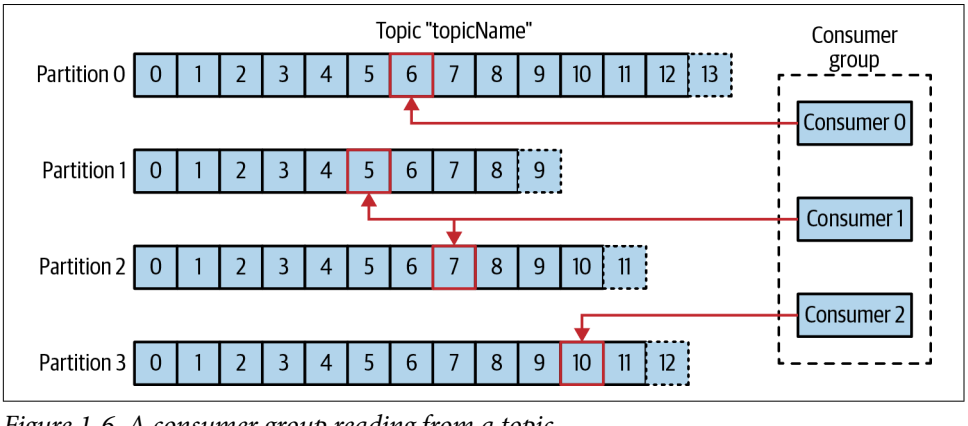

Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and writes the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one broker will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* of the partition. A replicated partition (as seen in Figure 1-7) is assigned to additional brokers, called *followers* of the partition. Replication provides redundancy of messages in the partition, such that one of the followers can take over leadership if there is a broker failure. All producers must connect to the leader in order to publish messages, but consumers may fetch from either the leader or one of the followers. Cluster operations, including partition replication, are covered in detail in Chapter 7.

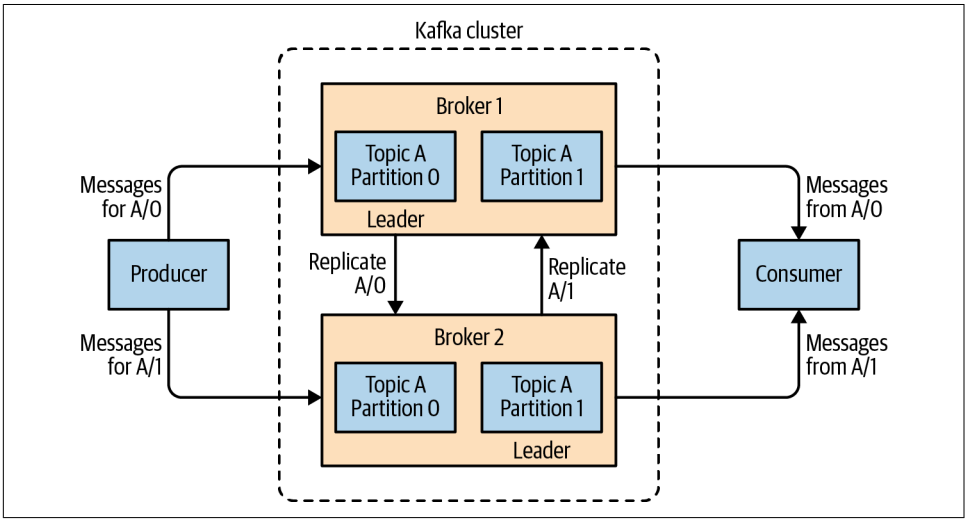

Figure 1-7. Replication of partitions in a cluster

A key feature of Apache Kafka is that of *retention*, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the partition reaches a certain size in bytes (e.g., 1 GB). Once these limits are reached, messages are expired and deleted. In this way, the retention configuration

defines a minimum amount of data available at any time. Individual topics can also be configured with their own retention settings so that messages are stored for only as long as they are useful. For example, a tracking topic might be retained for several days, whereas application metrics might be retained for only a few hours. Topics can also be configured as *log compacted*, which means that Kafka will retain only the last message produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Multiple Clusters

As Kafka deployments grow, it is often advantageous to have multiple clusters. There are several reasons why this can be useful:

- Segregation of types of data
- Isolation for security requirements
- Multiple datacenters (disaster recovery)

When working with multiple datacenters in particular, it is often required that messages be copied between them. In this way, online applications can have access to user activity at both sites. For example, if a user changes public information in their profile, that change will need to be visible regardless of the datacenter in which search results are displayed. Or, monitoring data can be collected from many sites into a single central location where the analysis and alerting systems are hosted. The replication mechanisms within the Kafka clusters are designed only to work within a single cluster, not between multiple clusters.

The Kafka project includes a tool called *MirrorMaker*, used for replicating data to other clusters. At its core, MirrorMaker is simply a Kafka consumer and producer, linked together with a queue. Messages are consumed from one Kafka cluster and produced to another. Figure 1-8 shows an example of an architecture that uses MirrorMaker, aggregating messages from two local clusters into an aggregate cluster and then copying that cluster to other datacenters. The simple nature of the application belies its power in creating sophisticated data pipelines, which will be detailed further in Chapter 9.

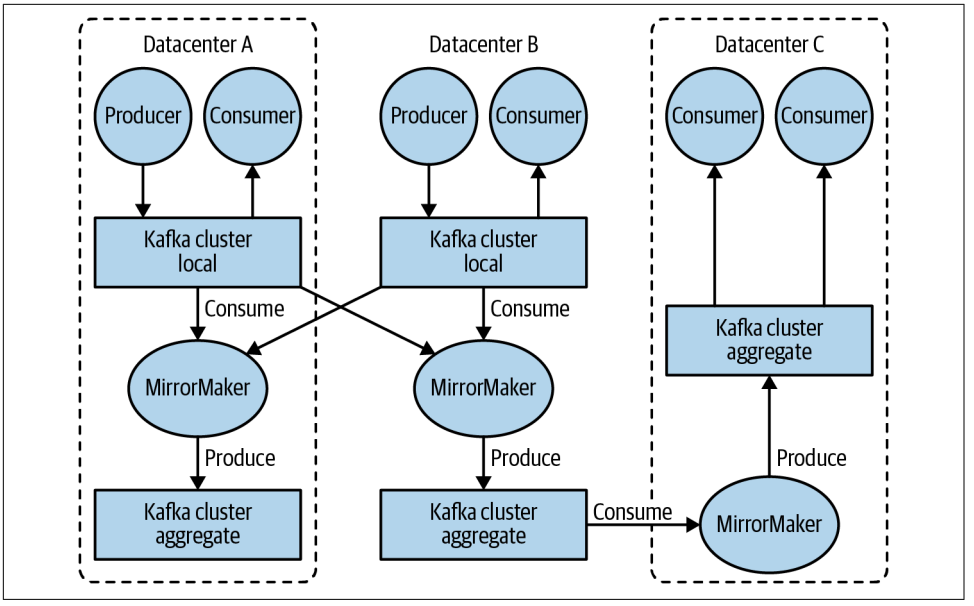

Figure 1-8. Multiple datacenters architecture

Why Kafka?

There are many choices for publish/subscribe messaging systems, so what makes Apache Kafka a good choice?

Multiple Producers

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many frontend systems and making it consistent. For example, a site that serves content to users via a number of microservices can have a single topic for page views that all services can write to using a common format. Consumer applications can then receive a single stream of page views for all applications on the site without having to coordinate consuming from multiple topics, one for each application.

Multiple Consumers

In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other client. This is in contrast to many queuing systems where once a message is consumed by one client, it is not available to any other. Multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk-Based Retention

Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. Messages are written to disk and will be stored with configurable retention rules. These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on the consumer needs. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost. Consumers can be stopped, and the messages will be retained in Kafka. This allows them to restart and pick up processing messages where they left off with no data loss.

Scalable

Kafka's flexible scalability makes it easy to handle any amount of data. Users can start with a single broker as a proof of concept, expand to a small development cluster of three brokers, and move into production with a larger cluster of tens or even hundreds of brokers that grows over time as the data scales up. Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole. This also means that a cluster of multiple brokers can handle the failure of an individual broker and continue servicing clients. Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors. Replication is discussed in more detail in Chapter 7.

High Performance

All of these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease. This can be done while still providing subsecond message latency from producing a message to availability to consumers.

Platform Features

The core Apache Kafka project has also added some streaming platform features that can make it much easier for developers to perform common types of work. While not full platforms, which typically include a structured runtime environment like YARN, these features are in the form of APIs and libraries that provide a solid foundation to build on and flexibility as to where they can be run. Kafka Connect assists with the task of pulling data from a source data system and pushing it into Kafka, or pulling data from Kafka and pushing it into a sink data system. Kafka Streams provides a library for easily developing stream processing applications that are scalable and fault

tolerant. Connect is discussed in Chapter 9, while Streams is covered in great detail in Chapter 14.

The Data Ecosystem

Many applications participate in the environments we build for data processing. We have defined inputs in the form of applications that create data or otherwise introduce it to the system. We have defined outputs in the form of metrics, reports, and other data products. We create loops, with some components reading data from the system, transforming it using data from other sources, and then introducing it back into the data infrastructure to be used elsewhere. This is done for numerous types of data, with each having unique qualities of content, size, and usage.

Apache Kafka provides the circulatory system for the data ecosystem, as shown in Figure 1-9. It carries messages between the various members of the infrastructure, providing a consistent interface for all clients. When coupled with a system to provide message schemas, producers and consumers no longer require tight coupling or direct connections of any sort. Components can be added and removed as business cases are created and dissolved, and producers do not need to be concerned about who is using the data or the number of consuming applications.

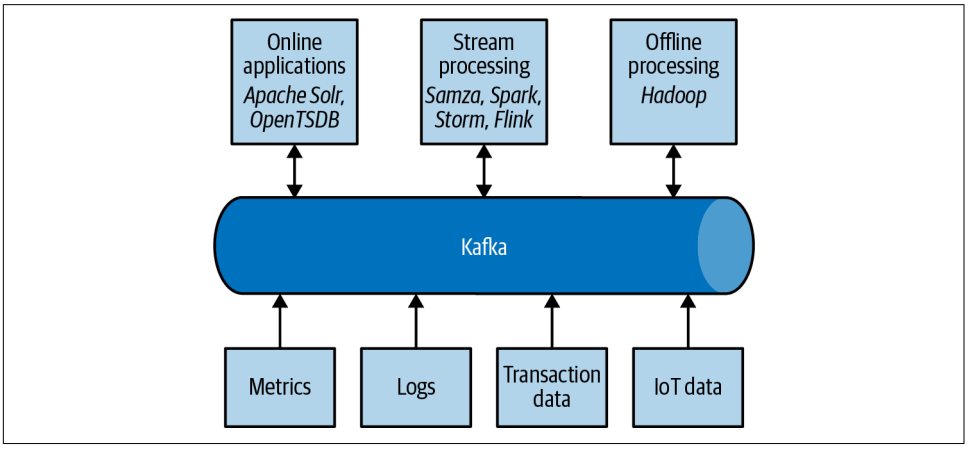

Figure 1-9. A big data ecosystem

Use Cases

Activity tracking

The original use case for Kafka, as it was designed at LinkedIn, is that of user activity tracking. A website's users interact with frontend applications, which generate messages regarding actions the user is taking. This can be passive information, such as

page views and click tracking, or it can be more complex actions, such as information that a user adds to their profile. The messages are published to one or more topics, which are then consumed by applications on the backend. These applications may be generating reports, feeding machine learning systems, updating search results, or performing other operations that are necessary to provide a rich user experience.

Messaging

Kafka is also used for messaging, where applications need to send notifications (such as emails) to users. Those applications can produce messages without needing to be concerned about formatting or how the messages will actually be sent. A single application can then read all the messages to be sent and handle them consistently, including:

- Formatting the messages (also known as *decorating*) using a common look and feel
- Collecting multiple messages into a single notification to be sent
- Applying a user's preferences for how they want to receive messages

Using a single application for this avoids the need to duplicate functionality in multiple applications, as well as allows operations like aggregation that would not otherwise be possible.

Metrics and logging

Kafka is also ideal for collecting application and system metrics and logs. This is a use case in which the ability to have multiple applications producing the same type of message shines. Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer-term analysis, such as growth projections. Log messages can be published in the same way and can be routed to dedicated log search systems like Elasticsearch or security analysis applications. Another added benefit of Kafka is that when the destination system needs to change (e.g., it's time to update the log storage system), there is no need to alter the frontend applications or the means of aggregation.

Commit log

Since Kafka is based on the concept of a commit log, database changes can be published to Kafka, and applications can easily monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view. Durable retention is useful here for providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the

consuming applications. Alternately, log-compacted topics can be used to provide longer retention by only retaining a single change per key.

Stream processing

Another area that provides numerous types of applications is stream processing. While almost all usage of Kafka can be thought of as stream processing, the term is typically used to refer to applications that provide similar functionality to map/reduce processing in Hadoop. Hadoop usually relies on aggregation of data over a long time frame, either hours or days. Stream processing operates on data in real time, as quickly as messages are produced. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources. Stream processing is covered in Chapter 14.

Kafka's Origin

Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time.

> Data really powers everything that we do.
>
> —Jeff Weiner, former CEO of LinkedIn

LinkedIn's Problem

Similar to the example described at the beginning of this chapter, LinkedIn had a system for collecting system and application metrics that used custom collectors and open source tools for storing and presenting data internally. In addition to traditional metrics, such as CPU usage and application performance, there was a sophisticated request-tracing feature that used the monitoring system and could provide introspection into how a single user request propagated through internal applications. The monitoring system had many faults, however. This included metrics collection based on polling, large intervals between metrics, and no ability for application owners to manage their own metrics. The system was high-touch, requiring human intervention for most simple tasks, and inconsistent, with differing metric names for the same measurement across different systems.

At the same time, there was a system created for tracking user activity information. This was an HTTP service that frontend servers would connect to periodically and publish a batch of messages (in XML format) to the HTTP service. These batches were then moved to offline processing platforms, which is where the files were parsed and collated. This system had many faults. The XML formatting was inconsistent,

and parsing it was computationally expensive. Changing the type of user activity that was tracked required a significant amount of coordinated work between frontends and offline processing. Even then, the system would break constantly due to changing schemas. Tracking was built on hourly batching, so it could not be used in real time.

Monitoring and user-activity tracking could not use the same backend service. The monitoring service was too clunky, the data format was not oriented for activity tracking, and the polling model for monitoring was not compatible with the push model for tracking. At the same time, the tracking service was too fragile to use for metrics, and the batch-oriented processing was not the right model for real-time monitoring and alerting. However, the monitoring and tracking data shared many traits, and correlation of the information (such as how specific types of user activity affected application performance) was highly desirable. A drop in specific types of user activity could indicate problems with the application that serviced it, but hours of delay in processing activity batches meant a slow response to these types of issues.

At first, existing off-the-shelf open source solutions were thoroughly investigated to find a new system that would provide real-time access to the data and scale out to handle the amount of message traffic needed. Prototype systems were set up using ActiveMQ, but at the time it could not handle the scale. It was also a fragile solution for the way LinkedIn needed to use it, discovering many flaws in ActiveMQ that would cause the brokers to pause. These pauses would back up connections to clients and interfere with the ability of the applications to serve requests to users. The decision was made to move forward with a custom infrastructure for the data pipeline.

The Birth of Kafka

The development team at LinkedIn was led by Jay Kreps, a principal software engineer who was previously responsible for the development and open source release of Voldemort, a distributed key-value storage system. The initial team also included Neha Narkhede and, later, Jun Rao. Together, they set out to create a messaging system that could meet the needs of both the monitoring and tracking systems, and scale for the future. The primary goals were to:

- Decouple producers and consumers by using a push-pull model
- Provide persistence for message data within the messaging system to allow multiple consumers
- Optimize for high throughput of messages
- Allow for horizontal scaling of the system to grow as the data streams grew

The result was a publish/subscribe messaging system that had an interface typical of messaging systems but a storage layer more like a log-aggregation system. Combined with the adoption of Apache Avro for message serialization, Kafka was effective for

handling both metrics and user-activity tracking at a scale of billions of messages per day. The scalability of Kafka has helped LinkedIn's usage grow in excess of seven trillion messages produced (as of February 2020) and over five petabytes of data consumed daily.

Open Source

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011. Apache Kafka graduated from the incubator in October of 2012. Since then, it has continuously been worked on and has found a robust community of contributors and committers outside of LinkedIn. Kafka is now used in some of the largest data pipelines in the world, including those at Netflix, Uber, and many other companies.

Widespread adoption of Kafka has created a healthy ecosystem around the core project as well. There are active meetup groups in dozens of countries around the world, providing local discussion and support of stream processing. There are also numerous open source projects related to Apache Kafka. LinkedIn continues to maintain several, including Cruise Control, Kafka Monitor, and Burrow. In addition to its commercial offerings, Confluent has released projects including ksqlDB, a schema registry, and a REST proxy under a community license (which is not strictly open source, as it includes use restrictions). Several of the most popular projects are listed in Appendix B.

Commercial Engagement

In the fall of 2014, Jay Kreps, Neha Narkhede, and Jun Rao left LinkedIn to found Confluent, a company centered around providing development, enterprise support, and training for Apache Kafka. They also joined other companies (such as Heroku) in providing cloud services for Kafka. Confluent, through a partnership with Google, provides managed Kafka clusters on Google Cloud Platform, as well as similar services on Amazon Web Services and Azure. One of the other major initiatives of Confluent is to organize the Kafka Summit conference series. Started in 2016, with conferences held annually in the United States and London, Kafka Summit provides a place for the community to come together on a global scale and share knowledge about Apache Kafka and related projects.

The Name

People often ask how Kafka got its name and if it signifies anything specific about the application itself. Jay Kreps offered the following insight:

> I thought that since Kafka was a system optimized for writing, using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.
>
> So basically there is not much of a relationship.

Getting Started with Kafka

Now that we know all about Kafka and its history, we can set it up and build our own data pipeline. In the next chapter, we will explore installing and configuring Kafka. We will also cover selecting the right hardware to run Kafka on, and some things to keep in mind when moving to production operations.

CHAPTER 2

Installing Kafka

This chapter describes how to get started with the Apache Kafka broker, including how to set up Apache ZooKeeper, which is used by Kafka for storing metadata for the brokers. The chapter will also cover basic configuration options for Kafka deployments, as well as some suggestions for selecting the correct hardware to run the brokers on. Finally, we cover how to install multiple Kafka brokers as part of a single cluster and things you should know when using Kafka in a production environment.

Environment Setup

Before using Apache Kafka, your environment needs to be set up with a few prerequisites to ensure it runs properly. The following sections will guide you through that process.

Choosing an Operating System

Apache Kafka is a Java application and can run on many operating systems. While Kafka is capable of being run on many OSs, including Windows, macOS, Linux, and others, Linux is the recommended OS for the general use case. The installation steps in this chapter will focus on setting up and using Kafka in a Linux environment. For information on installing Kafka on Windows and macOS, see Appendix A.

Installing Java

Prior to installing either ZooKeeper or Kafka, you will need a Java environment set up and functioning. Kafka and ZooKeeper work well with all OpenJDK-based Java implementations, including Oracle JDK. The latest versions of Kafka support both Java 8 and Java 11. The exact version installed can be the version provided by your OS or one directly downloaded from the web—for example, the Oracle website for the

Oracle version. Though ZooKeeper and Kafka will work with a runtime edition of Java, it is recommended when developing tools and applications to have the full Java Development Kit (JDK). It is recommended to install the latest released patch version of your Java environment, as older versions may have security vulnerabilities. The installation steps will assume you have installed JDK version 11 update 10 deployed at */usr/java/jdk-11.0.10*.

Installing ZooKeeper

Apache Kafka uses Apache ZooKeeper to store metadata about the Kafka cluster, as well as consumer client details, as shown in Figure 2-1. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. This book won't go into extensive detail about ZooKeeper but will limit explanations to only what is needed to operate Kafka. While it is possible to run a ZooKeeper server using scripts contained in the Kafka distribution, it is trivial to install a full version of ZooKeeper from the distribution.

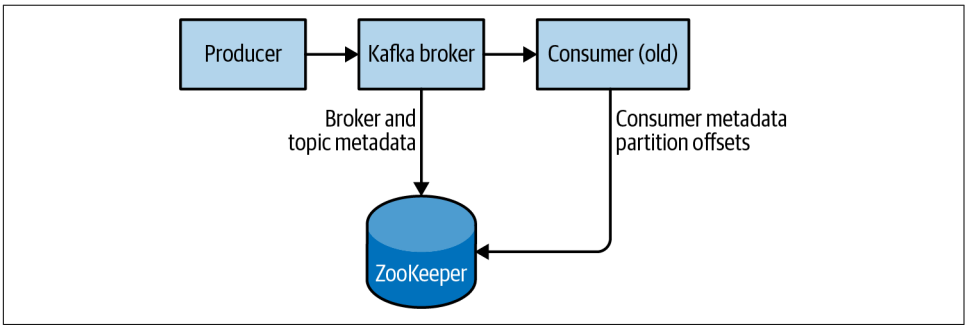

Figure 2-1. Kafka and ZooKeeper

Kafka has been tested extensively with the stable 3.5 release of ZooKeeper and is regularly updated to include the latest release. In this book, we will be using ZooKeeper 3.5.9, which can be downloaded from the ZooKeeper website.

Standalone server

ZooKeeper comes with a base example config file that will work well for most use cases in */usr/local/zookeeper/config/zoo_sample.cfg*. However, we will manually create ours with some basic settings for demo purposes in this book. The following example installs ZooKeeper with a basic configuration in */usr/local/zookeeper*, storing its data in */var/lib/zookeeper*:

```
# tar -zxf apache-zookeeper-3.5.9-bin.tar.gz
# mv apache-zookeeper-3.5.9-bin /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cp > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

You can now validate that ZooKeeper is running correctly in standalone mode by connecting to the client port and sending the four-letter command `srvr`. This will return basic ZooKeeper information from the running server:

```
# telnet localhost 2181
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.5.9-83df9301aa5c2a5d284a9940177808c01bc35cef, built on
01/06/2021 19:49 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 5
Connection closed by foreign host.
#
```

ZooKeeper ensemble

ZooKeeper is designed to work as a cluster, called an *ensemble*, to ensure high availability. Due to the balancing algorithm used, it is recommended that ensembles contain an odd number of servers (e.g., 3, 5, and so on) as a majority of ensemble members (a *quorum*) must be working in order for ZooKeeper to respond to requests. This means that in a three-node ensemble, you can run with one node missing. With a five-node ensemble, you can run with two nodes missing.

Sizing Your ZooKeeper Ensemble

Consider running ZooKeeper in a five-node ensemble. To make configuration changes to the ensemble, including swapping a node, you will need to reload nodes one at a time. If your ensemble cannot tolerate more than one node being down, doing maintenance work introduces additional risk. It is also not recommended to run more than seven nodes, as performance can start to degrade due to the nature of the consensus protocol.

Additionally, if you feel that five or seven nodes aren't supporting the load due to too many client connections, consider adding additional observer nodes for help in balancing read-only traffic.

To configure ZooKeeper servers in an ensemble, they must have a common configuration that lists all servers, and each server needs a *myid* file in the data directory that specifies the ID number of the server. If the hostnames of the servers in the ensemble are `zoo1.example.com`, `zoo2.example.com`, and `zoo3.example.com`, the configuration file might look like this:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

In this configuration, the `initLimit` is the amount of time to allow followers to connect with a leader. The `syncLimit` value limits how long out-of-sync followers can be with the leader. Both values are a number of `tickTime` units, which makes the `initLimit` 20 × 2,000 ms, or 40 seconds. The configuration also lists each server in the ensemble. The servers are specified in the format *`server.X=hostname:peerPort:leaderPort`*, with the following parameters:

`X`
: The ID number of the server. This must be an integer, but it does not need to be zero-based or sequential.

`hostname`
: The hostname or IP address of the server.

`peerPort`
: The TCP port over which servers in the ensemble communicate with one another.

`leaderPort`
: The TCP port over which leader election is performed.

Clients only need to be able to connect to the ensemble over the *`clientPort`*, but the members of the ensemble must be able to communicate with one another over all three ports.

In addition to the shared configuration file, each server must have a file in the *dataDir* directory with the name *myid*. This file must contain the ID number of the server, which must match the configuration file. Once these steps are complete, the servers will start up and communicate with one another in an ensemble.

Testing ZooKeeper Ensemble on a Single Machine

It is possible to test and run a ZooKeeper ensemble on a single machine by specifying all hostnames in the config as `localhost` and have unique ports specified for *`peerPort`* and *`leaderPort`* for each instance. Additionally, a separate *zoo.cfg* would need to be created for each instance with a unique *dataDir* and *`clientPort`* defined for each instance. This can be useful for testing purposes only, but it is *not* recommended for production systems.

Installing a Kafka Broker

Once Java and ZooKeeper are configured, you are ready to install Apache Kafka. The current release can be downloaded from the Kafka website. At press time, that version is 2.8.0 running under Scala version 2.13.0. The examples in this chapters are shown using version 2.7.0.

The following example installs Kafka in */usr/local/kafka*, configured to use the ZooKeeper server started previously and to store the message log segments stored in */tmp/kafka-logs*:

```
# tar -zxf kafka_2.13-2.7.0.tgz
# mv kafka_2.13-2.7.0 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Once the Kafka broker is started, we can verify that it is working by performing some simple operations against the cluster: creating a test topic, producing some messages, and consuming the same messages.

Create and verify a topic:

```
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --
create
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092
--describe --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
#
```

Produce messages to a test topic (use Ctrl-C to stop the producer at any time):

```
# /usr/local/kafka/bin/kafka-console-producer.sh --bootstrap-server
localhost:9092 --topic test
Test Message 1
Test Message 2
^C
#
```

Consume messages from a test topic:

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic test --from-beginning
Test Message 1
Test Message 2
^C
Processed a total of 2 messages
#
```

Deprecation of ZooKeeper Connections on Kafka CLI Utilities

If you are familiar with older versions of the Kafka utilities, you may be used to using a `--zookeeper` connection string. This has been deprecated in almost all cases. The current best practice is to use the newer `--bootstrap-server` option and connect directly to the Kafka broker. If you are running in a cluster, you can provide the host:port of any broker in the cluster.

Configuring the Broker

The example configuration provided with the Kafka distribution is sufficient to run a standalone server as a proof of concept, but most likely will not be sufficient for large installations. There are numerous configuration options for Kafka that control all aspects of setup and tuning. Most of the options can be left at the default settings, though, as they deal with tuning aspects of the Kafka broker that will not be applicable until you have a specific use case that requires adjusting these settings.

General Broker Parameters

There are several broker configuration parameters that should be reviewed when deploying Kafka for any environment other than a standalone broker on a single server. These parameters deal with the basic configuration of the broker, and most of them must be changed to run properly in a cluster with other brokers.

broker.id

Every Kafka broker must have an integer identifier, which is set using the `broker.id` configuration. By default, this integer is set to `0`, but it can be any value. It is essential that the integer must be unique for each broker within a single Kafka cluster. The selection of this number is technically arbitrary, and it can be moved between brokers if necessary for maintenance tasks. However, it is highly recommended to set this value to something intrinsic to the host so that when performing maintenance it is not onerous to map broker ID numbers to hosts. For example, if your hostnames contain a unique number (such as `host1.example.com`, `host2.example.com`, etc.), then `1` and `2` would be good choices for the `broker.id` values, respectively.

listeners

Older versions of Kafka used a simple `port` configuration. This can still be used as a backup for simple configurations but is a deprecated config. The example configuration file starts Kafka with a listener on TCP port 9092. The new `listeners` config is a comma-separated list of URIs that we listen on with the listener names. If the listener name is not a common security protocol, then another config `listener.security.protocol.map` must also be configured. A listener is defined as *`<protocol>://<hostname>:<port>`*. An example of a legal `listener` config is `PLAINTEXT://localhost:9092,SSL://:9091`. Specifying the hostname as `0.0.0.0` will bind to all interfaces. Leaving the hostname empty will bind it to the default interface. Keep in mind that if a port lower than 1024 is chosen, Kafka must be started as root. Running Kafka as root is not a recommended configuration.

zookeeper.connect

The location of the ZooKeeper used for storing the broker metadata is set using the `zookeeper.connect` configuration parameter. The example configuration uses a ZooKeeper running on port 2181 on the local host, which is specified as `localhost:2181`. The format for this parameter is a semicolon-separated list of `hostname:port/path` strings, which include:

`hostname`
: The hostname or IP address of the ZooKeeper server.

`port`
: The client port number for the server.

`/path`
: An optional ZooKeeper path to use as a chroot environment for the Kafka cluster. If it is omitted, the root path is used.

If a chroot path (a path designated to act as the root directory for a given application) is specified and does not exist, it will be created by the broker when it starts up.

Why Use a Chroot Path?

It is generally considered to be good practice to use a chroot path for the Kafka cluster. This allows the ZooKeeper ensemble to be shared with other applications, including other Kafka clusters, without a conflict. It is also best to specify multiple ZooKeeper servers (which are all part of the same ensemble) in this configuration. This allows the Kafka broker to connect to another member of the ZooKeeper ensemble in the event of server failure.

log.dirs

Kafka persists all messages to disk, and these log segments are stored in the directory specified in the `log.dir` configuration. For multiple directories, the config `log.dirs` is preferable. If this value is not set, it will default back to `log.dir`. `log.dirs` is a comma-separated list of paths on the local system. If more than one path is specified, the broker will store partitions on them in a "least-used" fashion, with one partition's log segments stored within the same path. Note that the broker will place a new partition in the path that has the least number of partitions currently stored in it, not the least amount of disk space used, so an even distribution of data across multiple directories is not guaranteed.

num.recovery.threads.per.data.dir

Kafka uses a configurable pool of threads for handling log segments. Currently, this thread pool is used:

- When starting normally, to open each partition's log segments
- When starting after a failure, to check and truncate each partition's log segments
- When shutting down, to cleanly close log segments

By default, only one thread per log directory is used. As these threads are only used during startup and shutdown, it is reasonable to set a larger number of threads in order to parallelize operations. Specifically, when recovering from an unclean shutdown, this can mean the difference of several hours when restarting a broker with a

large number of partitions! When setting this parameter, remember that the number configured is per log directory specified with `log.dirs`. This means that if `num.recovery.threads.per.data.dir` is set to 8, and there are 3 paths specified in `log.dirs`, this is a total of 24 threads.

auto.create.topics.enable

The default Kafka configuration specifies that the broker should automatically create a topic under the following circumstances:

- When a producer starts writing messages to the topic
- When a consumer starts reading messages from the topic
- When any client requests metadata for the topic

In many situations, this can be undesirable behavior, especially as there is no way to validate the existence of a topic through the Kafka protocol without causing it to be created. If you are managing topic creation explicitly, whether manually or through a provisioning system, you can set the `auto.create.topics.enable` configuration to `false`.

auto.leader.rebalance.enable

In order to ensure a Kafka cluster doesn't become unbalanced by having all topic leadership on one broker, this config can be specified to ensure leadership is balanced as much as possible. It enables a background thread that checks the distribution of partitions at regular intervals (this interval is configurable via `leader.imbalance.check.interval.seconds`). If leadership imbalance exceeds another config, `leader.imbalance.per.broker.percentage`, then a rebalance of preferred leaders for partitions is started.

delete.topic.enable

Depending on your environment and data retention guidelines, you may wish to lock down a cluster to prevent arbitrary deletions of topics. Disabling topic deletion can be set by setting this flag to `false`.

Topic Defaults

The Kafka server configuration specifies many default configurations for topics that are created. Several of these parameters, including partition counts and message retention, can be set per topic using the administrative tools (covered in Chapter 12). The defaults in the server configuration should be set to baseline values that are appropriate for the majority of the topics in the cluster.

Using Per-Topic Overrides

In older versions of Kafka, it was possible to specify per-topic overrides for these configurations in the broker configuration using the parameters `log.retention.hours.per.topic`, `log.retention.bytes.per.topic`, and `log.segment.bytes.per.topic`. These parameters are no longer supported, and overrides must be specified using the administrative tools.

num.partitions

The `num.partitions` parameter determines how many partitions a new topic is created with, primarily when automatic topic creation is enabled (which is the default setting). This parameter defaults to one partition. Keep in mind that the number of partitions for a topic can only be increased, never decreased. This means that if a topic needs to have fewer partitions than `num.partitions`, care will need to be taken to manually create the topic (discussed in Chapter 12).

As described in Chapter 1, partitions are the way a topic is scaled within a Kafka cluster, which makes it important to use partition counts that will balance the message load across the entire cluster as brokers are added. Many users will have the partition count for a topic be equal to, or a multiple of, the number of brokers in the cluster. This allows the partitions to be evenly distributed to the brokers, which will evenly distribute the message load. For example, a topic with 10 partitions operating in a Kafka cluster with 10 hosts with leadership balanced among all 10 hosts will have optimal throughput. This is not a requirement, however, as you can also balance message load in other ways, such as having multiple topics.

How to Choose the Number of Partitions

There are several factors to consider when choosing the number of partitions:

- What is the throughput you expect to achieve for the topic? For example, do you expect to write 100 KBps or 1 GBps?
- What is the maximum throughput you expect to achieve when consuming from a single partition? A partition will always be consumed completely by a single consumer (even when not using consumer groups, the consumer must read all messages in the partition). If you know that your slower consumer writes the data to a database and this database never handles more than 50 MBps from each thread writing to it, then you know you are limited to 50 MBps throughput when consuming from a partition.
- You can go through the same exercise to estimate the maximum throughput per producer for a single partition, but since producers are typically much faster than consumers, it is usually safe to skip this.

- If you are sending messages to partitions based on keys, adding partitions later can be very challenging, so calculate throughput based on your expected future usage, not the current usage.
- Consider the number of partitions you will place on each broker and available diskspace and network bandwidth per broker.
- Avoid overestimating, as each partition uses memory and other resources on the broker and will increase the time for metadata updates and leadership transfers.
- Will you be mirroring data? You may need to consider the throughput of your mirroring configuration as well. Large partitions can become a bottleneck in many mirroring configurations.
- If you are using cloud services, do you have IOPS (input/output operations per second) limitations on your VMs or disks? There may be hard caps on the number of IOPS allowed depending on your cloud service and VM configuration that will cause you to hit quotas. Having too many partitions can have the side effect of increasing the amount of IOPS due to the parallelism involved.

With all this in mind, it's clear that you want many partitions, but not too many. If you have some estimate regarding the target throughput of the topic and the expected throughput of the consumers, you can divide the target throughput by the expected consumer throughput and derive the number of partitions this way. So if we want to be able to write and read 1 GBps from a topic, and we know each consumer can only process 50 MBps, then we know we need at least 20 partitions. This way, we can have 20 consumers reading from the topic and achieve 1 GBps.

If you don't have this detailed information, our experience suggests that limiting the size of the partition on the disk to less than 6 GB per day of retention often gives satisfactory results. Starting small and expanding as needed is easier than starting too large.

default.replication.factor

If auto-topic creation is enabled, this configuration sets what the replication factor should be for new topics. Replication strategy can vary depending on the desired durability or availability of a cluster and will be discussed more in later chapters. The following is a brief recommendation if you are running Kafka in a cluster that will prevent outages due to factors outside of Kafka's internal capabilities, such as hardware failures.

It is highly recommended to set the replication factor to at least 1 above the `min.insync.replicas` setting. For more fault-resistant settings, if you have large enough clusters and enough hardware, setting your replication factor to 2 above the `min.insync.replicas` (abbreviated as RF++) can be preferable. RF++ will allow easier maintenance and prevent outages. The reasoning behind this recommendation is

to allow for one planned outage within the replica set and one unplanned outage to occur simultaneously. For a typical cluster, this would mean you'd have a minimum of three replicas of every partition. An example of this is if there is a network switch outage, disk failure, or some other unplanned problem during a rolling deployment or upgrade of Kafka or the underlying OS, you can be assured there will still be an additional replica available. This will be discussed more in Chapter 7.

log.retention.ms

The most common configuration for how long Kafka will retain messages is by time. The default is specified in the configuration file using the `log.retention.hours` parameter, and it is set to 168 hours, or one week. However, there are two other parameters allowed, `log.retention.minutes` and `log.retention.ms`. All three of these control the same goal (the amount of time after which messages may be deleted), but the recommended parameter to use is `log.retention.ms`, as the smaller unit size will take precedence if more than one is specified. This will ensure that the value set for `log.retention.ms` is always the one used. If more than one is specified, the smaller unit size will take precedence.

Retention by Time and Last Modified Times

Retention by time is performed by examining the last modified time (mtime) on each log segment file on disk. Under normal cluster operations, this is the time that the log segment was closed, and represents the timestamp of the last message in the file. However, when using administrative tools to move partitions between brokers, this time is not accurate and will result in excess retention for these partitions. For more information on this, see Chapter 12 discussing partition moves.

log.retention.bytes

Another way to expire messages is based on the total number of bytes of messages retained. This value is set using the `log.retention.bytes` parameter, and it is applied per partition. This means that if you have a topic with 8 partitions, and `log.retention.bytes` is set to 1 GB, the amount of data retained for the topic will be 8 GB at most. Note that all retention is performed for individual partitions, not the topic. This means that should the number of partitions for a topic be expanded, the retention will also increase if `log.retention.bytes` is used. Setting the value to –1 will allow for infinite retention.

Configuring Retention by Size and Time

If you have specified a value for both `log.retention.bytes` and `log.retention.ms` (or another parameter for retention by time), messages may be removed when either criteria is met. For example, if `log.retention.ms` is set to 86400000 (1 day) and `log.retention.bytes` is set to 1000000000 (1 GB), it is possible for messages that are less than 1 day old to get deleted if the total volume of messages over the course of the day is greater than 1 GB. Conversely, if the volume is less than 1 GB, messages can be deleted after 1 day even if the total size of the partition is less than 1 GB. It is recommended, for simplicity, to choose either size- or time-based retention—and not both—to prevent surprises and unwanted data loss, but both can be used for more advanced configurations.

log.segment.bytes

The log retention settings previously mentioned operate on log segments, not individual messages. As messages are produced to the Kafka broker, they are appended to the current log segment for the partition. Once the log segment has reached the size specified by the `log.segment.bytes` parameter, which defaults to 1 GB, the log segment is closed and a new one is opened. Once a log segment has been closed, it can be considered for expiration. A smaller log segment size means that files must be closed and allocated more often, which reduces the overall efficiency of disk writes.

Adjusting the size of the log segments can be important if topics have a low produce rate. For example, if a topic receives only 100 megabytes per day of messages, and `log.segment.bytes` is set to the default, it will take 10 days to fill one segment. As messages cannot be expired until the log segment is closed, if `log.retention.ms` is set to 604800000 (1 week), there will actually be up to 17 days of messages retained until the closed log segment expires. This is because once the log segment is closed with the current 10 days of messages, that log segment must be retained for 7 days before it expires based on the time policy (as the segment cannot be removed until the last message in the segment can be expired).

Retrieving Offsets by Timestamp

The size of the log segment also affects the behavior of fetching offsets by timestamp. When requesting offsets for a partition at a specific timestamp, Kafka finds the log segment file that was being written at that time. It does this by using the creation and last modified time of the file, and looking for a file that was created before the timestamp specified and last modified after the timestamp. The offset at the beginning of that log segment (which is also the filename) is returned in the response.

log.roll.ms

Another way to control when log segments are closed is by using the `log.roll.ms` parameter, which specifies the amount of time after which a log segment should be closed. As with the `log.retention.bytes` and `log.retention.ms` parameters, `log.segment.bytes` and `log.roll.ms` are not mutually exclusive properties. Kafka will close a log segment either when the size limit is reached or when the time limit is reached, whichever comes first. By default, there is no setting for `log.roll.ms`, which results in only closing log segments by size.

Disk Performance When Using Time-Based Segments

When using a time-based log segment limit, it is important to consider the impact on disk performance when multiple log segments are closed simultaneously. This can happen when there are many partitions that never reach the size limit for log segments, as the clock for the time limit will start when the broker starts and will always execute at the same time for these low-volume partitions.

min.insync.replicas

When configuring your cluster for data durability, setting `min.insync.replicas` to 2 ensures that at least two replicas are caught up and "in sync" with the producer. This is used in tandem with setting the producer config to ack "all" requests. This will ensure that at least two replicas (leader and one other) acknowledge a write for it to be successful. This can prevent data loss in scenarios where the leader acks a write, then suffers a failure and leadership is transferred to a replica that does not have a successful write. Without these durable settings, the producer would think it successfully produced, and the message(s) would be dropped on the floor and lost. However, configuring for higher durability has the side effect of being less efficient due to the extra overhead involved, so clusters with high-throughput that can tolerate occasional message loss aren't recommended to change this setting from the default of 1. See Chapter 7 for more information.

message.max.bytes

The Kafka broker limits the maximum size of a message that can be produced, configured by the `message.max.bytes` parameter, which defaults to 1000000, or 1 MB. A producer that tries to send a message larger than this will receive an error back from the broker, and the message will not be accepted. As with all byte sizes specified on the broker, this configuration deals with compressed message size, which means that producers can send messages that are much larger than this value uncompressed, provided they compress to under the configured `message.max.bytes` size.

There are noticeable performance impacts from increasing the allowable message size. Larger messages will mean that the broker threads that deal with processing network connections and requests will be working longer on each request. Larger messages also increase the size of disk writes, which will impact I/O throughput. Other storage solutions, such as blob stores and/or tiered storage, may be another method of addressing large disk write issues, but will not be covered in this chapter.

Coordinating Message Size Configurations

The message size configured on the Kafka broker must be coordinated with the `fetch.message.max.bytes` configuration on consumer clients. If this value is smaller than `message.max.bytes`, then consumers that encounter larger messages will fail to fetch those messages, resulting in a situation where the consumer gets stuck and cannot proceed. The same rule applies to the `replica.fetch.max.bytes` configuration on the brokers when configured in a cluster.

Selecting Hardware

Selecting an appropriate hardware configuration for a Kafka broker can be more art than science. Kafka itself has no strict requirement on a specific hardware configuration and will run without issue on most systems. Once performance becomes a concern, however, there are several factors that can contribute to the overall performance bottlenecks: disk throughput and capacity, memory, networking, and CPU. When scaling Kafka very large, there can also be constraints on the number of partitions that a single broker can handle due to the amount of metadata that needs to be updated. Once you have determined which performance types are the most critical for your environment, you can select an optimized hardware configuration appropriate for your budget.

Disk Throughput

The performance of producer clients will be most directly influenced by the throughput of the broker disk that is used for storing log segments. Kafka messages must be committed to local storage when they are produced, and most clients will wait until at least one broker has confirmed that messages have been committed before considering the send successful. This means that faster disk writes will equal lower produce latency.

The obvious decision when it comes to disk throughput is whether to use traditional spinning hard disk drives (HDDs) or solid-state disks (SSDs). SSDs have drastically lower seek and access times and will provide the best performance. HDDs, on the other hand, are more economical and provide more capacity per unit. You can also

improve the performance of HDDs by using more of them in a broker, whether by having multiple data directories or by setting up the drives in a redundant array of independent disks (RAID) configuration. Other factors, such as the specific drive technology (e.g., serial attached storage or serial ATA), as well as the quality of the drive controller, will affect throughput. Generally, observations show that HDD drives are typically more useful for clusters with very high storage needs but aren't accessed as often, while SSDs are better options if there is a very large number of client connections.

Disk Capacity

Capacity is the other side of the storage discussion. The amount of disk capacity that is needed is determined by how many messages need to be retained at any time. If the broker is expected to receive 1 TB of traffic each day, with 7 days of retention, then the broker will need a minimum of 7 TB of usable storage for log segments. You should also factor in at least 10% overhead for other files, in addition to any buffer that you wish to maintain for fluctuations in traffic or growth over time.

Storage capacity is one of the factors to consider when sizing a Kafka cluster and determining when to expand it. The total traffic for a cluster can be balanced across the cluster by having multiple partitions per topic, which will allow additional brokers to augment the available capacity if the density on a single broker will not suffice. The decision on how much disk capacity is needed will also be informed by the replication strategy chosen for the cluster (which is discussed in more detail in Chapter 7).

Memory

The normal mode of operation for a Kafka consumer is reading from the end of the partitions, where the consumer is caught up and lagging behind the producers very little, if at all. In this situation, the messages the consumer is reading are optimally stored in the system's page cache, resulting in faster reads than if the broker has to reread the messages from disk. Therefore, having more memory available to the system for page cache will improve the performance of consumer clients.

Kafka itself does not need much heap memory configured for the Java Virtual Machine (JVM). Even a broker that is handling 150,000 messages per second and a data rate of 200 megabits per second can run with a 5 GB heap. The rest of the system memory will be used by the page cache and will benefit Kafka by allowing the system to cache log segments in use. This is the main reason it is not recommended to have Kafka colocated on a system with any other significant application, as it will have to share the use of the page cache. This will decrease the consumer performance for Kafka.

Networking

The available network throughput will specify the maximum amount of traffic that Kafka can handle. This can be a governing factor, combined with disk storage, for cluster sizing. This is complicated by the inherent imbalance between inbound and outbound network usage that is created by Kafka's support for multiple consumers. A producer may write 1 MB per second for a given topic, but there could be any number of consumers that create a multiplier on the outbound network usage. Other operations, such as cluster replication (covered in Chapter 7) and mirroring (discussed in Chapter 10), will also increase requirements. Should the network interface become saturated, it is not uncommon for cluster replication to fall behind, which can leave the cluster in a vulnerable state. To prevent the network from being a major governing factor, it is recommended to run with at least 10 Gb NICs (Network Interface Cards). Older machines with 1 Gb NICs are easily saturated and aren't recommended.

CPU

Processing power is not as important as disk and memory until you begin to scale Kafka very large, but it will affect overall performance of the broker to some extent. Ideally, clients should compress messages to optimize network and disk usage. The Kafka broker must decompress all message batches, however, in order to validate the `checksum` of the individual messages and assign offsets. It then needs to recompress the message batch in order to store it on disk. This is where most of Kafka's requirement for processing power comes from. This should not be the primary factor in selecting hardware, however, unless clusters become very large with hundreds of nodes and millions of partitions in a single cluster. At that point, selecting more performant CPU can help reduce cluster sizes.

Kafka in the Cloud

In recent years, a more common installation for Kafka is within cloud computing environments, such as Microsoft Azure, Amazon's AWS, or Google Cloud Platform. There are many options to have Kafka set up in the cloud and managed for you via vendors like Confluent or even through Azure's own Kafka on HDInsight, but the following is some simple advice if you plan to manage your own Kafka clusters manually. In most cloud environments, you have a selection of many compute instances, each with a different combination of CPU, memory, IOPS, and disk. The various performance characteristics of Kafka must be prioritized in order to select the correct instance configuration to use.

Microsoft Azure

In Azure, you can manage the disks separately from the virtual machine (VM), so deciding your storage needs does not need to be related to the VM type selected. That being said, a good place to start on decisions is with the amount of data retention required, followed by the performance needed from the producers. If very low latency is necessary, I/O optimized instances utilizing premium SSD storage might be required. Otherwise, managed storage options (such as the Azure Managed Disks or the Azure Blob Storage) might be sufficient.

In real terms, experience in Azure shows that `Standard D16s v3` instance types are a good choice for smaller clusters and are performant enough for most use cases. To match high performant hardware and CPU needs, `D64s v4` instances have good performance that can scale for larger clusters. It is recommended to build out your cluster in an Azure availability set and balance partitions across Azure compute fault domains to ensure availability. Once you have a VM picked out, deciding on storage types can come next. It is highly recommended to use Azure Managed Disks rather than ephemeral disks. If a VM is moved, you run the risk of losing all the data on your Kafka broker. HDD Managed Disks are relatively inexpensive but do not have clearly defined SLAs from Microsoft on availability. Premium SSDs or Ultra SSD configurations are much more expensive but are much quicker and are well supported with 99.99% SLAs from Microsoft. Alternatively, using Microsoft Blob Storage is an option if you are not as latency sensitive.

Amazon Web Services

In AWS, if very low latency is necessary, I/O optimized instances that have local SSD storage might be required. Otherwise, ephemeral storage (such as the Amazon Elastic Block Store) might be sufficient.

A common choice in AWS is either the `m4` or `r3` instance types. The `m4` will allow for greater retention periods, but the throughput to the disk will be less because it is on elastic block storage. The `r3` instance will have much better throughput with local SSD drives, but those drives will limit the amount of data that can be retained. For the best of both worlds, it may be necessary to move up to either the `i2` or `d2` instance types, but they are significantly more expensive.

Configuring Kafka Clusters

A single Kafka broker works well for local development work, or for a proof-of-concept system, but there are significant benefits to having multiple brokers configured as a cluster, as shown in Figure 2-2. The biggest benefit is the ability to scale the load across multiple servers. A close second is using replication to guard against data loss due to single system failures. Replication will also allow for performing

maintenance work on Kafka or the underlying systems while still maintaining availability for clients. This section focuses on the steps to configure a Kafka basic cluster. Chapter 7 contains more information on replication of data and durability.

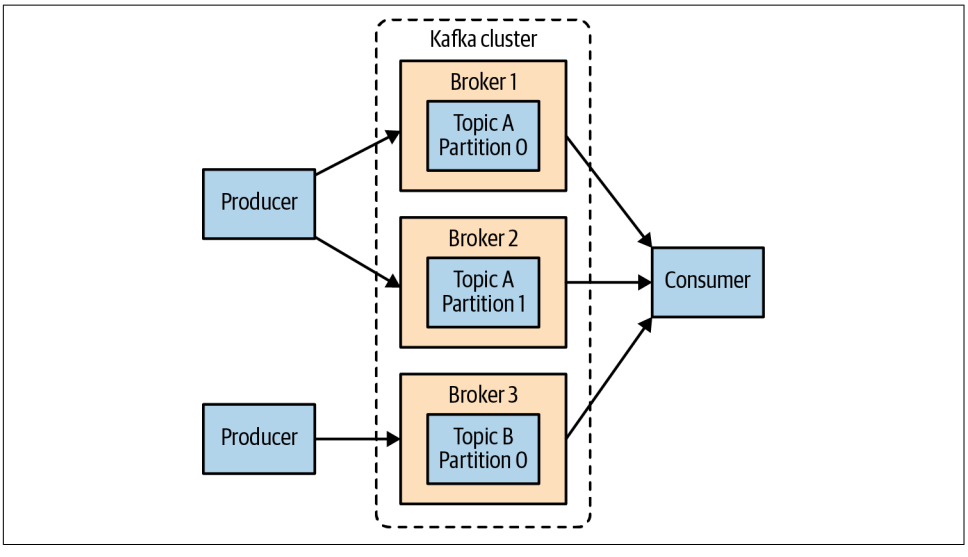

Figure 2-2. A simple Kafka cluster

How Many Brokers?

The appropriate size for a Kafka cluster is determined by several factors. Typically, the size of your cluster will be bound on the following key areas:

- Disk capacity
- Replica capacity per broker
- CPU capacity
- Network capacity

The first factor to consider is how much disk capacity is required for retaining messages and how much storage is available on a single broker. If the cluster is required to retain 10 TB of data and a single broker can store 2 TB, then the minimum cluster size is 5 brokers. In addition, increasing the replication factor will increase the storage requirements by at least 100%, depending on the replication factor setting chosen (see Chapter 7). Replicas in this case refer to the number of different brokers a single partition is copied to. This means that this same cluster, configured with a replication of 2, now needs to contain at least 10 brokers.

The other factor to consider is the capacity of the cluster to handle requests. This can exhibit through the other three bottlenecks mentioned earlier.

If you have a 10-broker Kafka cluster but have over 1 million replicas (i.e., 500,000 partitions with a replication factor of 2) in your cluster, each broker is taking on approximately 100,000 replicas in an evenly balanced scenario. This can lead to bottlenecks in the produce, consume, and controller queues. In the past, official recommendations have been to have no more than 4,000 partition replicas per broker and no more than 200,000 partition replicas per cluster. However, advances in cluster efficiency have allowed Kafka to scale much larger. Currently, in a well-configured environment, it is recommended to not have more than 14,000 partition replicas per broker and 1 million *replicas* per cluster.

As previously mentioned in this chapter, CPU usually is not a major bottleneck for most use cases, but it can be if there is an excessive amount of client connections and requests on a broker. Keeping an eye on overall CPU usage based on how many unique clients and consumer groups there are, and expanding to meet those needs, can help to ensure better overall performance in large clusters. Speaking to network capacity, it is important to keep in mind the capacity of the network interfaces and whether they can handle the client traffic if there are multiple consumers of the data or if the traffic is not consistent over the retention period of the data (e.g., bursts of traffic during peak times). If the network interface on a single broker is used to 80% capacity at peak, and there are two consumers of that data, the consumers will not be able to keep up with peak traffic unless there are two brokers. If replication is being used in the cluster, this is an additional consumer of the data that must be taken into account. You may also want to scale out to more brokers in a cluster in order to handle performance concerns caused by lesser disk throughput or system memory available.

Broker Configuration

There are only two requirements in the broker configuration to allow multiple Kafka brokers to join a single cluster. The first is that all brokers must have the same configuration for the `zookeeper.connect` parameter. This specifies the ZooKeeper ensemble and path where the cluster stores metadata. The second requirement is that all brokers in the cluster must have a unique value for the `broker.id` parameter. If two brokers attempt to join the same cluster with the same `broker.id`, the second broker will log an error and fail to start. There are other configuration parameters used when running a cluster—specifically, parameters that control replication, which are covered in later chapters.

OS Tuning

While most Linux distributions have an out-of-the-box configuration for the kernel-tuning parameters that will work fairly well for most applications, there are a few changes that can be made for a Kafka broker that will improve performance. These

primarily revolve around the virtual memory and networking subsystems, as well as specific concerns for the disk mount point used for storing log segments. These parameters are typically configured in the */etc/sysctl.conf* file, but you should refer to your Linux distribution documentation for specific details regarding how to adjust the kernel configuration.

Virtual memory

In general, the Linux virtual memory system will automatically adjust itself for the system workload. We can make some adjustments to how swap space is handled, as well as to dirty memory pages, to tune these for Kafka's workload.

As with most applications, specifically ones where throughput is a concern, it is best to avoid swapping at (almost) all costs. The cost incurred by having pages of memory swapped to disk will show up as a noticeable impact on all aspects of performance in Kafka. In addition, Kafka makes heavy use of the system page cache, and if the VM system is swapping to disk, there is not enough memory being allocated to page cache.

One way to avoid swapping is simply not to configure any swap space at all. Having swap is not a requirement, but it does provide a safety net if something catastrophic happens on the system. Having swap can prevent the OS from abruptly killing a process due to an out-of-memory condition. For this reason, the recommendation is to set the `vm.swappiness` parameter to a very low value, such as 1. The parameter is a percentage of how likely the VM subsystem is to use swap space rather than dropping pages from the page cache. It is preferable to reduce the amount of memory available for the page cache rather than utilize any amount of swap memory.

Why Not Set Swappiness to Zero?

Previously, the recommendation for `vm.swappiness` was always to set it to 0. This value used to mean "do not swap unless there is an out-of-memory condition." However, the meaning of this value changed as of Linux kernel version 3.5-rc1, and that change was backported into many distributions, including Red Hat Enterprise Linux kernels as of version 2.6.32-303. This changed the meaning of the value 0 to "never swap under any circumstances." This is why a value of 1 is now recommended.

There is also a benefit to adjusting how the kernel handles dirty pages that must be flushed to disk. Kafka relies on disk I/O performance to provide good response times to producers. This is also the reason that the log segments are usually put on a fast disk, whether that is an individual disk with a fast response time (e.g., SSD) or a disk subsystem with significant NVRAM for caching (e.g., RAID). The result is that the number of dirty pages that are allowed, before the flush background process starts

writing them to disk, can be reduced. Do this by setting the `vm.dirty_background_ratio` value lower than the default of 10. The value is a percentage of the total amount of system memory, and setting this value to 5 is appropriate in many situations. This setting should not be set to zero, however, as that would cause the kernel to continually flush pages, which would then eliminate the ability of the kernel to buffer disk writes against temporary spikes in the underlying device performance.

The total number of dirty pages allowed before the kernel forces synchronous operations to flush them to disk can also be increased by changing the value of `vm.dirty_ratio` to above the default of 20 (also a percentage of total system memory). There is a wide range of possible values for this setting, but between 60 and 80 is a reasonable number. This setting does introduce a small amount of risk, both in regard to the amount of unflushed disk activity as well as the potential for long I/O pauses if synchronous flushes are forced. If a higher setting for `vm.dirty_ratio` is chosen, it is highly recommended that replication be used in the Kafka cluster to guard against system failures.

When choosing values for these parameters, it is wise to review the number of dirty pages over time while the Kafka cluster is running under load, whether in production or simulated. The current number of dirty pages can be determined by checking the */proc/vmstat* file:

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 21845
nr_writeback 0
nr_writeback_temp 0
nr_dirty_threshold 32715981
nr_dirty_background_threshold 2726331
#
```

Kafka uses file descriptors for log segments and open connections. If a broker has a lot of partitions, then that broker needs at least *(number_of_partitions)* × *(partition_size/segment_size)* to track all the log segments in addition to the number of connections the broker makes. As such, it is recommended to update the `vm.max_map_count` to a very large number based on the above calculation. Depending on the environment, changing this value to 400,000 or 600,000 has generally been successful. It is also recommended to set `vm.overcommit_memory` to 0. Setting the default value of 0 indicates that the kernel determines the amount of free memory from an application. If the property is set to a value other than zero, it could lead the operating system to grab too much memory, depriving memory for Kafka to operate optimally. This is common for applications with high ingestion rates.

Disk

Outside of selecting the disk device hardware, as well as the configuration of RAID if it is used, the choice of filesystem for this disk can have the next largest impact on performance. There are many different filesystems available, but the most common choices for local filesystems are either Ext4 (fourth extended filesystem) or Extents File System (XFS). XFS has become the default filesystem for many Linux distributions, and this is for good reason: it outperforms Ext4 for most workloads with minimal tuning required. Ext4 can perform well but requires using tuning parameters that are considered less safe. This includes setting the commit interval to a longer time than the default of five to force less frequent flushes. Ext4 also introduced delayed allocation of blocks, which brings with it a greater chance of data loss and filesystem corruption in case of a system failure. The XFS filesystem also uses a delayed allocation algorithm, but it is generally safer than the one used by Ext4. XFS also has better performance for Kafka's workload without requiring tuning beyond the automatic tuning performed by the filesystem. It is also more efficient when batching disk writes, all of which combine to give better overall I/O throughput.

Regardless of which filesystem is chosen for the mount that holds the log segments, it is advisable to set the `noatime` mount option for the mount point. File metadata contains three timestamps: creation time (`ctime`), last modified time (`mtime`), and last access time (`atime`). By default, the `atime` is updated every time a file is read. This generates a large number of disk writes. The `atime` attribute is generally considered to be of little use, unless an application needs to know if a file has been accessed since it was last modified (in which case the `relatime` option can be used). The `atime` is not used by Kafka at all, so disabling it is safe. Setting `noatime` on the mount will prevent these timestamp updates from happening but will not affect the proper handling of the `ctime` and `mtime` attributes. Using the option `largeio` can also help improve efficiency for Kafka for when there are larger disk writes.

Networking

Adjusting the default tuning of the Linux networking stack is common for any application that generates a high amount of network traffic, as the kernel is not tuned by default for large, high-speed data transfers. In fact, the recommended changes for Kafka are the same as those suggested for most web servers and other networking applications. The first adjustment is to change the default and maximum amount of memory allocated for the send and receive buffers for each socket. This will significantly increase performance for large transfers. The relevant parameters for the send and receive buffer default size per socket are `net.core.wmem_default` and `net.core.rmem_default`, and a reasonable setting for these parameters is 131072, or 128 KiB. The parameters for the send and receive buffer maximum sizes are `net.core.wmem_max` and `net.core.rmem_max`, and a reasonable setting is 2097152, or

2 MiB. Keep in mind that the maximum size does not indicate that every socket will have this much buffer space allocated; it only allows up to that much if needed.

In addition to the socket settings, the send and receive buffer sizes for TCP sockets must be set separately using the `net.ipv4.tcp_wmem` and `net.ipv4.tcp_rmem` parameters. These are set using three space-separated integers that specify the minimum, default, and maximum sizes, respectively. The maximum size cannot be larger than the values specified for all sockets using `net.core.wmem_max` and `net.core.rmem_max`. An example setting for each of these parameters is "4096 65536 2048000," which is a 4 KiB minimum, 64 KiB default, and 2 MiB maximum buffer. Based on the actual workload of your Kafka brokers, you may want to increase the maximum sizes to allow for greater buffering of the network connections.

There are several other network tuning parameters that are useful to set. Enabling TCP window scaling by setting `net.ipv4.tcp_window_scaling` to 1 will allow clients to transfer data more efficiently, and allow that data to be buffered on the broker side. Increasing the value of `net.ipv4.tcp_max_syn_backlog` above the default of 1024 will allow a greater number of simultaneous connections to be accepted. Increasing the value of `net.core.netdev_max_backlog` to greater than the default of 1000 can assist with bursts of network traffic, specifically when using multigigabit network connection speeds, by allowing more packets to be queued for the kernel to process them.

Production Concerns

Once you are ready to move your Kafka environment out of testing and into your production operations, there are a few more things to think about that will assist with setting up a reliable messaging service.

Garbage Collector Options

Tuning the Java garbage-collection options for an application has always been something of an art, requiring detailed information about how the application uses memory and a significant amount of observation and trial and error. Thankfully, this has changed with Java 7 and the introduction of the Garbage-First garbage collector (G1GC). While G1GC was considered unstable initially, it saw marked improvement in JDK8 and JDK11. It is now recommended for Kafka to use G1GC as the default garbage collector. G1GC is designed to automatically adjust to different workloads and provide consistent pause times for garbage collection over the lifetime of the application. It also handles large heap sizes with ease by segmenting the heap into smaller zones and not collecting over the entire heap in each pause.

G1GC does all of this with a minimal amount of configuration in normal operation. There are two configuration options for G1GC used to adjust its performance:

`MaxGCPauseMillis`
: This option specifies the preferred pause time for each garbage-collection cycle. It is not a fixed maximum—G1GC can and will exceed this time if required. This value defaults to 200 milliseconds. This means that G1GC will attempt to schedule the frequency of garbage collector cycles, as well as the number of zones that are collected in each cycle, such that each cycle will take approximately 200 ms.

`InitiatingHeapOccupancyPercent`
: This option specifies the percentage of the total heap that may be in use before G1GC will start a collection cycle. The default value is 45. This means that G1GC will not start a collection cycle until after 45% of the heap is in use. This includes both the new (Eden) and old zone usage, in total.

The Kafka broker is fairly efficient with the way it utilizes heap memory and creates garbage objects, so it is possible to set these options lower. The garbage collector tuning options provided in this section have been found to be appropriate for a server with 64 GB of memory, running Kafka in a 5 GB heap. For `MaxGCPauseMillis`, this broker can be configured with a value of 20 ms. The value for `InitiatingHeapOccupancyPercent` is set to 35, which causes garbage collection to run slightly earlier than with the default value.

Kafka was originally released before the G1GC collector was available and considered stable. Therefore, Kafka defaults to using concurrent mark and sweep garbage collection to ensure compatibility with all JVMs. New best practice is to use G1GC for anything for Java 1.8 and later. The change is easy to make via environment variables. Using the `start` command from earlier in the chapter, modify it as follows:

```
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -Xmx6g -Xms6g
-XX:MetaspaceSize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:G1HeapRegionSize=16M -XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80 -XX:+ExplicitGCInvokesConcurrent"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Datacenter Layout

For testing and development environments, the physical location of the Kafka brokers within a datacenter is not as much of a concern, as there is not as severe an impact if the cluster is partially or completely unavailable for short periods of time. However, when serving production traffic, downtime usually means dollars lost, whether through loss of services to users or loss of telemetry on what the users are doing. This

is when it becomes critical to configure replication within the Kafka cluster (see Chapter 7), which is also when it is important to consider the physical location of brokers in their racks in the datacenter. A datacenter environment that has a concept of fault zones is preferable. If not addressed prior to deploying Kafka, expensive maintenance to move servers around may be needed.

Kafka can assign new partitions to brokers in a rack-aware manner, making sure that replicas for a single partition do not share a rack. To do this, the `broker.rack` configuration for each broker must be set properly. This config can be set to the fault domain in cloud environments as well for similar reasons. However, this only applies to partitions that are newly created. The Kafka cluster does not monitor for partitions that are no longer rack aware (for example, as a result of a partition reassignment), nor does it automatically correct this situation. It is recommend to use tools that keep your cluster balanced properly to maintain rack awareness, such as Cruise Control (see Appendix B). Configuring this properly will help to ensure continued rack awareness over time.

Overall, the best practice is to have each Kafka broker in a cluster installed in a different rack, or at the very least not share single points of failure for infrastructure services such as power and network. This typically means at least deploying the servers that will run brokers with dual power connections (to two different circuits) and dual network switches (with a bonded interface on the servers themselves to failover seamlessly). Even with dual connections, there is a benefit to having brokers in completely separate racks. From time to time, it may be necessary to perform physical maintenance on a rack or cabinet that requires it to be offline (such as moving servers around or rewiring power connections).

Colocating Applications on ZooKeeper

Kafka utilizes ZooKeeper for storing metadata information about the brokers, topics, and partitions. Writes to ZooKeeper are only performed on changes to the membership of consumer groups or on changes to the Kafka cluster itself. This amount of traffic is generally minimal, and it does not justify the use of a dedicated ZooKeeper ensemble for a single Kafka cluster. In fact, many deployments will use a single ZooKeeper ensemble for multiple Kafka clusters (using a chroot ZooKeeper path for each cluster, as described earlier in this chapter).

Kafka Consumers, Tooling, ZooKeeper, and You

As time goes on, dependency on ZooKeeper is shrinking. In version 2.8.0, Kafka is introducing an early-access look at a completely ZooKeeper-less Kafka, but it is still not production ready. However, we can still see this reduced reliance on ZooKeeper in versions leading up to this. For example, in older versions of Kafka, consumers (in addition to the brokers) utilized ZooKeeper to directly store information about the composition of the consumer group and what topics it was consuming, and to periodically commit offsets for each partition being consumed (to enable failover between consumers in the group). With version 0.9.0.0, the consumer interface was changed, allowing this to be managed directly with the Kafka brokers. In each 2.x release of Kafka, we see additional steps to removing ZooKeeper from other required paths of Kafka. Administration tools now connect directly to the cluster and have deprecated the need to connect to ZooKeeper directly for operations such as topic creations, dynamic configuration changes, etc. As such, many of the command-line tools that previously used the `--zookeeper` flags have been updated to use the `--bootstrap-server` option. The `--zookeeper` options can still be used but have been deprecated and will be removed in the future when Kafka is no longer required to connect to ZooKeeper to create, manage, or consume from topics.

However, there is a concern with consumers and ZooKeeper under certain configurations. While the use of ZooKeeper for such purposes is deprecated, consumers have a configurable choice to use either ZooKeeper or Kafka for committing offsets, and they can also configure the interval between commits. If the consumer uses ZooKeeper for offsets, each consumer will perform a ZooKeeper write at every interval for every partition it consumes. A reasonable interval for offset commits is 1 minute, as this is the period of time over which a consumer group will read duplicate messages in the case of a consumer failure. These commits can be a significant amount of ZooKeeper traffic, especially in a cluster with many consumers, and will need to be taken into account. It may be necessary to use a longer commit interval if the ZooKeeper ensemble is not able to handle the traffic. However, it is recommended that consumers using the latest Kafka libraries use Kafka for committing offsets, removing the dependency on ZooKeeper.

Outside of using a single ensemble for multiple Kafka clusters, it is not recommended to share the ensemble with other applications, if it can be avoided. Kafka is sensitive to ZooKeeper latency and timeouts, and an interruption in communications with the ensemble will cause the brokers to behave unpredictably. This can easily cause multiple brokers to go offline at the same time should they lose ZooKeeper connections, which will result in offline partitions. It also puts stress on the cluster controller,

which can show up as subtle errors long after the interruption has passed, such as when trying to perform a controlled shutdown of a broker. Other applications that can put stress on the ZooKeeper ensemble, either through heavy usage or improper operations, should be segregated to their own ensemble.

Summary

In this chapter we learned how to get Apache Kafka up and running. We also covered picking the right hardware for your brokers, and specific concerns around getting set up in a production environment. Now that you have a Kafka cluster, we will walk through the basics of Kafka client applications. The next two chapters will cover how to create clients for both producing messages to Kafka (Chapter 3) as well as consuming those messages out again (Chapter 4).

CHAPTER 3

Kafka Producers: Writing Messages to Kafka

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by creating a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles.

For example, in a credit card transaction processing system, there will be a client application, perhaps an online store, responsible for sending each transaction to Kafka immediately when a payment is made. Another application is responsible for immediately checking this transaction against a rules engine and determining whether the transaction is approved or denied. The approve/deny response can then be written back to Kafka, and the response can propagate back to the online store where the transaction was initiated. A third application can read both transactions and the approval status from Kafka and store them in a database where analysts can later review the decisions and perhaps improve the rules engine.

Apache Kafka ships with built-in client APIs that developers can use when developing applications that interact with Kafka.

In this chapter we will learn how to use the Kafka producer, starting with an overview of its design and components. We will show how to create `KafkaProducer` and `ProducerRecord` objects, how to send records to Kafka, and how to handle the errors that Kafka may return. We'll then review the most important configuration options used to control the producer behavior. We'll conclude with a deeper look at how to use different partitioning methods and serializers, and how to write your own serializers and partitioners.

In Chapter 4, we will look at Kafka's consumer client and reading data from Kafka.

Third-Party Clients

In addition to the built-in clients, Kafka has a binary wire protocol. This means that it is possible for applications to read messages from Kafka or write messages to Kafka simply by sending the correct byte sequences to Kafka's network port. There are multiple clients that implement Kafka's wire protocol in different programming languages, giving simple ways to use Kafka not just in Java applications but also in languages like C++, Python, Go, and many more. Those clients are not part of the Apache Kafka project, but a list of non-Java clients is maintained in the project wiki. The wire protocol and the external clients are outside the scope of the chapter.

Producer Overview

There are many reasons an application might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously with other applications, buffering information before writing to a database, and much more.

Those diverse use cases also imply diverse requirements: is every message critical, or can we tolerate loss of messages? Are we OK with accidentally duplicating messages? Are there any strict latency or throughput requirements we need to support?

In the credit card transaction processing example we introduced earlier, we can see that it is critical to never lose a single message or duplicate any messages. Latency should be low, but latencies up to 500 ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

A different use case might be to store click information from a website. In that case, some message loss or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience. In other words, we don't mind if it takes a few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicks on a link. Throughput will depend on the level of activity we anticipate on our website.

The different requirements will influence the way you use the producer API to write messages to Kafka and the configuration you use.

While the producer API is very simple, there is a bit more that goes on under the hood of the producer when we send data. Figure 3-1 shows the main steps involved in sending data to Kafka.

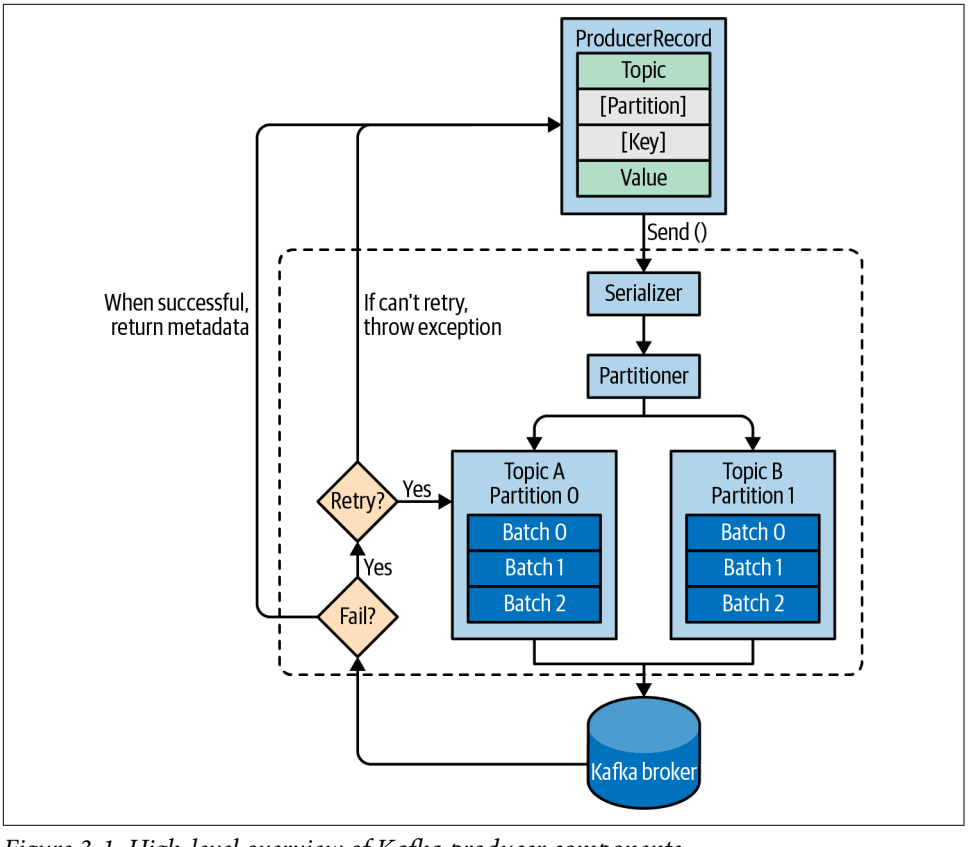

Figure 3-1. High-level overview of Kafka producer components

We start producing messages to Kafka by creating a `ProducerRecord`, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key, a partition, a timestamp, and/or a collection of headers. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to byte arrays so they can be sent over the network.

Next, if we didn't explicitly specify a partition, the data is sent to a partitioner. The partitioner will choose a partition for us, usually based on the `ProducerRecord` key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a `RecordMetadata` object with the topic, partition, and the offset of the record within the partition. If the broker failed

to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

Constructing a Kafka Producer

The first step in writing messages to Kafka is to create a producer object with the properties you want to pass to the producer. A Kafka producer has three mandatory properties:

`bootstrap.servers`
: List of `host:port` pairs of brokers that the producer will use to establish initial connection to the Kafka cluster. This list doesn't need to include all brokers, since the producer will get more information after the initial connection. But it is recommended to include at least two, so in case one broker goes down, the producer will still be able to connect to the cluster.

`key.serializer`
: Name of a class that will be used to serialize the keys of the records we will produce to Kafka. Kafka brokers expect byte arrays as keys and values of messages. However, the producer interface allows, using parameterized types, any Java object to be sent as a key and value. This makes for very readable code, but it also means that the producer has to know how to convert these objects to byte arrays. `key.serializer` should be set to a name of a class that implements the `org.apache.kafka.common.serialization.Serializer` interface. The producer will use this class to serialize the key object to a byte array. The Kafka client package includes `ByteArraySerializer` (which doesn't do much), `StringSerializer`, `IntegerSerializer`, and much more, so if you use common types, there is no need to implement your own serializers. Setting `key.serializer` is required even if you intend to send only values, but you can use the `Void` type for the key and the `VoidSerializer`.

`value.serializer`
: Name of a class that will be used to serialize the values of the records we will produce to Kafka. The same way you set `key.serializer` to a name of a class that will serialize the message key object to a byte array, you set `value.serializer` to a class that will serialize the message value object.

The following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
```

```
        "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

❶ We start with a `Properties` object.

❷ Since we plan on using strings for message key and value, we use the built-in `StringSerializer`.

❸ Here we create a new producer by setting the appropriate key and value types and passing the `Properties` object.

With such a simple interface, it is clear that most of the control over producer behavior is done by setting the correct configuration properties. Apache Kafka documentation covers all the configuration options, and we will go over the important ones later in this chapter.

Once we instantiate a producer, it is time to start sending messages. There are three primary methods of sending messages:

Fire-and-forget
: We send a message to the server and don't really care if it arrives successfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, in case of nonretriable errors or timeout, messages will get lost and the application will not get any information or exceptions about this.

Synchronous send
: Technically, Kafka producer is always asynchronous—we send a message and the `send()` method returns a `Future` object. However, we use `get()` to wait on the `Future` and see if the `send()` was successful or not before sending the next record.

Asynchronous send
: We call the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.

In the examples that follow, we will see how to send messages using these methods and how to handle the different types of errors that might occur.

While all the examples in this chapter are single threaded, a producer object can be used by multiple threads to send messages.

Sending a Message to Kafka

The simplest way to send a message is as follows:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products",
        "France"); ❶
try {
    producer.send(record); ❷
} catch (Exception e) {
    e.printStackTrace(); ❸
}
```

❶ The producer accepts `ProducerRecord` objects, so we start by creating one. `ProducerRecord` has multiple constructors, which we will discuss later. Here we use one that requires the name of the topic we are sending data to, which is always a string, and the key and value we are sending to Kafka, which in this case are also strings. The types of the key and value must match our `key serializer` and `value serializer` objects.

❷ We use the producer object `send()` method to send the `ProducerRecord`. As we've seen in the producer architecture diagram in Figure 3-1, the message will be placed in a buffer and will be sent to the broker in a separate thread. The `send()` method returns a Java `Future` object with `RecordMetadata`, but since we simply ignore the returned value, we have no way of knowing whether the message was sent successfully or not. This method of sending messages can be used when dropping a message silently is acceptable. This is not typically the case in production applications.

❸ While we ignore errors that may occur while sending messages to Kafka brokers or in the brokers themselves, we may still get an exception if the producer encountered errors before sending the message to Kafka. Those can be, for example, a `SerializationException` when it fails to serialize the message, a `BufferExhaustedException` or `TimeoutException` if the buffer is full, or an `InterruptException` if the sending thread was interrupted.

Sending a Message Synchronously

Sending a message synchronously is simple but still allows the producer to catch exceptions when Kafka responds to the produce request with an error, or when send retries were exhausted. The main trade-off involved is performance. Depending on how busy the Kafka cluster is, brokers can take anywhere from 2 ms to a few seconds to respond to produce requests. If you send messages synchronously, the sending thread will spend this time waiting and doing nothing else, not even sending additional messages. This leads to very poor performance, and as a result,

synchronous sends are usually not used in production applications (but are very common in code examples).

The simplest way to send a message synchronously is as follows:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
    producer.send(record).get(); ❶
} catch (Exception e) {
    e.printStackTrace(); ❷
}
```

❶ Here, we are using `Future.get()` to wait for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a `RecordMetadata` object that we can use to retrieve the offset the message was written to and other metadata.

❷ If there were any errors before or while sending the record to Kafka, we will encounter an exception. In this case, we just print any exception we ran into.

`KafkaProducer` has two types of errors. *Retriable* errors are those that can be resolved by sending the message again. For example, a connection error can be resolved because the connection may get reestablished. A "not leader for partition" error can be resolved when a new leader is elected for the partition and the client metadata is refreshed. `KafkaProducer` can be configured to retry those errors automatically, so the application code will get retriable exceptions only when the number of retries was exhausted and the error was not resolved. Some errors will not be resolved by retrying—for example, "Message size too large." In those cases, `KafkaProducer` will not attempt a retry and will return the exception immediately.

Sending a Message Asynchronously

Suppose the network round-trip time between our application and the Kafka cluster is 10 ms. If we wait for a reply after sending each message, sending 100 messages will take around 1 second. On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. In most cases, we really don't need a reply—Kafka sends back the topic, partition, and offset of the record after it was written, which is usually not required by the sending app. On the other hand, we do need to know when we failed to send a message completely so we can throw an exception, log an error, or perhaps write the message to an "errors" file for later analysis.

To send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. Here is an example of how we use a callback:

```
private class DemoProducerCallback implements Callback { ❶
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA"); ❸
producer.send(record, new DemoProducerCallback()); ❹
```

❶ To use callbacks, you need a class that implements the `org.apache.kafka.clients.producer.Callback` interface, which has a single function—`onCompletion()`.

❷ If Kafka returned an error, `onCompletion()` will have a nonnull exception. Here we "handle" it by printing, but production code will probably have more robust error handling functions.

❸ The records are the same as before.

❹ And we pass a `Callback` object along when sending the record.

The callbacks execute in the producer's main thread. This guarantees that when we send two messages to the same partition one after another, their callbacks will be executed in the same order that we sent them. But it also means that the callback should be reasonably fast to avoid delaying the producer and preventing other messages from being sent. It is not recommended to perform a blocking operation within the callback. Instead, you should use another thread to perform any blocking operation concurrently.

Configuring Producers

So far we've seen very few configuration parameters for the producers—just the mandatory `bootstrap.servers` URI and serializers.

The producer has a large number of configuration parameters that are documented in Apache Kafka documentation, and many have reasonable defaults, so there is no reason to tinker with every single parameter. However, some of the parameters have a significant impact on memory use, performance, and reliability of the producers. We will review those here.

client.id

`client.id` is a logical identifier for the client and the application it is used in. This can be any string and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics and for quotas. Choosing a good client name will make troubleshooting much easier—it is the difference between "We are seeing a high rate of authentication failures from IP 104.27.155.134" and "Looks like the Order Validation service is failing to authenticate—can you ask Laura to take a look?"

acks

The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful. By default, Kafka will respond that the record was written successfully after the leader received the record (release 3.0 of Apache Kafka is expected to change this default). This option has a significant impact on the durability of written messages, and depending on your use case, the default may not be the best choice. Chapter 7 discusses Kafka's reliability guarantees in depth, but for now let's review the three allowed values for the `acks` parameter:

`acks=0`
: The producer will not wait for a reply from the broker before assuming the message was sent successfully. This means that if something goes wrong and the broker does not receive the message, the producer will not know about it, and the message will be lost. However, because the producer is not waiting for any response from the server, it can send messages as fast as the network will support, so this setting can be used to achieve very high throughput.

`acks=1`
: The producer will receive a success response from the broker the moment the leader replica receives the message. If the message can't be written to the leader (e.g., if the leader crashed and a new leader was not elected yet), the producer will receive an error response and can retry sending the message, avoiding potential loss of data. The message can still get lost if the leader crashes and the latest messages were not yet replicated to the new leader.

`acks=all`
: The producer will receive a success response from the broker once all in sync replicas receive the message. This is the safest mode since you can make sure more than one broker has the message and that the message will survive even in case of a crash (more information on this in Chapter 6). However, the latency we discussed in the `acks=1` case will be even higher, since we will be waiting for more than just one broker to receive the message.

You will see that with lower and less reliable `acks` configuration, the producer will be able to send records faster. This means that you trade off reliability for *producer latency*. However, *end-to-end latency* is measured from the time a record was produced until it is available for consumers to read and is identical for all three options. The reason is that, in order to maintain consistency, Kafka will not allow consumers to read records until they are written to all in sync replicas. Therefore, if you care about end-to-end latency, rather than just the producer latency, there is no trade-off to make: you will get the same end-to-end latency if you choose the most reliable option.

Message Delivery Time

The producer has multiple configuration parameters that interact to control one of the behaviors that are of most interest to developers: how long will it take until a call to `send()` will succeed or fail. This is the time we are willing to spend until Kafka responds successfully, or until we are willing to give up and admit defeat.

The configurations and their behaviors were modified several times over the years. We will describe here the latest implementation, introduced in Apache Kafka 2.1.

Since Apache Kafka 2.1, we divide the time spent sending a `ProduceRecord` into two time intervals that are handled separately:

- Time until an async call to `send()` returns. During this interval, the thread that called `send()` will be blocked.
- From the time an async call to `send()` returned successfully until the callback is triggered (with success or failure). This is the same as from the point a `ProduceRecord` was placed in a batch for sending until Kafka responds with success, non-retriable failure, or we run out of time allocated for sending.

If you use `send()` synchronously, the sending thread will block for both time intervals continuously, and you won't be able to tell how much time was spent in each. We'll discuss the common and recommended case, where `send()` is used asynchronously, with a callback.

The flow of data within the producer and how the different configuration parameters affect each other can be summarized in Figure 3-2.[1]

1 Image contributed to the Apache Kafka project by Sumant Tambe under the ASLv2 license terms.

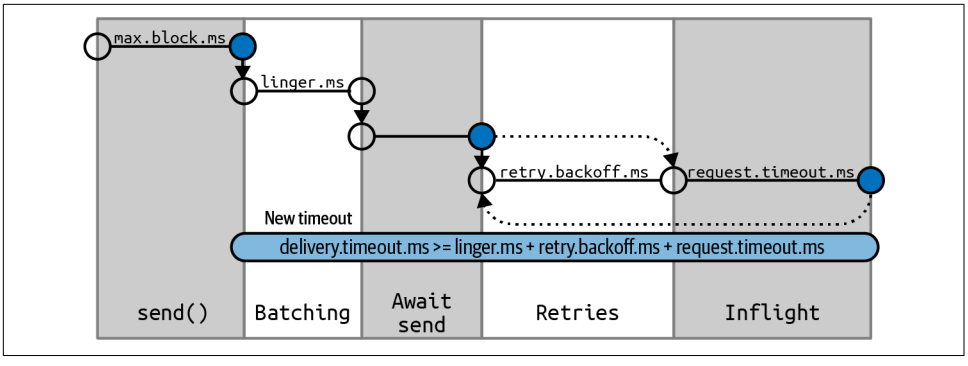

Figure 3-2. Sequence diagram of delivery time breakdown inside Kafka producer

We'll go through the different configuration parameters used to control the time spent waiting in these two intervals and how they interact.

max.block.ms

This parameter controls how long the producer may block when calling `send()` and when explicitly requesting metadata via `partitionsFor()`. Those methods may block when the producer's send buffer is full or when metadata is not available. When `max.block.ms` is reached, a timeout exception is thrown.

delivery.timeout.ms

This configuration will limit the amount of time spent from the point a record is ready for sending (`send()` returned successfully and the record is placed in a batch) until either the broker responds or the client gives up, including time spent on retries. As you can see in Figure 3-2, this time should be greater than `linger.ms` and `request.timeout.ms`. If you try to create a producer with an inconsistent timeout configuration, you will get an exception. Messages can be successfully sent much faster than `delivery.timeout.ms`, and typically will.

If the producer exceeds `delivery.timeout.ms` while retrying, the callback will be called with the exception that corresponds to the error that the broker returned before retrying. If `delivery.timeout.ms` is exceeded while the record batch was still waiting to be sent, the callback will be called with a timeout exception.

You can configure the delivery timeout to the maximum time you'll want to wait for a message to be sent, typically a few minutes, and then leave the default number of retries (virtually infinite). With this configuration, the producer will keep retrying for as long as it has time to keep trying (or until it succeeds). This is a much more reasonable way to think about retries. Our normal process for tuning retries is: "In case of a broker crash, it typically takes leader election 30 seconds to complete, so let's keep retrying for 120 seconds just to be on the safe side." Instead of converting this mental dialog to number of retries and time between retries, you just configure `deliver.timeout.ms` to 120.

request.timeout.ms

This parameter controls how long the producer will wait for a reply from the server when sending data. Note that this is the time spent waiting on each producer request before giving up; it does not include retries, time spent before sending, and so on. If the timeout is reached without reply, the producer will either retry sending or complete the callback with a `TimeoutException`.

retries and retry.backoff.ms

When the producer receives an error message from the server, the error could be transient (e.g., a lack of leader for a partition). In this case, the value of the `retries` parameter will control how many times the producer will retry sending the message before giving up and notifying the client of an issue. By default, the producer will wait 100 ms between retries, but you can control this using the `retry.backoff.ms` parameter.

We recommend against using these parameters in the current version of Kafka. Instead, test how long it takes to recover from a crashed broker (i.e., how long until all partitions get new leaders), and set `delivery.timeout.ms` such that the total amount of time spent retrying will be longer than the time it takes the Kafka cluster to recover from the crash—otherwise, the producer will give up too soon.

Not all errors will be retried by the producer. Some errors are not transient and will not cause retries (e.g., "message too large" error). In general, because the producer handles retries for you, there is no point in handling retries within your own application logic. You will want to focus your efforts on handling nonretriable errors or cases where retry attempts were exhausted.

If you want to completely disable retries, setting `retries=0` is the only way to do so.

linger.ms

`linger.ms` controls the amount of time to wait for additional messages before sending the current batch. `KafkaProducer` sends a batch of messages either when the current batch is full or when the `linger.ms` limit is reached. By default, the producer will send messages as soon as there is a sender thread available to send them, even if there's just one message in the batch. By setting `linger.ms` higher than 0, we instruct the producer to wait a few milliseconds to add additional messages to the batch before sending it to the brokers. This increases latency a little and significantly increases throughput—the overhead per message is much lower, and compression, if enabled, is much better.

buffer.memory

This config sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers. If messages are sent by the application faster than they can be delivered to the server, the producer may run out of space, and additional `send()` calls will block for `max.block.ms` and wait for space to free up before throwing an exception. Note that unlike most producer exceptions, this timeout is thrown by `send()` and not by the resulting `Future`.

compression.type

By default, messages are sent uncompressed. This parameter can be set to `snappy`, `gzip`, `lz4`, or `zstd`, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers. Snappy compression was invented by Google to provide decent compression ratios with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. Gzip compression will typically use more CPU and time but results in better compression ratios, so it is recommended in cases where network bandwidth is more restricted. By enabling compression, you reduce network utilization and storage, which is often a bottleneck when sending messages to Kafka.

batch.size

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes (not messages!) that will be used for each batch. When the batch is full, all the messages in the batch will be sent. However, this does not mean that the producer will wait for the batch to become full. The producer will send half-full batches and even batches with just a single message in them. Therefore, setting the batch size too large will not cause delays in sending messages; it will just use more memory for the batches. Setting the batch

size too small will add some overhead because the producer will need to send messages more frequently.

max.in.flight.requests.per.connection

This controls how many message batches the producer will send to the server without receiving responses. Higher settings can increase memory usage while improving throughput. Apache's wiki experiments show that in a single-DC environment, the throughput is maximized with only 2 in-flight requests; however, the default value is 5 and shows similar performance.

Ordering Guarantees

Apache Kafka preserves the order of messages within a partition. This means that if messages are sent from the producer in a specific order, the broker will write them to a partition in that order and all consumers will read them in that order. For some use cases, order is very important. There is a big difference between depositing $100 in an account and later withdrawing it, and the other way around! However, some use cases are less sensitive.

Setting the `retries` parameter to nonzero and the `max.in.flight.requests.per.connection` to more than 1 means that it is possible that the broker will fail to write the first batch of messages, succeed in writing the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.

Since we want at least two in-flight requests for performance reasons, and a high number of retries for reliability reasons, the best solution is to set `enable.idempotence=true`. This guarantees message ordering with up to five in-flight requests and also guarantees that retries will not introduce duplicates. Chapter 8 discusses the idempotent producer in depth.

max.request.size

This setting controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For example, with a default maximum request size of 1 MB, the largest message you can send is 1 MB, or the producer can batch 1,024 messages of size 1 KB each into one request. In addition, the broker has its own limit on the size of the largest message it will accept (`message.max.bytes`). It is usually a good idea to have these configurations match, so the producer will not attempt to send messages of a size that will be rejected by the broker.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to –1, the OS defaults will be used. It is a good idea to increase these when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

enable.idempotence

Starting in version 0.11, Kafka supports *exactly once* semantics. Exactly once is a fairly large topic, and we'll dedicate an entire chapter to it, but idempotent producer is a simple and highly beneficial part of it.

Suppose you configure your producer to maximize reliability: `acks=all` and a decently large `delivery.timeout.ms` to allow sufficient retries. These make sure each message will be written to Kafka at least once. In some cases, this means that messages will be written to Kafka more than once. For example, imagine that a broker received a record from the producer, wrote it to local disk, and the record was successfully replicated to other brokers, but then the first broker crashed before sending a response to the producer. The producer will wait until it reaches `request.timeout.ms` and then retry. The retry will go to the new leader that already has a copy of this record since the previous write was replicated successfully. You now have a duplicate record.

To avoid this, you can set `enable.idempotence=true`. When the idempotent producer is enabled, the producer will attach a sequence number to each record it sends. If the broker receives records with the same sequence number, it will reject the second copy and the producer will receive the harmless `DuplicateSequenceException`.

Enabling idempotence requires `max.in.flight.requests.per.connection` to be less than or equal to 5, `retries` to be greater than 0, and `acks=all`. If incompatible values are set, a `ConfigException` will be thrown.

Serializers

As seen in previous examples, producer configuration includes mandatory serializers. We've seen how to use the default `String` serializer. Kafka also includes serializers for integers, `ByteArrays`, and many more, but this does not cover most use cases. Eventually, you will want to be able to serialize more generic records.

We will start by showing how to write your own serializer and then introduce the Avro serializer as a recommended alternative.

Custom Serializers

When the object you need to send to Kafka is not a simple string or integer, you have a choice of either using a generic serialization library like Avro, Thrift, or Protobuf to create records, or creating a custom serialization for objects you are already using. We highly recommend using a generic serialization library. In order to understand how the serializers work and why it is a good idea to use a serialization library, let's see what it takes to write your own custom serializer.

Suppose that instead of recording just the customer name, you create a simple class to represent customers:

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

Now suppose we want to create a custom serializer for this class. It will look something like this:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

    @Override
    public void configure(Map configs, boolean isKey) {
        // nothing to configure
    }

    @Override
    /**
    We are serializing Customer as:
    4 byte int representing customerId
    4 byte int representing length of customerName in UTF-8 bytes (0 if
        name is Null)
    N bytes representing customerName in UTF-8
```

```
        **/
        public byte[] serialize(String topic, Customer data) {
            try {
                byte[] serializedName;
                int stringSize;
                if (data == null)
                    return null;
                else {
                    if (data.getName() != null) {
                        serializedName = data.getName().getBytes("UTF-8");
                        stringSize = serializedName.length;
                    } else {
                        serializedName = new byte[0];
                        stringSize = 0;
                    }
                }

                ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
                buffer.putInt(data.getID());
                buffer.putInt(stringSize);
                buffer.put(serializedName);

                return buffer.array();
            } catch (Exception e) {
                throw new SerializationException(
                    "Error when serializing Customer to byte[] " + e);
            }
        }

        @Override
        public void close() {
            // nothing to close
        }
    }
```

Configuring a producer with this `CustomerSerializer` will allow you to define `ProducerRecord<String, Customer>`, and send `Customer` data and pass `Customer` objects directly to the producer. This example is pretty simple, but you can see how fragile the code is. If we ever have too many customers, for example, and need to change `customerID` to `Long`, or if we ever decide to add a `startDate` field to `Customer`, we will have a serious issue in maintaining compatibility between old and new messages. Debugging compatibility issues between different versions of serializers and deserializers is fairly challenging: you need to compare arrays of raw bytes. To make matters even worse, if multiple teams in the same company end up writing `Customer` data to Kafka, they will all need to use the same serializers and modify the code at the exact same time.

For these reasons, we recommend using existing serializers and deserializers such as JSON, Apache Avro, Thrift, or Protobuf. In the following section, we will describe Apache Avro and then show how to serialize Avro records and send them to Kafka.

Serializing Using Apache Avro

Apache Avro is a language-neutral data serialization format. The project was created by Doug Cutting to provide a way to share data files with a large audience.

Avro data is described in a language-independent schema. The schema is usually described in JSON, and the serialization is usually to binary files, although serializing to JSON is also supported. Avro assumes that the schema is present when reading and writing files, usually by embedding the schema in the files themselves.

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka, is that when the application that is writing messages switches to a new but compatible schema, the applications reading the data can continue processing messages without requiring any change or update.

Suppose the original schema was:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"} ❶
 ]
}
```

❶ `id` and `name` fields are mandatory, while `faxNumber` is optional and defaults to `null`.

We used this schema for a few months and generated a few terabytes of data in this format. Now suppose we decide that in the new version, we will upgrade to the 21st century and will no longer include a fax number field and will instead use an email field.

The new schema would be:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

Now, after upgrading to the new version, old records will contain `faxNumber` and new records will contain `email`. In many organizations, upgrades are done slowly and over many months. So we need to consider how pre-upgrade applications that still

use the fax numbers and post-upgrade applications that use email will be able to handle all the events in Kafka.

The reading application will contain calls to methods similar to `getName()`, `getId()`, and `getFaxNumber()`. If it encounters a message written with the new schema, `getName()` and `getId()` will continue working with no modification, but `getFaxNumber()` will return `null` because the message will not contain a fax number.

Now suppose we upgrade our reading application and it no longer has the `getFaxNumber()` method but rather `getEmail()`. If it encounters a message written with the old schema, `getEmail()` will return `null` because the older messages do not contain an email address.

This example illustrates the benefit of using Avro: even though we changed the schema in the messages without changing all the applications reading the data, there will be no exceptions or breaking errors and no need for expensive updates of existing data.

However, there are two caveats to this scenario:

- The schema used for writing the data and the schema expected by the reading application must be compatible. The Avro documentation includes compatibility rules.
- The deserializer will need access to the schema that was used when writing the data, even when it is different from the schema expected by the application that accesses the data. In Avro files, the writing schema is included in the file itself, but there is a better way to handle this for Kafka messages. We will look at that next.

Using Avro Records with Kafka

Unlike Avro files, where storing the entire schema in the data file is associated with a fairly reasonable overhead, storing the entire schema in each record will usually more than double the record size. However, Avro still requires the entire schema to be present when reading the record, so we need to locate the schema elsewhere. To achieve this, we follow a common architecture pattern and use a *Schema Registry*. The Schema Registry is not part of Apache Kafka, but there are several open source options to choose from. We'll use the Confluent Schema Registry for this example. You can find the Schema Registry code on GitHub, or you can install it as part of the Confluent Platform. If you decide to use the Schema Registry, we recommend checking the documentation on Confluent.

The idea is to store all the schemas used to write data to Kafka in the registry. Then we simply store the identifier for the schema in the record we produce to Kafka. The consumers can then use the identifier to pull the record out of the Schema Registry

and deserialize the data. The key is that all this work—storing the schema in the registry and pulling it up when required—is done in the serializers and deserializers. The code that produces data to Kafka simply uses the Avro serializer just like it would any other serializer. Figure 3-3 demonstrates this process.

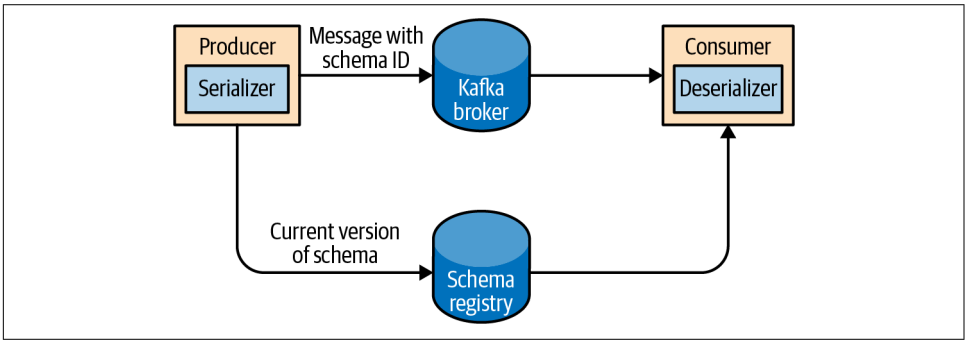

Figure 3-3. Flow diagram of serialization and deserialization of Avro records

Here is an example of how to produce generated Avro objects to Kafka (see the Avro documentation for how to generate objects from Avro schemas):

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";

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

// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext(); ❹
    System.out.println("Generated customer " +
        customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getName(), customer); ❺
    producer.send(record); ❻
}
```

❶ We use the `KafkaAvroSerializer` to serialize our objects with Avro. Note that the `KafkaAvroSerializer` can also handle primitives, which is why we can later use `String` as the record key and our `Customer` object as the value.

❷ `schema.registry.url` is the configuration of the Avro serializer that will be passed to the serializer by the producer. It simply points to where we store the schemas.

❸ `Customer` is our generated object. We tell the producer that our records will contain `Customer` as the value.

❹ `Customer` class is not a regular Java class (plain old Java object, or POJO) but rather a specialized Avro object, generated from a schema using Avro code generation. The Avro serializer can only serialize Avro objects, not POJO. Generating Avro classes can be done either using the *avro-tools.jar* or the Avro Maven plugin, both part of Apache Avro. See the Apache Avro Getting Started (Java) guide for details on how to generate Avro classes.

❺ We also instantiate `ProducerRecord` with `Customer` as the value type, and pass a `Customer` object when creating the new record.

❻ That's it. We send the record with our `Customer` object, and `KafkaAvroSerializer` will handle the rest.

Avro also allows you to use generic Avro objects, that are used as key-value maps, rather than generated Avro objects with getters and setters that match the schema that was used to generate them. To use generic Avro objects, you just need to provide the schema:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

String schemaString =
    "{\"namespace\": \"customerManagement.avro\",
     "\"type\": \"record\", " + ❸
     "\"name\": \"Customer\"," +
     "\"fields\": [" +
      "{\"name\": \"id\", \"type\": \"int\"}," +
      "{\"name\": \"name\", \"type\": \"string\"}," +
      "{\"name\": \"email\", \"type\": " + "[\"null\",\"string\"], " +
       "\"default\":\"null\" }" +
    "]}";
Producer<String, GenericRecord> producer =
   new KafkaProducer<String, GenericRecord>(props); ❹

Schema.Parser parser = new Schema.Parser();
```

```
Schema schema = parser.parse(schemaString);

for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
    String name = "exampleCustomer" + nCustomers;
    String email = "example " + nCustomers + "@example.com";

    GenericRecord customer = new GenericData.Record(schema); ❺
    customer.put("id", nCustomers);
    customer.put("name", name);
    customer.put("email", email);

    ProducerRecord<String, GenericRecord> data =
        new ProducerRecord<>("customerContacts", name, customer);
    producer.send(data);
}
```

❶ We still use the same `KafkaAvroSerializer`.

❷ And we provide the URI of the same Schema Registry.

❸ But now we also need to provide the Avro schema, since it is not provided by an Avro-generated object.

❹ Our object type is an Avro `GenericRecord`, which we initialize with our schema and the data we want to write.

❺ Then the value of the `ProducerRecord` is simply a `GenericRecord` that contains our schema and data. The serializer will know how to get the schema from this record, store it in the Schema Registry, and serialize the object data.

Partitions

In previous examples, the `ProducerRecord` objects we created included a topic name, key, and value. Kafka messages are key-value pairs, and while it is possible to create a `ProducerRecord` with just a topic and a value, with the key set to `null` by default, most applications produce records with keys. Keys serve two goals: they are additional information that gets stored with the message, and they are typically also used to decide which one of the topic partitions the message will be written to (keys also play an important role in compacted topics—we'll discuss those in Chapter 6). All messages with the same key will go to the same partition. This means that if a process is reading only a subset of the partitions in a topic (more on that in Chapter 4), all the records for a single key will be read by the same process. To create a key-value record, you simply create a `ProducerRecord` as follows:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

When creating messages with a null key, you can simply leave the key out:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ Here, the key will simply be set to `null`.

When the key is `null` and the default partitioner is used, the record will be sent to one of the available partitions of the topic at random. A round-robin algorithm will be used to balance the messages among the partitions. Starting in the Apache Kafka 2.4 producer, the round-robin algorithm used in the default partitioner when handling null keys is sticky. This means that it will fill a batch of messages sent to a single partition before switching to the next partition. This allows sending the same number of messages to Kafka in fewer requests, leading to lower latency and reduced CPU utilization on the broker.

If a key exists and the default partitioner is used, Kafka will hash the key (using its own hash algorithm, so hash values will not change when Java is upgraded) and use the result to map the message to a specific partition. Since it is important that a key is always mapped to the same partition, we use all the partitions in the topic to calculate the mapping—not just the available partitions. This means that if a specific partition is unavailable when you write data to it, you might get an error. This is fairly rare, as you will see in Chapter 7 when we discuss Kafka's replication and availability.

In addition to the default partitioner, Apache Kafka clients also provide `RoundRobinPartitioner` and `UniformStickyPartitioner`. These provide random partition assignment and sticky random partition assignment even when messages have keys. These are useful when keys are important for the consuming application (for example, there are ETL applications that use the key from Kafka records as the primary key when loading data from Kafka to a relational database), but the workload may be skewed, so a single key may have a disproportionately large workload. Using the `UniformStickyPartitioner` will result in an even distribution of workload across all partitions.

When the default partitioner is used, the mapping of keys to partitions is consistent only as long as the number of partitions in a topic does not change. So as long as the number of partitions is constant, you can be sure that, for example, records regarding user 045189 will always get written to partition 34. This allows all kinds of optimization when reading data from partitions. However, the moment you add new partitions to the topic, this is no longer guaranteed—the old records will stay in partition 34 while new records may get written to a different partition. When partitioning keys is important, the easiest solution is to create topics with sufficient partitions (the Confluent blog contains suggestions on how to choose the number of partitions) and never add partitions.

Implementing a custom partitioning strategy

So far, we have discussed the traits of the default partitioner, which is the one most commonly used. However, Kafka does not limit you to just hash partitions, and sometimes there are good reasons to partition data differently. For example, suppose that you are a B2B vendor and your biggest customer is a company that manufactures handheld devices called Bananas. Suppose that you do so much business with customer "Banana" that over 10% of your daily transactions are with this customer. If you use default hash partitioning, the Banana records will get allocated to the same partition as other accounts, resulting in one partition being much larger than the rest. This can cause servers to run out of space, processing to slow down, etc. What we really want is to give Banana its own partition and then use hash partitioning to map the rest of the accounts to all other partitions.

Here is an example of a custom partitioner:

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ❶

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes,
                         Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String))) ❷
            throw new InvalidRecordException("We expect all messages " +
                "to have customer name as key");

        if (((String) key).equals("Banana"))
            return numPartitions - 1; // Banana will always go to last partition

        // Other records will get hashed to the rest of the partitions
        return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
    }

    public void close() {}
}
```

❶ Partitioner interface includes `configure`, `partition`, and `close` methods. Here we only implement `partition`, although we really should have passed the special customer name through `configure` instead of hardcoding it in `partition`.

❷ We only expect `String` keys, so we throw an exception if that is not the case.

Headers

Records can, in addition to key and value, also include headers. Record headers give you the ability to add some metadata about the Kafka record, without adding any extra information to the key/value pair of the record itself. Headers are often used for lineage to indicate the source of the data in the record, and for routing or tracing messages based on header information without having to parse the message itself (perhaps the message is encrypted and the router doesn't have permissions to access the data).

Headers are implemented as an ordered collection of key/value pairs. The keys are always a `String`, and the values can be any serialized object—just like the message value.

Here is a small example that shows how to add headers to a `ProduceRecord`:

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

record.headers().add("privacy-level","YOLO".getBytes(StandardCharsets.UTF_8));
```

Interceptors

There are times when you want to modify the behavior of your Kafka client application without modifying its code, perhaps because you want to add identical behavior to all applications in the organization. Or perhaps you don't have access to the original code.

Kafka's `ProducerInterceptor` interceptor includes two key methods:

`ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)`
: This method will be called before the produced record is sent to Kafka, indeed before it is even serialized. When overriding this method, you can capture information about the sent record and even modify it. Just be sure to return a valid `ProducerRecord` from this method. The record that this method returns will be serialized and sent to Kafka.

`void onAcknowledgement(RecordMetadata metadata, Exception exception)`
: This method will be called if and when Kafka responds with an acknowledgment for a send. The method does not allow modifying the response from Kafka, but you can capture information about the response.

Common use cases for producer interceptors include capturing monitoring and tracing information; enhancing the message with standard headers, especially for lineage tracking purposes; and redacting sensitive information.

Here is an example of a very simple producer interceptor. This one simply counts the messages sent and acks received within specific time windows:

```
public class CountingProducerInterceptor implements ProducerInterceptor {

  ScheduledExecutorService executorService =
          Executors.newSingleThreadScheduledExecutor();
  static AtomicLong numSent = new AtomicLong(0);
  static AtomicLong numAcked = new AtomicLong(0);

  public void configure(Map<String, ?> map) {
      Long windowSize = Long.valueOf(
              (String) map.get("counting.interceptor.window.size.ms")); ❶
      executorService.scheduleAtFixedRate(CountingProducerInterceptor::run,
              windowSize, windowSize, TimeUnit.MILLISECONDS);
  }

  public ProducerRecord onSend(ProducerRecord producerRecord) {
      numSent.incrementAndGet();
      return producerRecord; ❷
  }

  public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
      numAcked.incrementAndGet(); ❸
  }

  public void close() {
      executorService.shutdownNow(); ❹
  }

  public static void run() {
      System.out.println(numSent.getAndSet(0));
      System.out.println(numAcked.getAndSet(0));
  }

}
```

❶ `ProducerInterceptor` is a `Configurable` interface. You can override the `configure` method and setup before any other method is called. This method receives the entire producer configuration, and you can access any configuration parameter. In this case, we added a configuration of our own that we reference here.

❷ When a record is sent, we increment the record count and return the record without modifying it.

❸ When Kafka responds with an ack, we increment the acknowledgment count and don't need to return anything.

❹ This method is called when the producer closes, giving us a chance to clean up the interceptor state. In this case, we close the thread we created. If you opened file handles, connections to remote data stores, or similar, this is the place to close everything and avoid leaks.

As we mentioned earlier, producer interceptors can be applied without any changes to the client code. To use the preceding interceptor with `kafka-console-producer`, an example application that ships with Apache Kafka, follow these three simple steps:

1. Add your jar to the classpath:

   ```
   export CLASSPATH=$CLASSPATH:~./target/CountProducerInterceptor-1.0-SNAPSHOT.jar
   ```

2. Create a config file that includes:

   ```
   interceptor.classes=com.shapira.examples.interceptors.CountProducerInterceptor counting.interceptor.window.size.ms=10000
   ```

3. Run the application as you normally would, but make sure to include the configuration that you created in the previous step:

   ```
   bin/kafka-console-producer.sh --broker-list localhost:9092 --topic interceptor-test --producer.config producer.config
   ```

Quotas and Throttling

Kafka brokers have the ability to limit the rate at which messages are produced and consumed. This is done via the quota mechanism. Kafka has three quota types: produce, consume, and request. Produce and consume quotas limit the rate at which clients can send and receive data, measured in bytes per second. Request quotas limit the percentage of time the broker spends processing client requests.

Quotas can be applied to all clients by setting default quotas, specific client-ids, specific users, or both. User-specific quotas are only meaningful in clusters where security is configured and clients authenticate.

The default produce and consume quotas that are applied to all clients are part of the Kafka broker configuration file. For example, to limit each producer to send no more than 2 MBps on average, add the following configuration to the broker configuration file: `quota.producer.default=2M`.

While not recommended, you can also configure specific quotas for certain clients that override the default quotas in the broker configuration file. To allow clientA to

produce 4 MBps and clientB 10 MBps, you can use the following: `quota.producer.override="clientA:4M,clientB:10M"`

Quotas that are specified in Kafka's configuration file are static, and you can only modify them by changing the configuration and then restarting all the brokers. Since new clients can arrive at any time, this is very inconvenient. Therefore the usual method of applying quotas to specific clients is through dynamic configuration that can be set using `kafka-config.sh` or the AdminClient API.

Let's look at few examples:

```
bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config 'pro-
ducer_byte_rate=1024' --entity-name clientC --entity-type clients ❶

bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config 'pro-
ducer_byte_rate=1024,consumer_byte_rate=2048' --entity-name user1 --entity-type
users ❷

bin/kafka-configs  --bootstrap-server localhost:9092 --alter --add-config 'con-
sumer_byte_rate=2048' --entity-type users ❸
```

❶ Limiting clientC (identified by client-id) to produce only 1024 bytes per second

❷ Limiting user1 (identified by authenticated principal) to produce only 1024 bytes per second and consume only 2048 bytes per second.

❸ Limiting all users to consume only 2048 bytes per second, except users with more specific override. This is the way to dynamically modify the default quota.

When a client reaches its quota, the broker will start throttling the client's requests to prevent it from exceeding the quota. This means that the broker will delay responses to client requests; in most clients this will automatically reduce the request rate (since the number of in-flight requests is limited) and bring the client traffic down to a level allowed by the quota. To protect the broker from misbehaved clients sending additional requests while being throttled, the broker will also mute the communication channel with the client for the period of time needed to achieve compliance with the quota.

The throttling behavior is exposed to clients via `produce-throttle-time-avg`, `produce-throttle-time-max`, `fetch-throttle-time-avg`, and `fetch-throttle-time-max`, the average and the maximum amount of time a produce request and fetch request was delayed due to throttling. Note that this time can represent throttling due to produce and consume throughput quotas, request time quotas, or both. Other types of client requests can only be throttled due to request time quotas, and those will also be exposed via similar metrics.

If you use async `Producer.send()` and continue to send messages at a rate that is higher than the rate the broker can accept (whether due to quotas or just plain old capacity), the messages will first be queued in the client memory. If the rate of sending continues to be higher than the rate of accepting messages, the client will eventually run out of buffer space for storing the excess messages and will block the next `Producer.send()` call. If the timeout delay is insufficient to let the broker catch up to the producer and clear some space in the buffer, eventually `Producer.send()` will throw `TimeoutException`. Alternatively, some of the records that were already placed in batches will wait for longer than `delivery.timeout.ms` and expire, resulting in calling the `send()` callback with a `TimeoutException`. It is therefore important to plan and monitor to make sure that the broker capacity over time will match the rate at which producers are sending data.

Summary

We began this chapter with a simple example of a producer—just 10 lines of code that send events to Kafka. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behavior of the producers. We discussed serializers, which let us control the format of the events we write to Kafka. We looked in-depth at Avro, one of many ways to serialize events but one that is very commonly used with Kafka. We concluded the chapter with a discussion of partitioning in Kafka and an example of an advanced custom partitioning technique.

Now that we know how to write events to Kafka, in Chapter 4 we'll learn all about consuming events from Kafka.

CHAPTER 4

Kafka Consumers: Reading Data from Kafka

Applications that need to read data from Kafka use a `KafkaConsumer` to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are a few unique concepts and ideas involved. It can be difficult to understand how to use the Consumer API without understanding these concepts first. We'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways Consumer APIs can be used to implement applications with varying requirements.

Kafka Consumer Concepts

To understand how to read data from Kafka, you first need to understand its consumers and consumer groups. The following sections cover those concepts.

Consumers and Consumer Groups

Suppose you have an application that needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store. In this case, your application will create a consumer object, subscribe to the appropriate topic, and start receiving messages, validating them, and writing the results. This may work well for a while, but what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall further and further behind, unable to keep up with the rate of incoming messages. Obviously there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data among them.

Kafka consumers are typically part of a *consumer group*. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Let's take topic T1 with four partitions. Now suppose we created a new consumer, C1, which is the only consumer in group G1, and use it to subscribe to topic T1. Consumer C1 will get all messages from all four T1 partitions. See Figure 4-1.

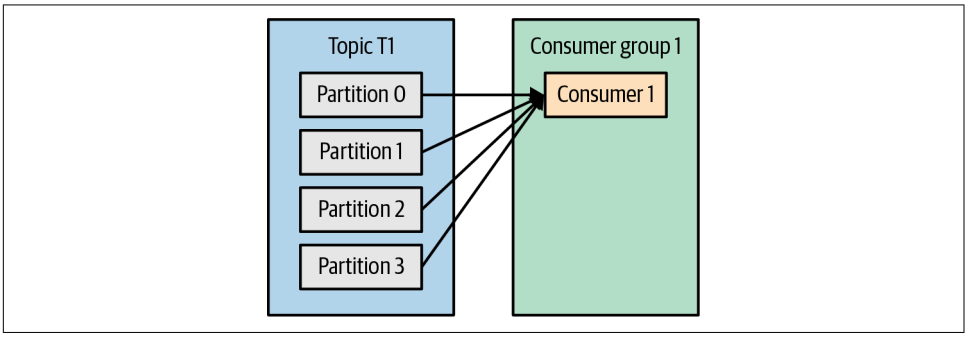

Figure 4-1. One consumer group with four partitions

If we add another consumer, C2, to group G1, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to C1, and messages from partitions 1 and 3 go to consumer C2. See Figure 4-2.

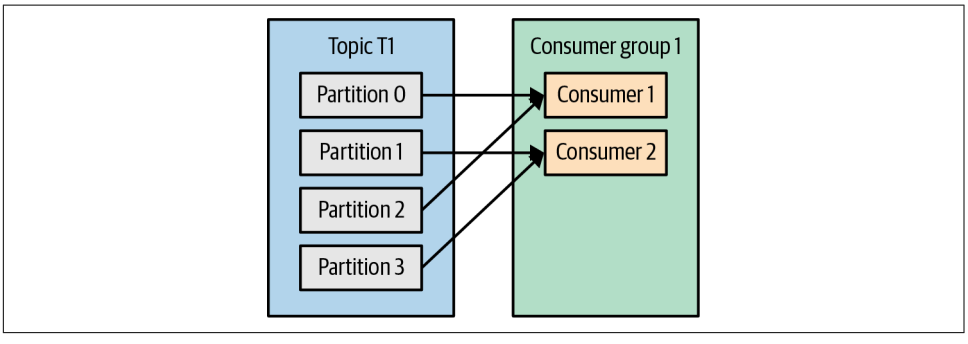

Figure 4-2. Four partitions split to two consumers in a group

If G1 has four consumers, then each will read messages from a single partition. See Figure 4-3.

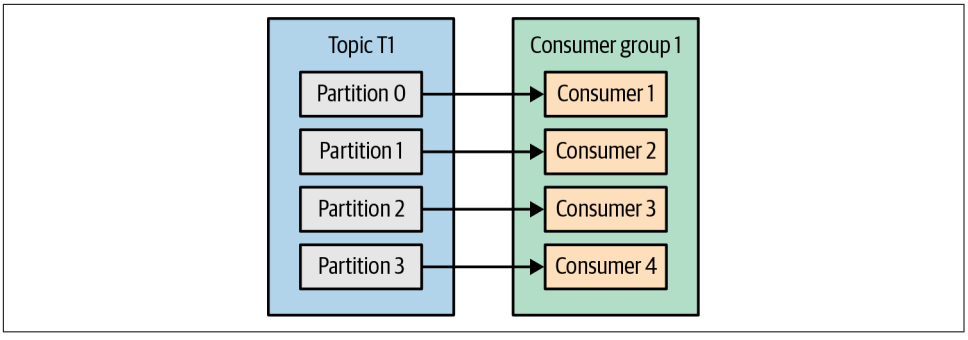

Figure 4-3. Four consumers in a group with one partition each

If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all. See Figure 4-4.

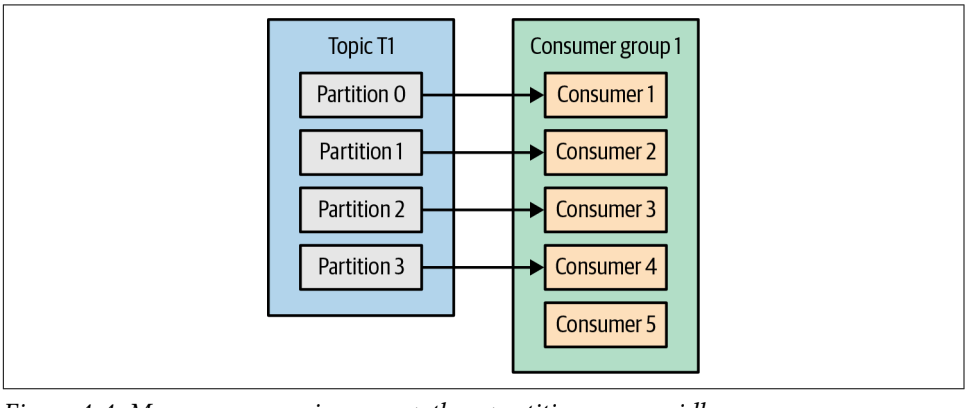

Figure 4-4. More consumers in a group than partitions means idle consumers

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data. In these cases, a single consumer can't possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic—some of the consumers will just be idle. Chapter 2 includes some suggestions on how to choose the number of partitions in a topic.

In addition to adding consumers in order to scale a single application, it is very common to have multiple applications that need to read data from the same topic. In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics

available for many use cases throughout the organization. In those cases, we want each application to get all of the messages, rather than just a subset. To make sure an application gets all the messages in a topic, ensure the application has its own consumer group. Unlike many traditional messaging systems, Kafka scales to a large number of consumers and consumer groups without reducing performance.

In the previous example, if we add a new consumer group (G2) with a single consumer, this consumer will get all the messages in topic T1 independent of what G1 is doing. G2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for G1, but G2 as a whole will still get all the messages regardless of other consumer groups. See Figure 4-5.

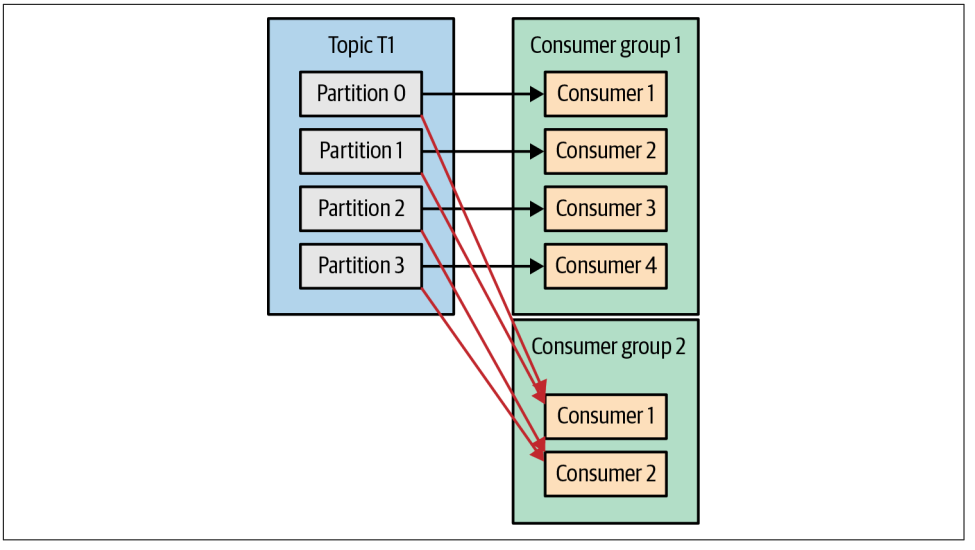

Figure 4-5. Adding a new consumer group, both groups receive all messages

To summarize, you create a new consumer group for each application that needs all the messages from one or more topics. You add consumers to an existing consumer group to scale the reading and processing of messages from the topics, so each additional consumer in a group will only get a subset of the messages.

Consumer Groups and Partition Rebalance

As we saw in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happens when

the topics the consumer group is consuming are modified (e.g., if an administrator adds new partitions).

Moving partition ownership from one consumer to another is called a *rebalance*. Rebalances are important because they provide the consumer group with high availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events they can be fairly undesirable.

There are two types of rebalances, depending on the partition assignment strategy that the consumer group uses:[1]

Eager rebalances

During an eager rebalance, all consumers stop consuming, give up their ownership of all partitions, rejoin the consumer group, and get a brand-new partition assignment. This is essentially a short window of unavailability of the entire consumer group. The length of the window depends on the size of the consumer group as well as on several configuration parameters. Figure 4-6 shows how eager rebalances have two distinct phases: first, all consumers give up their partition assigning, and second, after they all complete this and rejoin the group, they get new partition assignments and can resume consuming.

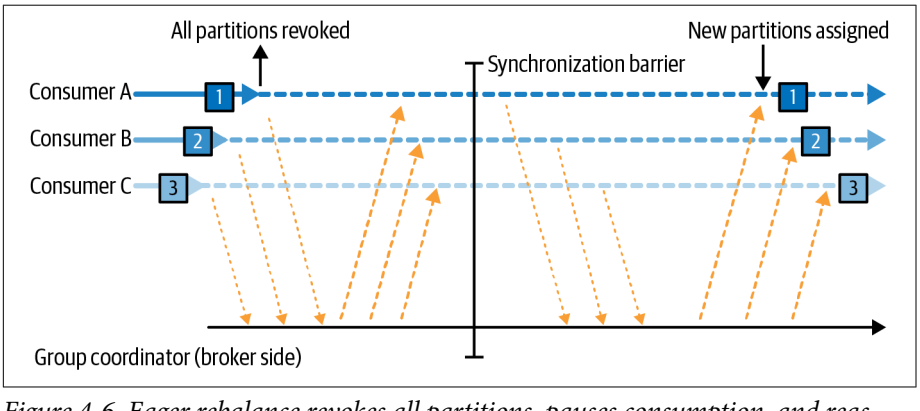

Figure 4-6. Eager rebalance revokes all partitions, pauses consumption, and reassigns them

Cooperative rebalances

Cooperative rebalances (also called *incremental rebalances*) typically involve reassigning only a small subset of the partitions from one consumer to another, and allowing consumers to continue processing records from all the partitions that are not reassigned. This is achieved by rebalancing in two or more phases.

1 Diagrams by Sophie Blee-Goldman, from her May 2020 blog post, "From Eager to Smarter in Apache Kafka Consumer Rebalances".

Initially, the consumer group leader informs all the consumers that they will lose ownership of a subset of their partitions, then the consumers stop consuming from these partitions and give up their ownership in them. In the second phase, the consumer group leader assigns these now orphaned partitions to their new owners. This incremental approach may take a few iterations until a stable partition assignment is achieved, but it avoids the complete "stop the world" unavailability that occurs with the eager approach. This is especially important in large consumer groups where rebalances can take a significant amount of time. Figure 4-7 shows how cooperative rebalances are incremental and that only a subset of the consumers and partitions are involved.

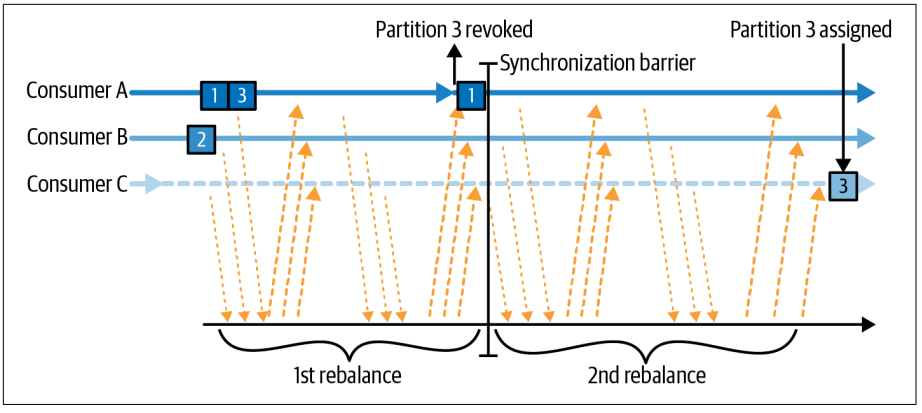

Figure 4-7. Cooperative rebalance only pauses consumption for the subset of partitions that will be reassigned

Consumers maintain membership in a consumer group and ownership of the partitions assigned to them by sending *heartbeats* to a Kafka broker designated as the *group coordinator* (this broker can be different for different consumer groups). The heartbeats are sent by a background thread of the consumer, and as long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive.

If the consumer stops sending heartbeats for long enough, its session will timeout and the group coordinator will consider it dead and trigger a rebalance. If a consumer crashed and stopped processing messages, it will take the group coordinator a few seconds without heartbeats to decide it is dead and trigger the rebalance. During those seconds, no messages will be processed from the partitions owned by the dead consumer. When closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately, reducing the gap in processing. Later in this chapter, we will discuss configuration options that control heartbeat frequency, session timeouts, and other configuration parameters that can be used to fine-tune the consumer behavior.

How Does the Process of Assigning Partitions to Consumers Work?

When a consumer wants to join a group, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and that are therefore considered alive) and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of `PartitionAssignor` to decide which partitions should be handled by which consumer.

Kafka has few built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer group leader sends the list of assignments to the `GroupCoordinator`, which sends this information to all the consumers. Each consumer only sees its own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Static Group Membership

By default, the identity of a consumer as a member of its consumer group is transient. When consumers leave a consumer group, the partitions that were assigned to the consumer are revoked, and when it rejoins, it is assigned a new member ID and a new set of partitions through the rebalance protocol.

All this is true unless you configure a consumer with a unique `group.instance.id`, which makes the consumer a *static* member of the group. When a consumer first joins a consumer group as a static member of the group, it is assigned a set of partitions according to the partition assignment strategy the group is using, as normal. However, when this consumer shuts down, it does not automatically leave the group—it remains a member of the group until its session times out. When the consumer rejoins the group, it is recognized with its static identity and is reassigned the same partitions it previously held without triggering a rebalance. The group coordinator that caches the assignment for each member of the group does not need to trigger a rebalance but can just send the cache assignment to the rejoining static member.

If two consumers join the same group with the same `group.instance.id`, the second consumer will get an error saying that a consumer with this ID already exists.

Static group membership is useful when your application maintains local state or cache that is populated by the partitions that are assigned to each consumer. When re-creating this cache is time-consuming, you don't want this process to happen every time a consumer restarts. On the flip side, it is important to remember that the

partitions owned by each consumer will not get reassigned when a consumer is restarted. For a certain duration, no consumer will consume messages from these partitions, and when the consumer finally starts back up, it will lag behind the latest messages in these partitions. You should be confident that the consumer that owns these partitions will be able to catch up with the lag after the restart.

It is important to note that static members of consumer groups do not leave the group proactively when they shut down, and detecting when they are "really gone" depends on the `session.timeout.ms` configuration. You'll want to set it high enough to avoid triggering rebalances on a simple application restart but low enough to allow automatic reassignment of their partitions when there is more significant downtime, to avoid large gaps in processing these partitions.

Creating a Kafka Consumer

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer`—you create a Java `Properties` instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start, we just need to use the three mandatory properties: `bootstrap.servers`, `key.deserializer`, and `value.deserializer`.

The first property, `bootstrap.servers`, is the connection string to a Kafka cluster. It is used the exact same way as in `KafkaProducer` (refer to Chapter 3 for details on how this is defined). The other two properties, `key.deserializer` and `value.deserializer`, are similar to the `serializers` defined for the producer, but rather than specifying classes that turn Java objects to byte arrays, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory but very commonly used. The property is `group.id`, and it specifies the consumer group the `KafkaConsumer` instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is uncommon, so for most of the chapter we will assume the consumer is part of a group.

The following code snippet shows how to create a `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
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<String, String>(props);
```

Most of what you see here should be familiar if you've read Chapter 3 on creating producers. We assume that the records we consume will have `String` objects as both the key and the value of the record. The only new property here is `group.id`, which is the name of the consumer group this consumer belongs to.

Subscribing to Topics

Once we create a consumer, the next step is to subscribe to one or more topics. The `subscribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system or streams processing applications.

For example, to subscribe to all test topics, we can call:

```
consumer.subscribe(Pattern.compile("test.*"));
```

If your Kafka cluster has large number of partitions, perhaps 30,000 or more, you should be aware that the filtering of topics for the subscription is done on the client side. This means that when you subscribe to a subset of topics via a regular expression rather than via an explicit list, the consumer will request the list of all topics and their partitions from the broker in regular intervals. The client will then use this list to detect new topics that it should include in its subscription and subscribe to them. When the topic list is large and there are many consumers, the size of the list of topics and partitions is significant, and the regular expression subscription has significant overhead on the broker, client, and network. There are cases where the bandwidth used by the topic metadata is larger than the bandwidth used to send data. This also means that in order to subscribe with a regular expression, the client needs permissions to describe all topics in the cluster—that is, a full `describe` grant on the entire cluster.

The Poll Loop

At the heart of the Consumer API is a simple loop for polling the server for more data. The main body of a consumer will look as follows:

```
Duration timeout = Duration.ofMillis(100);

while (true) { ❶
    ConsumerRecords<String, String> records = consumer.poll(timeout); ❷

    for (ConsumerRecord<String, String> record : records) { ❸
        System.out.printf("topic = %s, partition = %d, offset = %d, " +
                        "customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);

        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()); ❹
    }
}
```

❶ This is indeed an infinite loop. Consumers are usually long-running applications that continuously poll Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.

❷ This is the most important line in the chapter. The same way that sharks must keep moving or they die, consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group to continue consuming. The parameter we pass to `poll()` is a timeout interval and controls how long `poll()` will block if data is not available in the consumer buffer. If this is set to 0 or if there are records available already, `poll()` will return immediately; otherwise, it will wait for the specified number of milliseconds.

❸ `poll()` returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and, of course, the key and the value of the record. Typically, we want to iterate over the list and process the records individually.

❹ Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each country,

so we update a hash table and print the result as JSON. A more realistic example would store the updates result in a data store.

The `poll` loop does a lot more than just get data. The first time you call `poll()` with a new consumer, it is responsible for finding the `GroupCoordinator`, joining the consumer group, and receiving a partition assignment. If a rebalance is triggered, it will be handled inside the poll loop as well, including related callbacks. This means that almost everything that can go wrong with a consumer or in the callbacks used in its listeners is likely to show up as an exception thrown by `poll()`.

Keep in mind that if `poll()` is not invoked for longer than `max.poll.interval.ms`, the consumer will be considered dead and evicted from the consumer group, so avoid doing anything that can block for unpredictable intervals inside the poll loop.

Thread Safety

You can't have multiple consumers that belong to the same group in one thread, and you can't have multiple threads safely use the same consumer. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. It is useful to wrap the consumer logic in its own object and then use Java's `ExecutorService` to start multiple threads, each with its own consumer. The Confluent blog has a tutorial that shows how to do just that.

In older versions of Kafka, the full method signature was `poll(long)`; this signature is now deprecated and the new API is `poll(Duration)`. In addition to the change of argument type, the semantics of how the method blocks subtly changed. The original method, `poll(long)`, will block as long as it takes to get the needed metadata from Kafka, even if this is longer than the timeout duration. The new method, `poll(Duration)`, will adhere to the timeout restrictions and not wait for metadata. If you have existing consumer code that uses `poll(0)` as a method to force Kafka to get the metadata without consuming any records (a rather common hack), you can't just change it to `poll(Duration.ofMillis(0))` and expect the same behavior. You'll need to figure out a new way to achieve your goals. Often the solution is placing the logic in the `rebalanceListener.onPartitionAssignment()` method, which is guaranteed to get called after you have metadata for the assigned partitions but before records start arriving. Another solution was documented by Jesse Anderson in his blog post "Kafka's Got a Brand-New Poll".

Another approach can be to have one consumer populate a queue of events and have multiple worker threads perform work from this queue. You can see an example of this pattern in a blog post from Igor Buzatović.

Configuring Consumers

So far we have focused on learning the Consumer API, but we've only looked at a few of the configuration properties—just the mandatory `bootstrap.servers`, `group.id`, `key.deserializer`, and `value.deserializer`. All of the consumer configuration is documented in the Apache Kafka documentation. Most of the parameters have reasonable defaults and do not require modification, but some have implications on the performance and availability of the consumers. Let's take a look at some of the more important properties.

fetch.min.bytes

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records, by default one byte. If a broker receives a request for records from a consumer but the new records amount to fewer bytes than `fetch.min.bytes`, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the consumer and the broker, as they have to handle fewer back-and-forth messages in cases where the topics don't have much new activity (or for lower-activity hours of the day). You will want to set this parameter higher than the default if the consumer is using too much CPU when there isn't much data available, or reduce load on the brokers when you have a large number of consumers—although keep in mind that increasing this value can increase latency for low-throughput cases.

fetch.max.wait.ms

By setting `fetch.min.bytes`, you tell Kafka to wait until it has enough data to send before responding to the consumer. `fetch.max.wait.ms` lets you control how long to wait. By default, Kafka will wait up to 500 ms. This results in up to 500 ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set `fetch.max.wait.ms` to a lower value. If you set `fetch.max.wait.ms` to 100 ms and `fetch.min.bytes` to 1 MB, Kafka will receive a fetch request from the consumer and will respond with data either when it has 1 MB of data to return or after 100 ms, whichever happens first.

fetch.max.bytes

This property lets you specify the maximum bytes that Kafka will return whenever the consumer polls a broker (50 MB by default). It is used to limit the size of memory that the consumer will use to store data that was returned from the server, irrespective of how many partitions or messages were returned. Note that records are sent to the client in batches, and if the first record-batch that the broker has to send exceeds this size, the batch will be sent and the limit will be ignored. This guarantees that the consumer can continue making progress. It's worth noting that there is a matching broker configuration that allows the Kafka administrator to limit the maximum fetch size as well. The broker configuration can be useful because requests for large amounts of data can result in large reads from disk and long sends over the network, which can cause contention and increase load on the broker.

max.poll.records

This property controls the maximum number of records that a single call to `poll()` will return. Use this to control the amount of data (but not the size of data) your application will need to process in one iteration of the poll loop.

max.partition.fetch.bytes

This property controls the maximum number of bytes the server will return per partition (1 MB by default). When `KafkaConsumer.poll()` returns `ConsumerRecords`, the record object will use at most `max.partition.fetch.bytes` per partition assigned to the consumer. Note that controlling memory usage using this configuration can be quite complex, as you have no control over how many partitions will be included in the broker response. Therefore, we highly recommend using `fetch.max.bytes` instead, unless you have special reasons to try and process similar amounts of data from each partition.

session.timeout.ms and heartbeat.interval.ms

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 10 seconds. If more than `session.timeout.ms` passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`, which controls how frequently the Kafka consumer will send a heartbeat to the group coordinator, whereas `session.timeout.ms` controls how long a consumer can go without sending a heartbeat. Therefore, those two properties are typically modified together—`heartbeat.interval.ms` must be lower than `session.timeout.ms` and is usually set to one-third

of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than the default will allow consumer groups to detect and recover from failure sooner but may also cause unwanted rebalances. Setting `session.timeout.ms` higher will reduce the chance of accidental rebalance but also means it will take longer to detect a real failure.

max.poll.interval.ms

This property lets you set the length of time during which the consumer can go without polling before it is considered dead. As mentioned earlier, heartbeats and session timeouts are the main mechanism by which Kafka detects dead consumers and takes their partitions away. However, we also mentioned that heartbeats are sent by a background thread. There is a possibility that the main thread consuming from Kafka is deadlocked, but the background thread is still sending heartbeats. This means that records from partitions owned by this consumer are not being processed. The easiest way to know whether the consumer is still processing records is to check whether it is asking for more records. However, the intervals between requests for more records are difficult to predict and depend on the amount of available data, the type of processing done by the consumer, and sometimes on the latency of additional services. In applications that need to do time-consuming processing on each record that is returned, `max.poll.records` is used to limit the amount of data returned and therefore limit the duration before the application is available to `poll()` again. Even with `max.poll.records` defined, the interval between calls to `poll()` is difficult to predict, and `max.poll.interval.ms` is used as a fail-safe or backstop. It has to be an interval large enough that it will very rarely be reached by a healthy consumer but low enough to avoid significant impact from a hanging consumer. The default value is 5 minutes. When the timeout is hit, the background thread will send a "leave group" request to let the broker know that the consumer is dead and the group must rebalance, and then stop sending heartbeats.

default.api.timeout.ms

This is the timeout that will apply to (almost) all API calls made by the consumer when you don't specify an explicit timeout while calling the API. The default is 1 minute, and since it is higher than the request timeout default, it will include a retry when needed. The notable exception to APIs that use this default is the `poll()` method that always requires an explicit timeout.

request.timeout.ms

This is the maximum amount of time the consumer will wait for a response from the broker. If the broker does not respond within this time, the client will assume the

broker will not respond at all, close the connection, and attempt to reconnect. This configuration defaults to 30 seconds, and it is recommended not to lower it. It is important to leave the broker with enough time to process the request before giving up—there is little to gain by resending requests to an already overloaded broker, and the act of disconnecting and reconnecting adds even more overhead.

auto.offset.reset

This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset, or if the committed offset it has is invalid (usually because the consumer was down for so long that the record with that offset was already aged out of the broker). The default is "latest," which means that lacking a valid offset, the consumer will start reading from the newest records (records that were written after the consumer started running). The alternative is "earliest," which means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning. Setting `auto.offset.reset` to `none` will cause an exception to be thrown when attempting to consume from an invalid offset.

enable.auto.commit

This parameter controls whether the consumer will commit offsets automatically, and defaults to `true`. Set it to `false` if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set `enable.auto.commit` to `true`, then you might also want to control how frequently offsets will be committed using `auto.commit.interval.ms`. We'll discuss the different options for committing offsets in more depth later in this chapter.

partition.assignment.strategy

We learned that partitions are assigned to consumers in a consumer group. A `PartitionAssignor` is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default, Kafka has the following assignment strategies:

Range

Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and

the number of consumers does not divide the number of partitions in each topic neatly.

RoundRobin
: Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1, and partition 1 from topic T2. C2 would have partition 1 from topic T1, and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most one partition difference).

Sticky
: The Sticky Assignor has two goals: the first is to have an assignment that is as balanced as possible, and the second is that in case of a rebalance, it will leave as many assignments as possible in place, minimizing the overhead associated with moving partition assignments from one consumer to another. In the common case where all consumers are subscribed to the same topic, the initial assignment from the Sticky Assignor will be as balanced as that of the RoundRobin Assignor. Subsequent assignments will be just as balanced but will reduce the number of partition movements. In cases where consumers in the same group subscribe to different topics, the assignment achieved by Sticky Assignor is more balanced than that of the RoundRobin Assignor.

Cooperative Sticky
: This assignment strategy is identical to that of the Sticky Assignor but supports cooperative rebalances in which consumers can continue consuming from the partitions that are not reassigned. See "Consumer Groups and Partition Rebalance" on page 80 to read more about cooperative rebalancing, and note that if you are upgrading from a version older than 2.3, you'll need to follow a specific upgrade path in order to enable the cooperative sticky assignment strategy, so pay extra attention to the upgrade guide.

The `partition.assignment.strategy` allows you to choose a partition assignment strategy. The default is `org.apache.kafka.clients.consumer.RangeAssignor`, which implements the Range strategy described earlier. You can replace it with `org.apache.kafka.clients.consumer.RoundRobinAssignor`, `org.apache.kafka.clients.consumer.StickyAssignor`, or `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`. A more advanced option is to implement your own assignment strategy, in which case `partition.assignment.strategy` should point to the name of your class.

client.id

This can be any string, and will be used by the brokers to identify requests sent from the client, such as fetch requests. It is used in logging and metrics, and for quotas.

client.rack

By default, consumers will fetch messages from the leader replica of each partition. However, when the cluster spans multiple datacenters or multiple cloud availability zones, there are advantages both in performance and in cost to fetching messages from a replica that is located in the same zone as the consumer. To enable fetching from the closest replica, you need to set the `client.rack` configuration and identify the zone in which the client is located. Then you can configure the brokers to replace the default `replica.selector.class` with `org.apache.kafka.common.replica.RackAwareReplicaSelector`.

You can also implement your own `replica.selector.class` with custom logic for choosing the best replica to consume from, based on client metadata and partition metadata.

group.instance.id

This can be any unique string and is used to provide a consumer with static group membership.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to –1, the OS defaults will be used. It can be a good idea to increase these when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

offsets.retention.minutes

This is a broker configuration, but it is important to be aware of it due to its impact on consumer behavior. As long as a consumer group has active members (i.e., members that are actively maintaining membership in the group by sending heartbeats), the last offset committed by the group for each partition will be retained by Kafka, so it can be retrieved in case of reassignment or restart. However, once a group becomes empty, Kafka will only retain its committed offsets to the duration set by this configuration—7 days by default. Once the offsets are deleted, if the group becomes active again it will behave like a brand-new consumer group with no memory of anything it consumed in the past. Note that this behavior changed a few times, so if you use

versions older than 2.1.0, check the documentation for your version for the expected behavior.

Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group have not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As discussed before, one of Kafka's unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

We call the action of updating the current position in the partition an `offset commit`. Unlike traditional message queues, Kafka does not commit records individually. Instead, consumers commit the last message they've successfully processed from a partition and implicitly assume that every message before the last was also successfully processed.

How does a consumer commit an offset? It sends a message to Kafka, which updates a special `__consumer_offsets` topic with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice. See Figure 4-8.

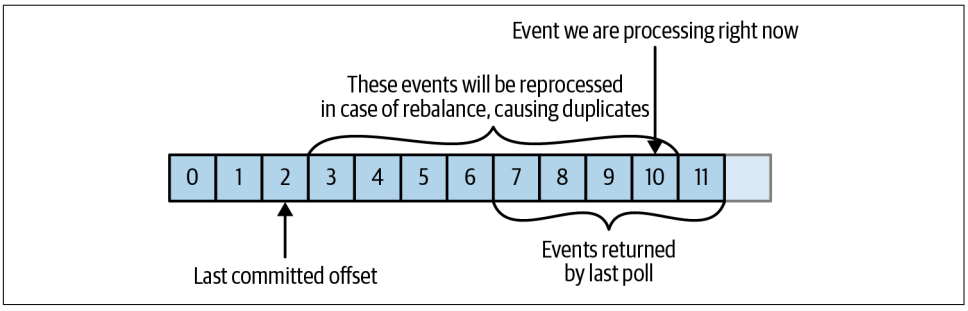

Figure 4-8. Reprocessed messages

If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group. See Figure 4-9.

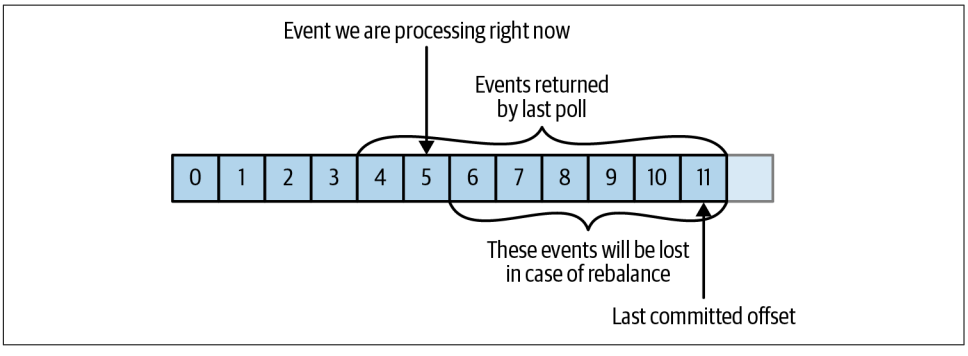

Figure 4-9. Missed messages between offsets

Clearly, managing offsets has a big impact on the client application. The `KafkaConsumer` API provides multiple ways of committing offsets.

Which Offset Is Committed?

When committing offsets either automatically or without specifying the intended offsets, the default behavior is to commit the offset after the last offset that was returned by `poll()`. This is important to keep in mind when attempting to manually commit specific offsets or seek to commit specific offsets. However, it is also tedious to repeatedly read "Commit the offset that is one larger than the last offset the client received from `poll()`," and 99% of the time it does not matter. So, we are going to write "Commit the last offset" when we refer to the default behavior, and if you need to manually manipulate offsets, please keep this note in mind.

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit=true`, then every five seconds the consumer will commit the latest offset that your client received from `poll()`. The five-second interval is the default and is controlled by setting `auto.commit.interval.ms`. Just like everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll.

Before using this convenient option, however, it is important to understand the consequences.

Consider that, by default, automatic commits occur every five seconds. Suppose that we are three seconds after the most recent commit our consumer crashed. After the rebalancing, the surviving consumers will start consuming the partitions that were previously owned by the crashed broker. But they will start from the last offset committed. In this case, the offset is three seconds old, so all the events that arrived in those three seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them.

With autocommit enabled, when it is time to commit offsets, the next poll will commit the last offset returned by the previous poll. It doesn't know which events were actually processed, so it is critical to always process all the events returned by `poll()` before calling `poll()` again. (Just like `poll()`, `close()` also commits offsets automatically.) This is usually not an issue, but pay attention when you handle exceptions or exit the poll loop prematurely.

Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commit Current Offset

Most developers exercise more control over the time at which offsets are committed—both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The Consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `enable.auto.commit=false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if the commit fails for some reason.

It is important to remember that `commitSync()` will commit the latest offset returned by `poll()`, so if you call `commitSync()` before you are done processing all the records in the collection, you risk missing the messages that were committed but not processed, in case the application crashes. If the application crashes while it is still processing records in the collection, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice—this may or may not be preferable to missing messages.

Here is how we would use `commitSync` to commit offsets after we finished processing the latest batch of messages:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %d, offset =
            %d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
    }
    try {
        consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ❸
    }
}
```

❶ Let's assume that by printing the contents of a record, we are done processing it. Your application will likely do a lot more with the records—modify them, enrich them, aggregate them, display them on a dashboard, or notify users of important events. You should determine when you are "done" with a record according to your use case.

❷ Once we are done "processing" all the records in the current batch, we call `commitSync` to commit the last offset in the batch, before polling for additional messages.

❸ `commitSync` retries committing as long as there is no error that can't be recovered. If this happens, there is not much we can do except log an error.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance may create.

Another option is the asynchronous commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
```

```
                offset = %d, customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
```

❶ Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a nonretriable failure, `commitAsync()` will not retry. The reason it does not retry is that by the time `commitAsync()` receives a response from the server, there may have been a later commit that was already successful. Imagine that we sent a request to commit offset 2000. There is a temporary communication problem, so the broker never gets the request and therefore never responds. Meanwhile, we processed another batch and successfully committed offset 3000. If `commitAsync()` now retries the previously failed commit, it might succeed in committing offset 2000 *after* offset 3000 was already processed and committed. In the case of a rebalance, this will cause more duplicates.

We mention this complication and the importance of correct order of commits because `commitAsync()` also gives you an option to pass in a callback that will be triggered when the broker responds. It is common to use the callback to log commit errors or to count them in a metric, but if you want to use the callback for retries, you need to be aware of the problem with commit order:

```
    Duration timeout = Duration.ofMillis(100);

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s,
            offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
        }
        consumer.commitAsync(new OffsetCommitCallback() {
            public void onComplete(Map<TopicPartition,
            OffsetAndMetadata> offsets, Exception e) {
                if (e != null)
                    log.error("Commit failed for offsets {}", offsets, e);
            }
        }); ❶
    }
```

❶ We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged.

Retrying Async Commits

A simple pattern to get the commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit, and add the sequence number at the time of the commit to the `commitAsync` callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable; if it is, there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry because a newer commit was already sent.

Combining Synchronous and Asynchronous Commits

Normally, occasional failures to commit without retrying are not a huge problem because if the problem is temporary, the following commit will be successful. But if we know that this is the last commit before we close the consumer, or before a rebalance, we want to make extra sure that the commit succeeds.

Therefore, a common pattern is to combine `commitAsync()` with `commitSync()` just before shutdown. Here is how it works (we will discuss how to commit just before rebalance when we get to the section about rebalance listeners):

```
Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(),
                record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
    consumer.commitSync(); ❷
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
        consumer.close();
}
```

❶ While everything is fine, we use `commitAsync`. It is faster, and if one commit fails, the next commit will serve as a retry.

❷ But if we are closing, there is no "next commit." We call `commitSync()`, because it will retry until it succeeds or suffers unrecoverable failure.

Committing a Specified Offset

Committing the latest offset only allows you to commit as often as you finish processing batches. But what if you want to commit more frequently than that? What if `poll()` returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occurs? You can't just call `commitSync()` or `commitAsync()`—this will commit the last offset returned, which you didn't get to process yet.

Fortunately, the Consumer API allows you to call `commitSync()` and `commitAsync()` and pass a map of partitions and offsets that you wish to commit. If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic "customers" has offset 5000, you can call `commitSync()` to commit offset 5001 for partition 3 in topic "customers." Since your consumer may be consuming more than a single partition, you will need to track offsets on all of them, which adds complexity to your code.

Here is what a commit of specific offsets looks like:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value()); ❷
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0)    ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ This is the map we will use to manually track offsets.

❷ Remember, `println` is a stand-in for whatever processing you do for the records you consume.

❸ After reading each record, we update the offsets map with the offset of the next message we expect to process. The committed offset should always be the offset of the next message that your application will read. This is where we'll start reading next time we start.

❹ Here, we decide to commit current offsets every 1,000 records. In your application, you can commit based on time or perhaps content of the records.

❺ I chose to call `commitAsync()` (without a callback, therefore the second parameter is `null`), but `commitSync()` is also completely valid here. Of course, when committing specific offsets you still need to perform all the error handling we've seen in previous sections.

Rebalance Listeners

As we mentioned in the previous section about committing offsets, a consumer will want to do some cleanup work before exiting and also before partition rebalancing.

If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed. Perhaps you also need to close file handles, database connections, and such.

The Consumer API allows you to run your own code when partitions are added or removed from the consumer. You do this by passing a `ConsumerRebalanceListener` when calling the `subscribe()` method we discussed previously. `ConsumerRebalanceListener` has three methods you can implement:

`public void onPartitionsAssigned(Collection<TopicPartition> partitions)`
: Called after partitions have been reassigned to the consumer but before the consumer starts consuming messages. This is where you prepare or load any state that you want to use with the partition, seek to the correct offsets if needed, or similar. Any preparation done here should be guaranteed to return within `max.poll.timeout.ms` so the consumer can successfully join the group.

`public void onPartitionsRevoked(Collection<TopicPartition> partitions)`
: Called when the consumer has to give up partitions that it previously owned—either as a result of a rebalance or when the consumer is being closed. In the common case, when an eager rebalancing algorithm is used, this method is invoked before the rebalancing starts and after the consumer stopped consuming messages. If a cooperative rebalancing algorithm is used, this method is invoked at the end of the rebalance, with just the subset of partitions that the consumer has to give up. This is where you want to commit offsets, so whoever gets this partition next will know where to start.

`public void onPartitionsLost(Collection<TopicPartition> partitions)`

Only called when a cooperative rebalancing algorithm is used, and only in exceptional cases where the partitions were assigned to other consumers without first being revoked by the rebalance algorithm (in normal cases, `onPartitionsRevoked()` will be called). This is where you clean up any state or resources that are used with these partitions. Note that this has to be done carefully—the new owner of the partitions may have already saved its own state, and you'll need to avoid conflicts. Note that if you don't implement this method, `onPartitionsRevoked()` will be called instead.

If you use a cooperative rebalancing algorithm, note that:

- `onPartitionsAssigned()` will be invoked on every rebalance, as a way of notifying the consumer that a rebalance happened. However, if there are no new partitions assigned to the consumer, it will be called with an empty collection.
- `onPartitionsRevoked()` will be invoked in normal rebalancing conditions, but only if the consumer gave up the ownership of partitions. It will not be called with an empty collection.
- `onPartitionsLost()` will be invoked in exceptional rebalancing conditions, and the partitions in the collection will already have new owners by the time the method is invoked.

If you implemented all three methods, you are guaranteed that during a normal rebalance, `onPartitionsAssigned()` will be called by the new owner of the partitions that are reassigned only after the previous owner completed `onPartitionsRevoked()` and gave up its ownership.

This example will show how to use `onPartitionsRevoked()` to commit offsets before losing ownership of a partition:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>();
Duration timeout = Duration.ofMillis(100);

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Lost partitions in rebalance. " +
            "Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
```

```
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                 customer = %s, country = %s\n",
                 record.topic(), record.partition(), record.offset(),
                 record.key(), record.value());
             currentOffsets.put(
                 new TopicPartition(record.topic(), record.partition()),
                 new OffsetAndMetadata(record.offset()+1, null));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // ignore, we're closing
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}
```

❶ We start by implementing a `ConsumerRebalanceListener`.

❷ In this example we don't need to do anything when we get a new partition; we'll just start consuming messages.

❸ However, when we are about to lose a partition due to rebalancing, we need to commit offsets. We are committing offsets for all partitions, not just the partitions we are about to lose—because the offsets are for events that were already processed, there is no harm in that. And we are using `commitSync()` to make sure the offsets are committed before the rebalance proceeds.

❹ The most important part: pass the `ConsumerRebalanceListener` to the `subscribe()` method so it will get invoked by the consumer.

Consuming Records with Specific Offsets

So far we've seen how to use `poll()` to start consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence. However, sometimes you want to start reading at a different offset. Kafka offers a variety of methods that cause the next `poll()` to start consuming in a different offset.

If you want to start reading all messages from the beginning of the partition, or you want to skip all the way to the end of the partition and start consuming only new messages, there are APIs specifically for that: `seekToBeginning(Collection<TopicPartition> tp)` and `seekToEnd(Collection<TopicPartition> tp)`.

The Kafka API also lets you seek a specific offset. This ability can be used in a variety of ways; for example, a time-sensitive application could skip ahead a few records when falling behind, or a consumer that writes data to a file could be reset back to a specific point in time in order to recover data if the file was lost.

Here's a quick example of how to set the current offset on all partitions to records that were produced at a specific point in time:

```
Long oneHourEarlier = Instant.now().atZone(ZoneId.systemDefault())
          .minusHours(1).toEpochSecond();
Map<TopicPartition, Long> partitionTimestampMap = consumer.assignment()
        .stream()
        .collect(Collectors.toMap(tp -> tp, tp -> oneHourEarlier)); ❶
Map<TopicPartition, OffsetAndTimestamp> offsetMap
        = consumer.offsetsForTimes(partitionTimestampMap); ❷

for(Map.Entry<TopicPartition,OffsetAndTimestamp> entry: offsetMap.entrySet()) {
    consumer.seek(entry.getKey(), entry.getValue().offset()); ❸
}
```

❶ We create a map from all the partitions assigned to this consumer (via `consumer.assignment()`) to the timestamp we wanted to revert the consumers to.

❷ Then we get the offsets that were current at these timestamps. This method sends a request to the broker where a timestamp index is used to return the relevant offsets.

❸ Finally, we reset the offset on each partition to the offset that was returned in the previous step.

But How Do We Exit?

Earlier in this chapter, when we discussed the poll loop, we told you not to worry about the fact that the consumer polls in an infinite loop, and that we would discuss how to exit the loop cleanly. So, let's discuss how to exit cleanly.

When you decide to shut down the consumer, and you want to exit immediately even though the consumer may be waiting on a long `poll()`, you will need another thread to call `consumer.wakeup()`. If you are running the consumer loop in the main thread, this can be done from `ShutdownHook`. Note that `consumer.wakeup()` is the only consumer method that is safe to call from a different thread. Calling `wakeup` will cause `poll()` to exit with `WakeupException`, or if `consumer.wakeup()` was called while the thread was not waiting on poll, the exception will be thrown on the next iteration when `poll()` is called. The `WakeupException` doesn't need to be handled, but before exiting the thread, you must call `consumer.close()`. Closing the consumer will commit offsets if needed and will send the group coordinator a message that the consumer is leaving the group. The consumer coordinator will trigger rebalancing immediately, and you won't need to wait for the session to timeout before partitions from the consumer you are closing will be assigned to another consumer in the group.

Here is what the exit code will look like if the consumer is running in the main application thread. This example is a bit truncated, but you can view the full example on GitHub:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...
Duration timeout = Duration.ofMillis(10000); ❷

try {
    // looping until ctrl-c, the shutdown hook will cleanup on exit
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "--  waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
```

```
                System.out.printf("offset = %d, key = %s, value = %s\n",
                    record.offset(), record.key(), record.value());
            }
            for (TopicPartition tp: consumer.assignment())
                System.out.println("Committing offset at position:" +
                    consumer.position(tp));
            movingAvg.consumer.commitSync();
        }
    } catch (WakeupException e) {
        // ignore for shutdown ❸
    } finally {
        consumer.close(); ❹
        System.out.println("Closed consumer and we are done");
    }
```

❶ `ShutdownHook` runs in a separate thread, so the only safe action you can take is to call `wakeup` to break out of the `poll` loop.

❷ A particularly long poll timeout. If the poll loop is short enough and you don't mind waiting a bit before exiting, you don't need to call `wakeup`—just checking an atomic boolean in each iteration would be enough. Long poll timeouts are useful when consuming low-throughput topics; this way, the client uses less CPU for constantly looping while the broker has no new data to return.

❸ Another thread calling `wakeup` will cause poll to throw a `WakeupException`. You'll want to catch the exception to make sure your application doesn't exit unexpectedly, but there is no need to do anything with it.

❹ Before exiting the consumer, make sure you close it cleanly.

Deserializers

As discussed in the previous chapter, Kafka producers require *serializers* to convert objects into byte arrays that are then sent to Kafka. Similarly, Kafka consumers require *deserializers* to convert byte arrays received from Kafka into Java objects. In previous examples, we just assumed that both the key and the value of each message are strings, and we used the default `StringDeserializer` in the consumer configuration.

In Chapter 3 about the Kafka producer, we saw how to serialize custom types and how to use Avro and `AvroSerializers` to generate Avro objects from schema definitions and then serialize them when producing messages to Kafka. We will now look at how to create custom deserializers for your own objects and how to use Avro and its deserializers.

It should be obvious that the serializer used to produce events to Kafka must match the deserializer that will be used when consuming events. Serializing with `IntSerializer` and then deserializing with `StringDeserializer` will not end well. This means that, as a developer, you need to keep track of which serializers were used to write into each topic and make sure each topic only contains data that the deserializers you use can interpret. This is one of the benefits of using Avro and the Schema Registry for serializing and deserializing—the `AvroSerializer` can make sure that all the data written to a specific topic is compatible with the schema of the topic, which means it can be deserialized with the matching deserializer and schema. Any errors in compatibility—on the producer or the consumer side—will be caught easily with an appropriate error message, which means you will not need to try to debug byte arrays for serialization errors.

We will start by quickly showing how to write a custom deserializer, even though this is the less common method, and then we will move on to an example of how to use Avro to deserialize message keys and values.

Custom Deserializers

Let's take the same custom object we serialized in Chapter 3 and write a deserializer for it:

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

The custom deserializer will look as follows:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> { ❶
```

```
    @Override
    public void configure(Map configs, boolean isKey) {
        // nothing to configure
    }

    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
        String name;

        try {
            if (data == null)
                return null;
            if (data.length < 8)
                throw new SerializationException("Size of data received " +
                    "by deserializer is shorter than expected");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, "UTF-8");

            return new Customer(id, name); ❷

        } catch (Exception e) {
           throw new SerializationException("Error when deserializing " +
                "byte[] to Customer " + e);
        }
    }

    @Override
    public void close() {
        // nothing to close
    }
}
```

❶ The consumer also needs the implementation of the `Customer` class, and both the class and the serializer need to match on the producing and consuming applications. In a large organization with many consumers and producers sharing access to the data, this can become challenging.

❷ We are just reversing the logic of the serializer here—we get the customer ID and name out of the byte array and use them to construct the object we need.

The consumer code that uses this deserializer will look similar to this example:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    CustomerDeserializer.class.getName());


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

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout);
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("current customer Id: " +
            record.value().getID() + " and
            current customer name: " +  record.value().getName());
    }
    consumer.commitSync();
}
```

Again, it is important to note that implementing a custom serializer and deserializer is not recommended. It tightly couples producers and consumers and is fragile and error prone. A better solution would be to use a standard message format, such as JSON, Thrift, Protobuf, or Avro. We'll now see how to use Avro deserializers with the Kafka consumer. For background on Apache Avro, its schemas, and schema-compatibility capabilities, refer back to Chapter 3.

Using Avro Deserialization with Kafka Consumer

Let's assume we are using the implementation of the `Customer` class in Avro that was shown in Chapter 3. In order to consume those objects from Kafka, you want to implement a consuming application similar to this:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader","true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"
```

```
KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
            record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ We use `KafkaAvroDeserializer` to deserialize the Avro messages.

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas. This way, the consumer can use the schema that was registered by the producer to deserialize the message.

❸ We specify the generated class, `Customer`, as the type for the record value.

❹ `record.value()` is a `Customer` instance, and we can use it accordingly.

Standalone Consumer: Why and How to Use a Consumer Without a Group

So far, we have discussed consumer groups, which are where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case, there is no reason for groups or rebalances—just assign the consumer-specific topic and/or partitions, consume messages, and commit offsets on occasion (although you still need to configure `group.id` to commit offsets, without calling *subscribe* the consumer won't join any group).

When you know exactly which partitions the consumer should read, you don't *subscribe* to a topic—instead, you *assign* yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group) or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```
Duration timeout = Duration.ofMillis(100);
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

❶ We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.

❷ Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply by bouncing the application whenever partitions are added.

Summary

We started this chapter with an in-depth explanation of Kafka's consumer groups and the way they allow multiple consumers to share the work of reading events from topics. We followed the theoretical discussion with a practical example of a consumer subscribing to a topic and continuously reading events. We then looked into the most important consumer configuration parameters and how they affect consumer behavior. We dedicated a large part of the chapter to discussing offsets and how consumers keep track of them. Understanding how consumers commit offsets is critical when writing reliable consumers, so we took time to explain the different ways this can be done. We then discussed additional parts of the Consumer APIs, handling rebalances, and closing the consumer.

We concluded by discussing the deserializers used by consumers to turn bytes stored in Kafka into Java objects that the applications can process. We discussed Avro deserializers in some detail, even though they are just one type of deserializer you can use, because these are most commonly used with Kafka.

CHAPTER 5

Managing Apache Kafka Programmatically

There are many CLI and GUI tools for managing Kafka (we'll discuss them in Chapter 9), but there are also times when you want to execute some administrative commands from within your client application. Creating new topics on demand based on user input or data is an especially common use case: Internet of Things (IoT) apps often receive events from user devices, and write events to topics based on the device type. If the manufacturer produces a new type of device, you either have to remember, via some process, to also create a topic, or the application can dynamically create a new topic if it receives events with an unrecognized device type. The second alternative has downsides, but avoiding the dependency on an additional process to generate topics is an attractive feature in the right scenarios.

Apache Kafka added the AdminClient in version 0.11 to provide a programmatic API for administrative functionality that was previously done in the command line: listing, creating, and deleting topics; describing the cluster; managing ACLs; and modifying configuration.

Here's one example. Your application is going to produce events to a specific topic. This means that before producing the first event, the topic has to exist. Before Apache Kafka added the AdminClient, there were few options, none of them particularly user-friendly: you could capture an `UNKNOWN_TOPIC_OR_PARTITION` exception from the `producer.send()` method and let your user know that they needed to create the topic, or you could hope that the Kafka cluster you were writing to enabled automatic topic creation, or you could try to rely on internal APIs and deal with the consequences of no compatibility guarantees. Now that Apache Kafka provides AdminClient, there is a much better solution: use AdminClient to check whether the topic exists, and if it does not, create it on the spot.

In this chapter we'll give an overview of the AdminClient before we drill down into the details of how to use it in your applications. We'll focus on the most commonly used functionality: management of topics, consumer groups, and entity configuration.

AdminClient Overview

As you start using Kafka AdminClient, it helps to be aware of its core design principles. When you understand how the AdminClient was designed and how it should be used, the specifics of each method will be much more intuitive.

Asynchronous and Eventually Consistent API

Perhaps the most important thing to understand about Kafka's AdminClient is that it is asynchronous. Each method returns immediately after delivering a request to the cluster controller, and each method returns one or more `Future` objects. `Future` objects are the result of asynchronous operations, and they have methods for checking the status of the asynchronous operation, canceling it, waiting for it to complete, and executing functions after its completion. Kafka's AdminClient wraps the `Future` objects into `Result` objects, which provide methods to wait for the operation to complete and helper methods for common follow-up operations. For example, `Kafka AdminClient.createTopics` returns the `CreateTopicsResult` object, which lets you wait until all topics are created, check each topic status individually, and retrieve the configuration of a specific topic after it was created.

Because Kafka's propagation of metadata from the controller to the brokers is asynchronous, the `Futures` that AdminClient APIs return are considered complete when the controller state has been fully updated. At that point, not every broker might be aware of the new state, so a `listTopics` request may end up handled by a broker that is not up-to-date and will not contain a topic that was very recently created. This property is also called *eventual consistency*: eventually every broker will know about every topic, but we can't guarantee exactly when this will happen.

Options

Every method in AdminClient takes as an argument an `Options` object that is specific to that method. For example, the `listTopics` method takes the `ListTopicsOptions` object as an argument, and `describeCluster` takes `DescribeClusterOptions` as an argument. Those objects contain different settings for how the request will be handled by the broker. The one setting that all AdminClient methods have is `timeoutMs`: this controls how long the client will wait for a response from the cluster before throwing a `TimeoutException`. This limits the time in which your application may be blocked by AdminClient operation. Other options include whether `listTopics`

should also return internal topics and whether `describeCluster` should also return which operations the client is authorized to perform on the cluster.

Flat Hierarchy

All admin operations supported by the Apache Kafka protocol are implemented in `KafkaAdminClient` directly. There is no object hierarchy or namespaces. This is a bit controversial as the interface can be quite large and perhaps a bit overwhelming, but the main benefit is that if you want to know how to programmatically perform any admin operation on Kafka, you have exactly one JavaDoc to search, and your IDE autocomplete will be quite handy. You don't have to wonder whether you are just missing the right place to look. If it isn't in AdminClient, it was not implemented yet (but contributions are welcome!).

If you are interested in contributing to Apache Kafka, take a look at our "How to Contribute" guide. Start with smaller, noncontroversial bug fixes and improvements before tackling a more significant change to the architecture or the protocol. Noncode contributions such as bug reports, documentation improvements, responses to questions, and blog posts are also encouraged.

Additional Notes

All the operations that modify the cluster state—create, delete, and alter—are handled by the controller. Operations that read the cluster state—list and describe—can be handled by any broker and are directed to the least-loaded broker (based on what the client knows). This shouldn't impact you as an API user, but it can be good to know in case you are seeing unexpected behavior, you notice that some operations succeed while others fail, or if you are trying to figure out why an operation is taking too long.

At the time we are writing this chapter (Apache Kafka 2.5 is about to be released), most admin operations can be performed either through AdminClient or directly by modifying the cluster metadata in ZooKeeper. We highly encourage you to never use ZooKeeper directly, and if you absolutely have to, report this as a bug to Apache Kafka. The reason is that in the near future, the Apache Kafka community will remove the ZooKeeper dependency, and every application that uses ZooKeeper directly for admin operations will have to be modified. On the other hand, the AdminClient API will remain exactly the same, just with a different implementation inside the Kafka cluster.

AdminClient Lifecycle: Creating, Configuring, and Closing

To use Kafka's AdminClient, the first thing you have to do is construct an instance of the AdminClient class. This is quite straightforward:

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
AdminClient admin = AdminClient.create(props);
// TODO: Do something useful with AdminClient
admin.close(Duration.ofSeconds(30));
```

The static `create` method takes as an argument a `Properties` object with configuration. The only mandatory configuration is the URI for your cluster: a comma-separated list of brokers to connect to. As usual, in production environments, you want to specify at least three brokers just in case one is currently unavailable. We'll discuss how to configure a secure and authenticated connection separately in Chapter 11.

If you start an AdminClient, eventually you want to close it. It is important to remember that when you call `close`, there could still be some AdminClient operations in progress. Therefore, the `close` method accepts a timeout parameter. Once you call `close`, you can't call any other methods and send any more requests, but the client will wait for responses until the timeout expires. After the timeout expires, the client will abort all ongoing operations with timeout exception and release all resources. Calling `close` without a timeout implies that the client will wait as long as it takes for all ongoing operations to complete.

You probably recall from Chapters 3 and 4 that the `KafkaProducer` and `KafkaConsumer` have quite a few important configuration parameters. The good news is that AdminClient is much simpler, and there is not much to configure. You can read about all the configuration parameters in the Kafka documentation. In our opinion, the important configuration parameters are described in the following sections.

client.dns.lookup

This configuration was introduced in the Apache Kafka 2.1.0 release.

By default, Kafka validates, resolves, and creates connections based on the hostname provided in the bootstrap server configuration (and later in the names returned by the brokers as specified in the `advertised.listeners` configuration). This simple model works most of the time but fails to cover two important use cases: the use of DNS aliases, especially in a bootstrap configuration, and the use of a single DNS that maps to multiple IP addresses. These sound similar but are slightly different. Let's look at each of these mutually exclusive scenarios in a bit more detail.

Use of a DNS alias

Suppose you have multiple brokers with the following naming convention: `broker1.hostname.com`, `broker2.hostname.com`, etc. Rather than specifying all of them in a bootstrap server configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them.

You'll use `all-brokers.hostname.com` for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate `all-brokers.hostname.com`, but the server principal will be `broker2.hostname.com`. If the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

In this scenario, you'll want to use `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. With this configuration, the client will "expend" the DNS alias, and the result will be the same as if you included all the broker names the DNS alias connects to as brokers in the original bootstrap list.

DNS name with multiple IP addresses

With modern network architectures, it is common to put all the brokers behind a proxy or a load balancer. This is especially common if you use Kubernetes, where load balancers are necessary to allow connections from outside the Kubernetes cluster. In these cases, you don't want the load balancers to become a single point of failure. It is therefore very common to have `broker1.hostname.com` point at a list of IPs, all of which resolve to load balancers, and all of which route traffic to the same broker. These IPs are also likely to change over time. By default, the Kafka client will just try to connect to the first IP that the hostname resolves. This means that if that IP becomes unavailable, the client will fail to connect, even though the broker is fully available. It is therefore highly recommended to use `client.dns.lookup=use_all_dns_ips` to make sure the client doesn't miss out on the benefits of a highly available load balancing layer.

request.timeout.ms

This configuration limits the time that your application can spend waiting for AdminClient to respond. This includes the time spent on retrying if the client receives a retriable error.

The default value is 120 seconds, which is quite long, but some AdminClient operations, especially consumer group management commands, can take a while to respond. As we mentioned in "AdminClient Overview" on page 114, each AdminClient method accepts an `Options` object, which can contain a timeout value that applies specifically to that call. If an AdminClient operation is on the critical path for your application, you may want to use a lower timeout value and handle a lack of timely response from Kafka in a different way. A common example is that services try to validate the existence of specific topics when they first start, but if Kafka takes longer than 30 seconds to respond, you may want to continue starting the server and validate the existence of topics later (or skip this validation entirely).

Essential Topic Management

Now that we created and configured an AdminClient, it's time to see what we can do with it. The most common use case for Kafka's AdminClient is topic management. This includes listing topics, describing them, creating topics, and deleting them.

Let's start by listing all topics in the cluster:

```
ListTopicsResult topics = admin.listTopics();
topics.names().get().forEach(System.out::println);
```

Note that `admin.listTopics()` returns the `ListTopicsResult` object, which is a thin wrapper over a collection of `Futures`. Note also that `topics.name()` returns a `Future` set of `name`. When we call `get()` on this `Future`, the executing thread will wait until the server responds with a set of topic names, or we get a timeout exception. Once we get the list, we iterate over it to print all the topic names.

Now let's try something a bit more ambitious: check if a topic exists, and create it if it doesn't. One way to check if a specific topic exists is to get a list of all topics and check if the topic you need is in the list. On a large cluster, this can be inefficient. In addition, sometimes you want to check for more than just whether the topic exists—you want to make sure the topic has the right number of partitions and replicas. For example, Kafka Connect and Confluent Schema Registry use a Kafka topic to store configuration. When they start up, they check if the configuration topic exists, that it has only one partition to guarantee that configuration changes will arrive in strict order, that it has three replicas to guarantee availability, and that the topic is compacted so the old configuration will be retained indefinitely:

```
DescribeTopicsResult demoTopic = admin.describeTopics(TOPIC_LIST); ❶

try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get(); ❷
    System.out.println("Description of demo topic:" + topicDescription);

    if (topicDescription.partitions().size() != NUM_PARTITIONS) { ❸
      System.out.println("Topic has wrong number of partitions. Exiting.");
      System.exit(-1);
    }
} catch (ExecutionException e) { ❹
    // exit early for almost all exceptions
    if (! (e.getCause() instanceof UnknownTopicOrPartitionException)) {
        e.printStackTrace();
        throw e;
    }

    // if we are here, topic doesn't exist
    System.out.println("Topic " + TOPIC_NAME +
        " does not exist. Going to create it now");
    // Note that number of partitions and replicas is optional. If they are
```

```
        // not specified, the defaults configured on the Kafka brokers will be used
        CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(
                new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❺

        // Check that the topic was created correctly:
        if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ❻
            System.out.println("Topic has wrong number of partitions.");
            System.exit(-1);
        }
    }
```

❶ To check that the topic exists with the correct configuration, we call `describeTopics()` with a list of topic names we want to validate. This returns `DescribeTopicResult` object, which wraps a map of topic names to `Future` descriptions.

❷ We've already seen that if we wait for the `Future` to complete, using `get()` we can get the result we wanted, in this case, a `TopicDescription`. But there is also a possibility that the server can't complete the request correctly—if the topic does not exist, the server can't respond with its description. In this case, the server will send back an error, and the `Future` will complete by throwing an `ExecutionException`. The actual error sent by the server will be the `cause` of the exception. Since we want to handle the case where the topic doesn't exist, we handle these exceptions.

❸ If the topic does exist, the `Future` completes by returning a `TopicDescription`, which contains a list of all the partitions of the topic, and for each partition in which a broker is the leader, a list of replicas and a list of in-sync replicas. Note that this does not include the configuration of the topic. We'll discuss configuration later in this chapter.

❹ Note that all AdminClient result objects throw `ExecutionException` when Kafka responds with an error. This is because AdminClient results are wrapped `Future` objects, and those wrap exceptions. You always need to examine the cause of `ExecutionException` to get the error that Kafka returned.

❺ If the topic does not exist, we create a new topic. When creating a topic, you can specify just the name and use default values for all the details. You can also specify the number of partitions, number of replicas, and the configuration.

❻ Finally, you want to wait for topic creation to return, and perhaps validate the result. In this example, we are checking the number of partitions. Since we specified the number of partitions when we created the topic, we are fairly certain it is correct. Checking the result is more common if you relied on broker defaults when creating the topic. Note that since we are again calling `get()` to check the

results of `CreateTopic`, this method could throw an exception. `TopicExistsException` is common in this scenario, and you'll want to handle it (perhaps by describing the topic to check for the correct configuration).

Now that we have a topic, let's delete it:

```
admin.deleteTopics(TOPIC_LIST).all().get();

// Check that it is gone. Note that due to the async nature of deletes,
// it is possible that at this point the topic still exists
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get();
    System.out.println("Topic " + TOPIC_NAME + " is still around");
} catch (ExecutionException e) {
    System.out.println("Topic " + TOPIC_NAME + " is gone");
}
```

At this point the code should be quite familiar. We call the method `deleteTopics` with a list of topic names to delete, and we use `get()` to wait for this to complete.

Although the code is simple, please remember that in Kafka, deletion of topics is final—there is no recycle bin or trash can to help you rescue the deleted topic, and no checks to validate that the topic is empty and that you really meant to delete it. Deleting the wrong topic could mean unrecoverable loss of data, so handle this method with extra care.

All the examples so far have used the blocking `get()` call on the `Future` returned by the different `AdminClient` methods. Most of the time, this is all you need—admin operations are rare, and waiting until the operation succeeds or times out is usually acceptable. There is one exception: if you are writing to a server that is expected to process a large number of admin requests. In this case, you don't want to block the server threads while waiting for Kafka to respond. You want to continue accepting requests from your users and sending them to Kafka, and when Kafka responds, send the response to the client. In these scenarios, the versatility of `KafkaFuture` becomes quite useful. Here's a simple example.

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);

    DescribeTopicsResult demoTopic = admin.describeTopics( ❸
            Collections.singletonList(topic),
            new DescribeTopicsOptions().timeoutMs(timeoutMs));

    demoTopic.values().get(topic).whenComplete( ❹
            new KafkaFuture.BiConsumer<TopicDescription, Throwable>() {
```

```
                @Override
                public void accept(final TopicDescription topicDescription,
                                   final Throwable throwable) {
                    if (throwable != null) {
                      request.response().end("Error trying to describe topic "
                              + topic + " due to " + throwable.getMessage()); ❺
                    } else {
                        request.response().end(topicDescription.toString()); ❻
                    }
                }
            });
}).listen(8080);
```

❶ We are using Vert.x to create a simple HTTP server. Whenever this server receives a request, it calls the `requestHandler` that we are defining here.

❷ The request includes a topic name as a parameter, and we'll respond with a description of this topic.

❸ We call `AdminClient.describeTopics` as usual and get a wrapped `Future` in response.

❹ Instead of using the blocking `get()` call, we construct a function that will be called when the `Future` completes.

❺ If the `Future` completes with an exception, we send the error to the HTTP client.

❻ If the `Future` completes successfully, we respond to the client with the topic description.

The key here is that we are not waiting for a response from Kafka. `DescribeTopicResult` will send the response to the HTTP client when a response arrives from Kafka. Meanwhile, the HTTP server can continue processing other requests. You can check this behavior by using `SIGSTOP` to pause Kafka (don't try this in production!) and send two HTTP requests to Vert.x: one with a long timeout value and one with a short value. Even though you sent the second request after the first, it will respond earlier thanks to the lower timeout value, and not block behind the first request.

Configuration Management

Configuration management is done by describing and updating collections of `ConfigResource`. Config resources can be brokers, broker loggers, and topics. Checking and modifying broker and broker logging configuration is typically done using tools like `kafka-config.sh` or other Kafka management tools, but checking and updating topic configuration from the applications that use them is quite common.

For example, many applications rely on compacted topics for correct operation. It makes sense that periodically (more frequently than the default retention period, just to be safe), those applications will check that the topic is indeed compacted and take action to correct the topic configuration if it is not.

Here's an example of how this is done:

```
ConfigResource configResource =
        new ConfigResource(ConfigResource.Type.TOPIC, TOPIC_NAME); ❶
DescribeConfigsResult configsResult =
        admin.describeConfigs(Collections.singleton(configResource));
Config configs = configsResult.all().get().get(configResource);

// print nondefault configs
configs.entries().stream().filter(
        entry -> !entry.isDefault()).forEach(System.out::println); ❷


// Check if topic is compacted
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
        TopicConfig.CLEANUP_POLICY_COMPACT);
if (!configs.entries().contains(compaction)) {
    // if topic is not compacted, compact it
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET)); ❸
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Topic " + TOPIC_NAME + " is compacted topic");
}
```

❶ As mentioned above, there are several types of `ConfigResource`; here we are checking the configuration for a specific topic. You can specify multiple different resources from different types in the same request.

❷ The result of `describeConfigs` is a map from each `ConfigResource` to a collection of configurations. Each configuration entry has an `isDefault()` method that lets us know which configs were modified. A topic configuration is considered nondefault if a user configured the topic to have a nondefault value, or if a broker-level configuration was modified and the topic that was created inherited this nondefault value from the broker.

❸ To modify a configuration, specify a map of the `ConfigResource` you want to modify and a collection of operations. Each configuration modifying operation consists of a configuration entry (the name and value of the configuration; in this case, `cleanup.policy` is the configuration name and `compacted` is the value) and the operation type. Four types of operations modify configuration in Kafka: `SET`,

which sets the configuration value; `DELETE`, which removes the value and resets to the default; `APPEND`; and `SUBSTRACT`. The last two apply only to configurations with a `List` type and allow adding and removing values from the list without having to send the entire list to Kafka every time.

Describing the configuration can be surprisingly handy in an emergency. We remember a time when during an upgrade, the configuration file for the brokers was accidentally replaced with a broken copy. This was discovered after restarting the first broker and noticing that it failed to start. The team did not have a way to recover the original, and we prepared for significant trial and error as we attempted to reconstruct the correct configuration and bring the broker back to life. A site reliability engineer (SRE) saved the day by connecting to one of the remaining brokers and dumping its configuration using the AdminClient.

Consumer Group Management

We've mentioned before that unlike most message queues, Kafka allows you to reprocess data in the exact order in which it was consumed and processed earlier. In Chapter 4, where we discussed consumer groups, we explained how to use the Consumer APIs to go back and reread older messages from a topic. But using these APIs means that you programmed the ability to reprocess data in advance into your application. Your application itself must expose the "reprocess" functionality.

There are several scenarios in which you'll want to cause an application to reprocess messages, even if this capability was not built into the application in advance. Troubleshooting a malfunctioning application during an incident is one such scenario. Another is when preparing an application to start running on a new cluster during a disaster recovery failover scenario (we'll discuss this in more detail in Chapter 9, when we discuss disaster recovery techniques).

In this section, we'll look at how you can use the AdminClient to programmatically explore and modify consumer groups and the offsets that were committed by those groups. In Chapter 10 we'll look at external tools available to perform the same operations.

Exploring Consumer Groups

If you want to explore and modify consumer groups, the first step is to list them:

```
admin.listConsumerGroups().valid().get().forEach(System.out::println);
```

Note that by using `valid()` method, the collection that `get()` will return will only contain the consumer groups that the cluster returned without errors, if any. Any errors will be completely ignored, rather than thrown as exceptions. The `errors()` method can be used to get all the exceptions. If you use `all()` as we did in other

examples, only the first error the cluster returned will be thrown as an exception. Likely causes of such errors are authorization, where you don't have permission to view the group, or cases when the coordinator for some of the consumer groups is not available.

If we want more information about some of the groups, we can describe them:

```
ConsumerGroupDescription groupDescription = admin
        .describeConsumerGroups(CONSUMER_GRP_LIST)
        .describedGroups().get(CONSUMER_GROUP).get();
        System.out.println("Description of group " + CONSUMER_GROUP
                + ":" + groupDescription);
```

The description contains a wealth of information about the group. This includes the group members, their identifiers and hosts, the partitions assigned to them, the algorithm used for the assignment, and the host of the group coordinator. This description is very useful when troubleshooting consumer groups. One of the most important pieces of information about a consumer group is missing from this description—inevitably, we'll want to know what was the last offset committed by the group for each partition that it is consuming and how much it is lagging behind the latest messages in the log.

In the past, the only way to get this information was to parse the commit messages that the consumer groups wrote to an internal Kafka topic. While this method accomplished its intent, Kafka does not guarantee compatibility of the internal message formats, and therefore the old method is not recommended. We'll take a look at how Kafka's AdminClient allows us to retrieve this information:

```
Map<TopicPartition, OffsetAndMetadata> offsets =
        admin.listConsumerGroupOffsets(CONSUMER_GROUP)
                .partitionsToOffsetAndMetadata().get(); ❶

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new HashMap<>();

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ❷
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> latestOffsets =
        admin.listOffsets(requestLatestOffsets).all().get();

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ❸
    String topic = e.getKey().topic();
    int partition =  e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offset();

    System.out.println("Consumer group " + CONSUMER_GROUP
            + " has committed offset " + committedOffset
            + " to topic " + topic + " partition " + partition
```

```
                + ". The latest offset in the partition is "
                +  latestOffset + " so consumer group is "
                + (latestOffset - committedOffset) + " records behind");
    }
```

❶ We retrieve a map of all topics and partitions that the consumer group handles, and the latest committed offset for each. Note that unlike `describeConsumerGroups`, `listConsumerGroupOffsets` only accepts a single consumer group and not a collection.

❷ For each topic and partition in the results, we want to get the offset of the last message in the partition. `OffsetSpec` has three very convenient implementations: `earliest()`, `latest()`, and `forTimestamp()`, which allow us to get the earlier and latest offsets in the partition, as well as the offset of the record written on or immediately after the time specified.

❸ Finally, we iterate over all the partitions, and for each partition print the last committed offset, the latest offset in the partition, and the lag between them.

Modifying Consumer Groups

Until now, we just explored available information. AdminClient also has methods for modifying consumer groups: deleting groups, removing members, deleting committed offsets, and modifying offsets. These are commonly used by SREs to build ad hoc tooling to recover from an emergency.

From all those, modifying offsets is the most useful. Deleting offsets might seem like a simple way to get a consumer to "start from scratch," but this really depends on the configuration of the consumer—if the consumer starts and no offsets are found, will it start from the beginning? Or jump to the latest message? Unless we have the value of `auto.offset.reset`, we can't know. Explicitly modifying the committed offsets to the earliest available offsets will force the consumer to start processing from the beginning of the topic, and essentially cause the consumer to "reset."

Do keep in mind that consumer groups don't receive updates when offsets change in the offset topic. They only read offsets when a consumer is assigned a new partition or on startup. To prevent you from making changes to offsets that the consumers will not know about (and will therefore override), Kafka will prevent you from modifying offsets while the consumer group is active.

Also keep in mind that if the consumer application maintains state (and most stream processing applications maintain state), resetting the offsets and causing the consumer group to start processing from the beginning of the topic can have a strange impact on the stored state. For example, suppose you have a stream application that is continuously counting shoes sold in your store, and suppose that at 8:00 a.m. you

discover that there was an error in inputs and you want to completely recalculate the count since 3:00 a.m. If you reset the offsets to 3:00 a.m. without appropriately modifying the stored aggregate, you will count every shoe that was sold today twice (you will also process all the data between 3:00 a.m. and 8:00 a.m., but let's assume that this is necessary to correct the error). You need to take care to update the stored state accordingly. In a development environment, we usually delete the state store completely before resetting the offsets to the start of the input topic.

With all these warnings in mind, let's look at an example:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> earliestOffsets =
    admin.listOffsets(requestEarliestOffsets).all().get(); ❶

Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:
        earliestOffsets.entrySet()) {
  resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ❷
}

try {
  admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ❸
} catch (ExecutionException e) {
  System.out.println("Failed to update the offsets committed by group "
            + CONSUMER_GROUP + " with error " + e.getMessage());
  if (e.getCause() instanceof UnknownMemberIdException)
      System.out.println("Check if consumer group is still active."); ❹
}
```

❶ To reset the consumer group so it will start processing from the earliest offset, we need to get the earliest offsets first. Getting the earliest offsets is similar to getting the latest, shown in the previous example.

❷ In this loop we convert the map with `ListOffsetsResultInfo` values that were returned by `listOffsets` into a map with `OffsetAndMetadata` values that are required by `alterConsumerGroupOffsets`.

❸ After calling `alterConsumerGroupOffsets`, we are waiting on the `Future` to complete so we can see if it completed successfully.

❹ One of the most common reasons that `alterConsumerGroupOffsets` fails is that we didn't stop the consumer group first (this has to be done by shutting down the consuming application directly; there is no admin command for shutting down a consumer group). If the group is still active, our attempt to modify the offsets will appear to the consumer coordinator as if a client that is not a member of the group is committing an offset for that group. In this case, we'll get `UnknownMemberIdException`.

Cluster Metadata

It is rare that an application has to explicitly discover anything at all about the cluster to which it connected. You can produce and consume messages without ever learning how many brokers exist and which one is the controller. Kafka clients abstract away this information—clients only need to be concerned with topics and partitions.

But just in case you are curious, this little snippet will satisfy your curiosity:

```
DescribeClusterResult cluster = admin.describeCluster();

System.out.println("Connected to cluster " + cluster.clusterId().get()); ❶
System.out.println("The brokers in the cluster are:");
cluster.nodes().get().forEach(node -> System.out.println("    * " + node));
System.out.println("The controller is: " + cluster.controller().get());
```

❶ Cluster identifier is a GUID and therefore is not human readable. It is still useful to check whether your client connected to the correct cluster.

Advanced Admin Operations

In this section, we'll discuss a few methods that are rarely used, and can be risky to use, but are incredibly useful when needed. Those are mostly important for SREs during incidents—but don't wait until you are in an incident to learn how to use them. Read and practice before it is too late. Note that the methods here have little to do with one another, except that they all fit into this category.

Adding Partitions to a Topic

Usually the number of partitions in a topic is set when a topic is created. And since each partition can have very high throughput, bumping against the capacity limits of a topic is rare. In addition, if messages in the topic have keys, then consumers can assume that all messages with the same key will always go to the same partition and will be processed in the same order by the same consumer.

For these reasons, adding partitions to a topic is rarely needed and can be risky. You'll need to check that the operation will not break any application that consumes from the topic. At times, however, you will really hit the ceiling of how much throughput you can process with the existing partitions and have no choice but to add some.

You can add partitions to a collection of topics using the `createPartitions` method. Note that if you try to expand multiple topics at once, it is possible that some of the topics will be successfully expanded, while others will fail.

```
Map<String, NewPartitions> newPartitions = new HashMap<>();
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_PARTITIONS+2)); ❶
admin.createPartitions(newPartitions).all().get();
```

❶ When expanding topics, you need to specify the total number of partitions the topic will have after the partitions are added, not the number of new partitions.

Since the `createPartition` method takes as a parameter the total number of partitions in the topic after new partitions are added, you may need to describe the topic and find out how many partitions exist prior to expanding it.

Deleting Records from a Topic

Current privacy laws mandate specific retention policies for data. Unfortunately, while Kafka has retention policies for topics, they were not implemented in a way that guarantees legal compliance. A topic with a retention policy of 30 days can store older data if all the data fits into a single segment in each partition.

The `deleteRecords` method will mark as deleted all the records with offsets older than those specified when calling the method and make them inaccessible by Kafka consumers. The method returns the highest deleted offsets, so we can check if the deletion indeed happened as expected. Full cleanup from disk will happen asynchronously. Remember that the `listOffsets` method can be used to get offsets for records that were written on or immediately after a specific time. Together, these methods can be used to delete records older than any specific point in time:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> olderOffsets =
        admin.listOffsets(requestOlderOffsets).all().get();
Map<TopicPartition, RecordsToDelete> recordsToDelete = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo>  e:
        olderOffsets.entrySet())
    recordsToDelete.put(e.getKey(),
            RecordsToDelete.beforeOffset(e.getValue().offset()));
 admin.deleteRecords(recordsToDelete).all().get();
```

Leader Election

This method allows you to trigger two different types of leader election:

Preferred leader election
: Each partition has a replica that is designated as the *preferred leader*. It is preferred because if all partitions use their preferred leader replica as the leader, the number of leaders on each broker should be balanced. By default, Kafka will check every five minutes if the preferred leader replica is indeed the leader, and if it isn't but it is eligible to become the leader, it will elect the preferred leader replica as leader. If `auto.leader.rebalance.enable` is `false`, or if you want this to happen faster, the `electLeader()` method can trigger this process.

Unclean leader election
: If the leader replica of a partition becomes unavailable, and the other replicas are not eligible to become leaders (usually because they are missing data), the partition will be without a leader and therefore unavailable. One way to resolve this is to trigger *unclean leader* election, which means electing a replica that is otherwise ineligible to become a leader as the leader anyway. This will cause data loss—all the events that were written to the old leader and were not replicated to the new leader will be lost. The `electLeader()` method can also be used to trigger unclean leader elections.

The method is asynchronous, which means that even after it returns successfully, it takes a while until all brokers become aware of the new state, and calls to `describeTopics()` can return inconsistent results. If you trigger leader election for multiple partitions, it is possible that the operation will be successful for some partitions and fail for others:

```
Set<TopicPartition> electableTopics = new HashSet<>();
electableTopics.add(new TopicPartition(TOPIC_NAME, 0));
try {
    admin.electLeaders(ElectionType.PREFERRED, electableTopics).all().get(); ❶
} catch (ExecutionException e) {
    if (e.getCause() instanceof ElectionNotNeededException) {
        System.out.println("All leaders are preferred already"); ❷
    }
}
```

❶ We are electing the preferred leader on a single partition of a specific topic. We can specify any number of partitions and topics. If you call the command with `null` instead of a collection of partitions, it will trigger the election type you chose for all partitions.

❷ If the cluster is in a healthy state, the command will do nothing. Preferred leader election and unclean leader election only take effect when a replica other than the preferred leader is the current leader.

Reassigning Replicas

Sometimes, you don't like the current location of some of the replicas. Maybe a broker is overloaded and you want to move some replicas. Maybe you want to add more replicas. Maybe you want to move all replicas from a broker so you can remove the machine. Or maybe a few topics are so noisy that you need to isolate them from the rest of the workload. In all these scenarios, `alterPartitionReassignments` gives you fine-grain control over the placement of every single replica for a partition. Keep in mind that reassigning replicas from one broker to another may involve copying large amounts of data from one broker to another. Be mindful of the available

network bandwidth, and throttle replication using quotas if needed; quotas are a broker configuration, so you can describe them and update them with `AdminClient`.

For this example, assume that we have a single broker with ID 0. Our topic has several partitions, all with one replica on this broker. After adding a new broker, we want to use it to store some of the replicas of the topic. We are going to assign each partition in the topic in a slightly different way:

```
Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new Hash-
Map<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
        Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹

admin.alterPartitionReassignments(reassignment).all().get();

System.out.println("currently reassigning: " +
        admin.listPartitionReassignments().reassignments().get()); ❺
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Description of demo topic:" + topicDescription); ❻
```

❶ We've added another replica to partition 0, placed the new replica on the new broker, which has ID 1, but left the leader unchanged.

❷ We didn't add any replicas to partition 1; we simply moved the one existing replica to the new broker. Since we have only one replica, it is also the leader.

❸ We've added another replica to partition 2 and made it the preferred leader. The next preferred leader election will switch leadership to the new replica on the new broker. The existing replica will then become a follower.

❹ There is no ongoing reassignment for partition 3, but if there was, this would have canceled it and returned the state to what it was before the reassignment operation started.

❺ We can list the ongoing reassignments.

❻ We can also print the new state, but remember that it can take awhile until it shows consistent results.

Testing

Apache Kafka provides a test class, `MockAdminClient`, which you can initialize with any number of brokers and use to test that your applications behave correctly without having to run an actual Kafka cluster and really perform the admin operations on it. While `MockAdminClient` is not part of the Kafka API and therefore subject to change without warning, it mocks methods that are public, and therefore the method signatures will remain compatible. There is a bit of a trade-off on whether the convenience of this class is worth the risk that it will change and break your tests, so keep this in mind.

What makes this test class especially compelling is that some of the common methods have very comprehensive mocking: you can create topics with `MockAdminClient`, and a subsequent call to `listTopics()` will list the topics you "created."

However, not all methods are mocked. If you use `AdminClient` with version 2.5 or earlier and call `incrementalAlterConfigs()` of the `MockAdminClient`, you will get an `UnsupportedOperationException`, but you can handle this by injecting your own implementation.

To demonstrate how to test using `MockAdminClient`, let's start by implementing a class that is instantiated with an admin client and uses it to create topics:

```
public TopicCreator(AdminClient admin) {
    this.admin = admin;
}

// Example of a method that will create a topic if its name starts with "test"
public void maybeCreateTopic(String topicName)
        throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
    if (topicName.toLowerCase().startsWith("test")) {
        admin.createTopics(topics);

        // alter configs just to demonstrate a point
        ConfigResource configResource =
                  new ConfigResource(ConfigResource.Type.TOPIC, topicName);
        ConfigEntry compaction =
                  new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
                          TopicConfig.CLEANUP_POLICY_COMPACT);
        Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
        configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET));
        Map<ConfigResource, Collection<AlterConfigOp>> alterConf =
            new HashMap<>();
        alterConf.put(configResource, configOp);
        admin.incrementalAlterConfigs(alterConf).all().get();
    }
}
```

The logic here isn't sophisticated: `maybeCreateTopic` will create the topic if the topic name starts with "test." We are also modifying the topic configuration, so we can show how to handle a case where the method we use isn't implemented in the mock client.

We are using the Mockito testing framework to verify that the `MockAdminClient` methods are called as expected and to fill in for the unimplemented methods. Mockito is a fairly simple mocking framework with nice APIs, which makes it a good fit for a small example of a unit test.

We'll start testing by instantiating our mock client:

```
@Before
public void setUp() {
    Node broker = new Node(0,"localhost",9092);
    this.admin = spy(new MockAdminClient(Collections.singletonList(broker),
        broker)); ❶

    // without this, the tests will throw
    // `java.lang.UnsupportedOperationException: Not implemented yet`
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}
```

❶ `MockAdminClient` is instantiated with a list of brokers (here we're using just one), and one broker that will be our controller. The brokers are just the broker ID, hostname, and port—all fake, of course. No brokers will run while executing these tests. We'll use Mockito's `spy` injection, so we can later check that `TopicCreator` executed correctly.

❷ Here we use Mockito's `doReturn` methods to make sure the mock admin client doesn't throw exceptions. The method we are testing expects the `AlterConfigResult` object with an `all()` method that returns a `KafkaFuture`. We made sure that the fake `incrementalAlterConfigs` returns exactly that.

Now that we have a properly fake AdminClient, we can use it to test whether the `maybeCreateTopic()` method works properly:

```
@Test
public void testCreateTestTopic()
        throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}
```

```
@Test
public void testNotTopic() throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ❷
}
```

❶ The topic name starts with "test," so we expect `maybeCreateTopic()` to create a topic. We check that `createTopics()` was called once.

❷ When the topic name doesn't start with "test," we verify that `createTopics()` was not called at all.

One last note: Apache Kafka published `MockAdminClient` in a test jar, so make sure your *pom.xml* includes a test dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.5.0</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>
```

Summary

AdminClient is a useful tool to have in your Kafka development kit. It is useful for application developers who want to create topics on the fly and validate that the topics they are using are configured correctly for their application. It is also useful for operators and SREs who want to create tooling and automation around Kafka or need to recover from an incident. AdminClient has so many useful methods that SREs can think of it as a Swiss Army knife for Kafka operations.

In this chapter we covered all the basics of using Kafka's AdminClient: topic management, configuration management, and consumer group management, plus a few other useful methods that are good to have in your back pocket—you never know when you'll need them.

CHAPTER 6

Kafka Internals

It is not strictly necessary to understand Kafka's internals in order to run Kafka in production or write applications that use it. However, knowing how Kafka works does provide context when troubleshooting or trying to understand why Kafka behaves the way it does. Since covering every single implementation detail and design decision is beyond the scope of this book, in this chapter we focus on a few topics that are especially relevant to Kafka practitioners:

- Kafka controller
- How Kafka replication works
- How Kafka handles requests from producers and consumers
- How Kafka handles storage, such as file format and indexes

Understanding these topics in-depth will be especially useful when tuning Kafka—understanding the mechanisms that the tuning knobs control goes a long way toward using them with precise intent rather than fiddling with them randomly.

Cluster Membership

Kafka uses Apache ZooKeeper to maintain the list of brokers that are currently members of a cluster. Every broker has a unique identifier that is either set in the broker configuration file or automatically generated. Every time a broker process starts, it registers itself with its ID in ZooKeeper by creating an *ephemeral node*. Kafka brokers, the controller, and some of the ecosystem tools subscribe to the */brokers/ids* path in ZooKeeper where brokers are registered so that they get notified when brokers are added or removed.

If you try to start another broker with the same ID, you will get an error—the new broker will try to register but fail because we already have a ZooKeeper node for the same broker ID.

When a broker loses connectivity to ZooKeeper (usually as a result of the broker stopping, but this can also happen as a result of network partition or a long garbage-collection pause), the ephemeral node that the broker created when starting will be automatically removed from ZooKeeper. Kafka components that are watching the list of brokers will be notified that the broker is gone.

Even though the node representing the broker is gone when the broker is stopped, the broker ID still exists in other data structures. For example, the list of replicas of each topic (see "Replication" on page 139) contains the broker IDs for the replica. This way, if you completely lose a broker and start a brand-new broker with the ID of the old one, it will immediately join the cluster in place of the missing broker with the same partitions and topics assigned to it.

The Controller

The controller is one of the Kafka brokers that, in addition to the usual broker functionality, is responsible for electing partition leaders. The first broker that starts in the cluster becomes the controller by creating an ephemeral node in ZooKeeper called `/controller`. When other brokers start, they also try to create this node but receive a "node already exists" exception, which causes them to "realize" that the controller node already exists and that the cluster already has a controller. The brokers create a *ZooKeeper watch* on the controller node so they get notified of changes to this node. This way, we guarantee that the cluster will only have one controller at a time.

When the controller broker is stopped or loses connectivity to ZooKeeper, the ephemeral node will disappear. This includes any scenario in which the ZooKeeper client used by the controller stops sending heartbeats to ZooKeeper for longer than `zookeeper.session.timeout.ms`. When the ephemeral node disappears, other brokers in the cluster will be notified through the ZooKeeper watch that the controller is gone and will attempt to create the controller node in ZooKeeper themselves. The first node to create the new controller in ZooKeeper becomes the next controller, while the other nodes will receive a "node already exists" exception and re-create the watch on the new controller node. Each time a controller is elected, it receives a new, higher *controller epoch* number through a ZooKeeper conditional increment operation. The brokers know the current controller epoch, and if they receive a message from a controller with an older number, they know to ignore it. This is important because the controller broker can disconnect from ZooKeeper due to a long garbage collection pause—during this pause a new controller will be elected. When the previous leader resumes operations after the pause, it can continue sending messages to brokers without knowing that there is a new controller—in this case, the old

controller is considered a zombie. The controller epoch in the message, which allows brokers to ignore messages from old controllers, is a form of zombie fencing.

When the controller first comes up, it has to read the latest replica state map from ZooKeeper before it can start managing the cluster metadata and performing leader elections. The loading process uses async APIs, and pipelines the read requests to ZooKeeper to hide latencies. But even so, in clusters with large numbers of partitions, the loading process can take several seconds—several tests and comparisons are described in an Apache Kafka 1.1.0 blog post.

When the controller notices that a broker left the cluster (by watching the relevant ZooKeeper path or because it received a `ControlledShutdownRequest` from the broker), it knows that all the partitions that had a leader on that broker will need a new leader. It goes over all the partitions that need a new leader and determines who the new leader should be (simply the next replica in the replica list of that partition). Then it persists the new state to ZooKeeper (again, using pipelined async requests to reduce latency) and then sends a `LeaderAndISR` request to all the brokers that contain replicas for those partitions. The request contains information on the new leader and followers for the partitions. These requests are batched for efficiency, so each request includes new leadership information for multiple partitions that have a replica on the same broker. Each new leader knows that it needs to start serving producer and consumer requests from clients, while the followers know that they need to start replicating messages from the new leader. Since every broker in the cluster has a `MetadataCache` that includes a map of all brokers and all replicas in the cluster, the controller sends all brokers information about the leadership change in an `UpdateMetadata` request so they can update their caches. A similar process repeats when a broker starts back up—the main difference is that all replicas in the broker start as followers and need to catch up to the leader before they are eligible to be elected as leaders themselves.

To summarize, Kafka uses ZooKeeper's ephemeral node feature to elect a controller and to notify the controller when nodes join and leave the cluster. The controller is responsible for electing leaders among the partitions and replicas whenever it notices nodes join and leave the cluster. The controller uses the epoch number to prevent a "split brain" scenario where two nodes believe each is the current controller.

KRaft: Kafka's New Raft-Based Controller

Starting in 2019, the Apache Kafka community started on an ambitious project: moving away from the ZooKeeper-based controller to a Raft-based controller quorum. The preview version of the new controller, named KRaft, is part of the Apache Kafka 2.8 release. The Apache Kafka 3.0 release, planned for mid 2021, will include the first production version of KRaft, and Kafka clusters will be able to run with either the traditional ZooKeeper-based controller or KRaft.

Why did the Kafka community decide to replace the controller? Kafka's existing controller already underwent several rewrites, but despite improvements to the way it uses ZooKeeper to store the topic, partition, and replica information, it became clear that the existing model will not scale to the number of partitions we want Kafka to support. Several known concerns motivated the change:

- Metadata updates are written to ZooKeeper synchronously but are sent to brokers asynchronously. In addition, receiving updates from ZooKeeper is asynchronous. All this leads to edge cases where metadata is inconsistent between brokers, controller, and ZooKeeper. These cases are challenging to detect.
- Whenever the controller is restarted, it has to read all the metadata for all brokers and partitions from ZooKeeper and then send this metadata to all brokers. Despite years of effort, this remains a major bottleneck—as the number of partitions and brokers increases, restarting the controller becomes slower.
- The internal architecture around metadata ownership is not great—some operations were done via the controller, others via any broker, and others directly on ZooKeeper.
- ZooKeeper is its own distributed system, and, just like Kafka, it requires some expertise to operate. Developers who want to use Kafka therefore need to learn two distributed systems, not just one.

With all these concerns in mind, the Apache Kafka community chose to replace the existing ZooKeeper-based controller.

In the existing architecture, ZooKeeper has two important functions: it is used to elect a controller and to store the cluster metadata—registered brokers, configuration, topics, partitions, and replicas. In addition, the controller itself manages the metadata—it is used to elect leaders, create and delete topics, and reassign replicas. All this functionality will have to be replaced in the new controller.

The core idea behind the new controller design is that Kafka itself has a log-based architecture, where users represent state as a stream of events. The benefits of such representation are well understood in the community—multiple consumers can quickly catch up to the latest state by replaying events. The log establishes a clear ordering between events and ensures that the consumers always move along a single timeline. The new controller architecture brings the same benefits to the management of Kafka's metadata.

In the new architecture, the controller nodes are a Raft quorum that manages the log of metadata events. This log contains information about each change to the cluster metadata. Everything that is currently stored in ZooKeeper, such as topics, partitions, ISRs, configurations, and so on, will be stored in this log.

Using the Raft algorithm, the controller nodes will elect a leader from among themselves, without relying on any external system. The leader of the metadata log is called the *active controller*. The active controller handles all RPCs made from the brokers. The follower controllers replicate the data that is written to the active controller and serve as hot standbys if the active controller should fail. Because the controllers will now all track the latest state, controller failover will not require a lengthy reloading period in which we transfer all the state to the new controller.

Instead of the controller pushing out updates to the other brokers, those brokers will fetch updates from the active controller via a new `MetadataFetch` API. Similar to a fetch request, brokers will track the offset of the latest metadata change they fetched and will only request newer updates from the controller. Brokers will persist the metadata to disk, which will allow them to start up quickly, even with millions of partitions.

Brokers will register with the controller quorum and will remain registered until unregistered by an admin, so once a broker shuts down, it is offline but still registered. Brokers that are online but are not up-to-date with the latest metadata will be fenced and will not be able to serve client requests. The new fenced state will prevent cases where a client produces events to a broker that is no longer a leader but is too out-of-date to be aware that it isn't a leader.

As part of the migration to the controller quorum, all operations that previously involved either clients or brokers communicating directly to ZooKeeper will be routed via the controller. This will allow seamless migration by replacing the controller without having to change anything on any broker.

Overall design of the new architecture is described in KIP-500. Details on how the Raft protocol was adapted for Kafka is described in KIP-595. Detailed design on the new controller quorum, including controller configuration and a new CLI for interacting with cluster metadata, are found in KIP-631.

Replication

Replication is at the heart of Kafka's architecture. Indeed, Kafka is often described as "a distributed, partitioned, replicated commit log service." Replication is critical because it is the way Kafka guarantees availability and durability when individual nodes inevitably fail.

As we've already discussed, data in Kafka is organized by topics. Each topic is partitioned, and each partition can have multiple replicas. Those replicas are stored on brokers, and each broker typically stores hundreds or even thousands of replicas belonging to different topics and partitions.

There are two types of replicas:

Leader replica
: Each partition has a single replica designated as the leader. All produce requests go through the leader to guarantee consistency. Clients can consume from either the lead replica or its followers.

Follower replica
: All replicas for a partition that are not leaders are called followers. Unless configured otherwise, followers don't serve client requests; their main job is to replicate messages from the leader and stay up-to-date with the most recent messages the leader has. If a leader replica for a partition crashes, one of the follower replicas will be promoted to become the new leader for the partition.

Read from Follower

The ability to read from follower replicas was added in KIP-392. The main goal of this feature is to decrease network traffic costs by allowing clients to consume from the nearest in-sync replica rather than from the lead replica. To use this feature, consumer configuration should include `client.rack` identifying the location of the client. Broker configuration should include `replica. selector.class`. This configuration defaults to `LeaderSelector` (always consume from leader) but can be set to `RackAwareReplicaSelector`, which will select a replica that resides on a broker with a `rack.id` configuration that matches `client.rack` on the client. We can also implement our own replica selection logic by implementing the `ReplicaSelector` interface and using our own implementation instead.

The replication protocol was extended to guarantee that only committed messages will be available when consuming from a follower replica. This means that we get the same reliability guarantees we always did, even when fetching from a follower. To provide this guarantee, all replicas need to know which messages were committed by the leader. To achieve this, the leader includes the current high-water mark (latest committed offset) in the data that it sends to the follower. The propagation of the high-water mark introduces a small delay, which means that data is available for consuming from the leader earlier than it is available on the follower. It is important to remember this additional delay, since it is tempting to attempt to decrease consumer latency by consuming from the leader replica.

Another task the leader is responsible for is knowing which of the follower replicas is up-to-date with the leader. Followers attempt to stay up-to-date by replicating all the messages from the leader as the messages arrive, but they can fail to stay in sync for various reasons, such as when network congestion slows down replication or when a broker crashes and all replicas on that broker start falling behind until we start the broker and they can start replicating again.

To stay in sync with the leader, the replicas send the leader `Fetch` requests, the exact same type of requests that consumers send in order to consume messages. In response to those requests, the leader sends the messages to the replicas. Those `Fetch` requests contain the offset of the message that the replica wants to receive next, and will always be in order. This means that the leader can know that a replica got all messages up to the last messages that the replica fetched, and none of the messages that came after. By looking at the last offset requested by each replica, the leader can tell how far behind each replica is. If a replica hasn't requested a message in more than 10 seconds, or if it has requested messages but hasn't caught up to the most recent message in more than 10 seconds, the replica is considered *out of sync*. If a replica fails to keep up with the leader, it can no longer become the new leader in the event of failure—after all, it does not contain all the messages.

The inverse of this, replicas that are consistently asking for the latest messages are called *in-sync replicas*. Only in-sync replicas are eligible to be elected as partition leaders in case the existing leader fails.

The amount of time a follower can be inactive or behind before it is considered out of sync is controlled by the `replica.lag.time.max.ms` configuration parameter. This allowed lag has implications on client behavior and data retention during leader election. We discuss this in depth in Chapter 7 when we discuss reliability guarantees.

In addition to the current leader, each partition has a *preferred leader*—the replica that was the leader when the topic was originally created. It is preferred because when partitions are first created, the leaders are balanced among brokers. As a result, we expect that when the preferred leader is indeed the leader for all partitions in the cluster, load will be evenly balanced between brokers. By default, Kafka is configured with `auto.leader.rebalance.enable=true`, which will check if the preferred leader replica is not the current leader but is in sync, and will trigger leader election to make the preferred leader the current leader.

Finding the Preferred Leaders

The best way to identify the current preferred leader is by looking at the list of replicas for a partition. (You can see details of partitions and replicas in the output of the `kafka-topics.sh` tool. We'll discuss this and other admin tools in Chapter 13.) The first replica in the list is always the preferred leader. This is true no matter who is the current leader and even if the replicas were reassigned to different brokers using the replica reassignment tool. In fact, if you manually reassign replicas, it is important to remember that the replica you specify first will be the preferred replica, so make sure you spread those around different brokers to avoid overloading some brokers with leaders while other brokers are not handling their fair share of the work.

Request Processing

Most of what a Kafka broker does is process requests sent to the partition leaders from clients, partition replicas, and the controller. Kafka has a binary protocol (over TCP) that specifies the format of the requests and how brokers respond to them—both when the request is processed successfully or when the broker encounters errors while processing the request.

The Apache Kafka project includes Java clients that were implemented and maintained by contributors to the Apache Kafka project; there are also clients in other languages, such as C, Python, Go, and many others. You can see the full list on the Apache Kafka website. They all communicate with Kafka brokers using this protocol.

Clients always initiate connections and send requests, and the broker processes the requests and responds to them. All requests sent to the broker from a specific client will be processed in the order in which they were received—this guarantee is what allows Kafka to behave as a message queue and provide ordering guarantees on the messages it stores.

All requests have a standard header that includes:

- Request type (also called *API key*)
- Request version (so the brokers can handle clients of different versions and respond accordingly)
- Correlation ID: a number that uniquely identifies the request and also appears in the response and in the error logs (the ID is used for troubleshooting)
- Client ID: used to identify the application that sent the request

We will not describe the protocol here because it is described in significant detail in the Kafka documentation. However, it is helpful to take a look at how requests are

processed by the broker—later, when we discuss how to monitor Kafka and the various configuration options, you will have context about which queues and threads the metrics and configuration parameters refer to.

For each port the broker listens on, the broker runs an *acceptor* thread that creates a connection and hands it over to a *processor* thread for handling. The number of processor threads (also called *network threads*) is configurable. The network threads are responsible for taking requests from client connections, placing them in a *request queue*, and picking up responses from a *response queue* and sending them back to clients. At times, responses to clients have to be delays—consumers only receive responses when data is available, and admin clients receive a response to a `DeleteTopic` request after topic deletion is underway. The delayed responses are held in a *purgatory* until they can be completed. See Figure 6-1 for a visual of this process.

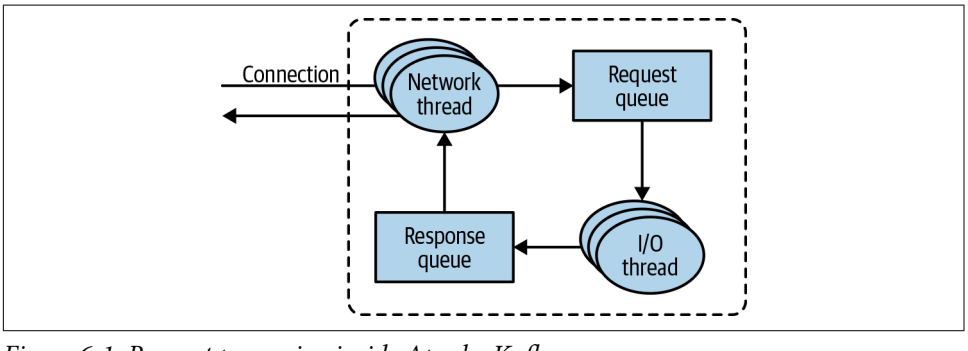

Figure 6-1. Request processing inside Apache Kafka

Once requests are placed on the request queue, *I/O threads* (also called request *handler threads*) are responsible for picking them up and processing them. The most common types of client requests are:

Produce requests
: Sent by producers and contain messages the clients write to Kafka brokers

Fetch requests
: Sent by consumers and follower replicas when they read messages from Kafka brokers

Admin requests
: Sent by admin clients when performing metadata operations such as creating and deleting topics

Both produce requests and fetch requests have to be sent to the leader replica of a partition. If a broker receives a produce request for a specific partition and the leader for this partition is on a different broker, the client that sent the produce request will get an error response of "Not a Leader for Partition." The same error will occur if a

fetch request for a specific partition arrives at a broker that does not have the leader for that partition. Kafka's clients are responsible for sending produce and fetch requests to the broker that contains the leader for the relevant partition for the request.

How do the clients know where to send the requests? Kafka clients use another request type called a *metadata request*, which includes a list of topics the client is interested in. The server response specifies which partitions exist in the topics, the replicas for each partition, and which replica is the leader. Metadata requests can be sent to any broker because all brokers have a metadata cache that contains this information.

Clients typically cache this information and use it to direct produce and fetch requests to the correct broker for each partition. They also need to occasionally refresh this information (refresh intervals are controlled by the `metadata.max.age.ms` configuration parameter) by sending another metadata request so they know if the topic metadata changed—for example, if a new broker was added or some replicas were moved to a new broker (Figure 6-2). In addition, if a client receives the "Not a Leader" error to one of its requests, it will refresh its metadata before trying to send the request again, since the error indicates that the client is using outdated information and is sending requests to the wrong broker.

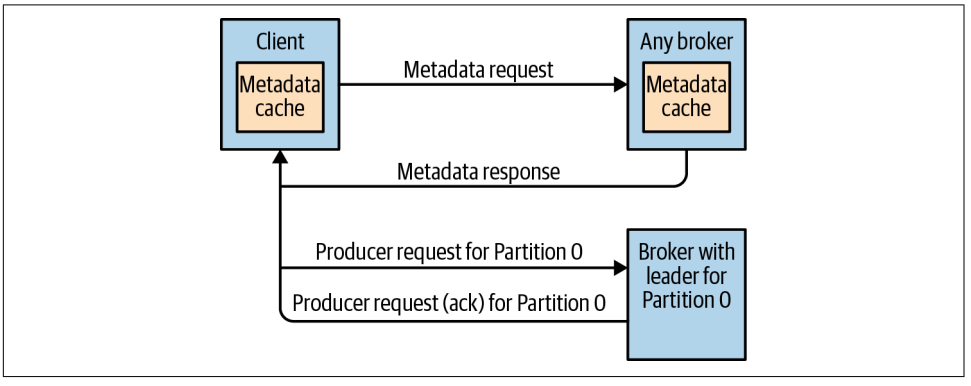

Figure 6-2. Client routing requests

Produce Requests

As we saw in Chapter 3, a configuration parameter called `acks` is the number of brokers that need to acknowledge receiving the message before it is considered a successful write. Producers can be configured to consider messages as "written successfully" when the message was accepted by just the leader (`acks=1`), or by all in-sync replicas (`acks=all`), or the moment the message was sent without waiting for the broker to accept it at all (`acks=0`).

When the broker that contains the lead replica for a partition receives a produce request for this partition, it will start by running a few validations:

- Does the user sending the data have write privileges on the topic?
- Is the number of `acks` specified in the request valid (only 0, 1, and "all" are allowed)?
- If `acks` is set to `all`, are there enough in-sync replicas for safely writing the message? (Brokers can be configured to refuse new messages if the number of in-sync replicas falls below a configurable number; we will discuss this in more detail in Chapter 7, when we discuss Kafka's durability and reliability guarantees.)

Then the broker will write the new messages to local disk. On Linux, the messages are written to the filesystem cache, and there is no guarantee about when they will be written to disk. Kafka does not wait for the data to get persisted to disk—it relies on replication for message durability.

Once the message is written to the leader of the partition, the broker examines the `acks` configuration: if `acks` is set to 0 or 1, the broker will respond immediately; if `acks` is set to `all`, the request will be stored in a buffer called *purgatory* until the leader observes that the follower replicas replicated the message, at which point a response is sent to the client.

Fetch Requests

Brokers process fetch requests in a way that is very similar to how produce requests are handled. The client sends a request, asking the broker to send messages from a list of topics, partitions, and offsets—something like "Please send me messages starting at offset 53 in partition 0 of topic Test and messages starting at offset 64 in partition 3 of topic Test." Clients also specify a limit to how much data the broker can return for each partition. The limit is important because clients need to allocate memory that will hold the response sent back from the broker. Without this limit, brokers could send back replies large enough to cause clients to run out of memory.

As we've discussed earlier, the request has to arrive to the leaders of the partitions specified in the request, and the client will make the necessary metadata requests to make sure it is routing the fetch requests correctly. When the leader receives the request, it first checks if the request is valid—does this offset even exist for this particular partition? If the client is asking for a message that is so old it got deleted from the partition or an offset that does not exist yet, the broker will respond with an error.

If the offset exists, the broker will read messages from the partition, up to the limit set by the client in the request, and send the messages to the client. Kafka famously uses a `zero-copy` method to send the messages to the clients—this means that Kafka sends messages from the file (or more likely, the Linux filesystem cache) directly to the

network channel without any intermediate buffers. This is different than most databases where data is stored in a local cache before being sent to clients. This technique removes the overhead of copying bytes and managing buffers in memory, and results in much improved performance.

In addition to setting an upper boundary on the amount of data the broker can return, clients can also set a lower boundary on the amount of data returned. Setting the lower boundary to 10K, for example, is the client's way of telling the broker, "Only return results once you have at least 10K bytes to send me." This is a great way to reduce CPU and network utilization when clients are reading from topics that are not seeing much traffic. Instead of the clients sending requests to the brokers every few milliseconds asking for data and getting very few or no messages in return, the clients send a request, the broker waits until there is a decent amount of data, and returns the data, and only then will the client ask for more (Figure 6-3). The same amount of data is read overall but with much less back-and-forth and therefore less overhead.

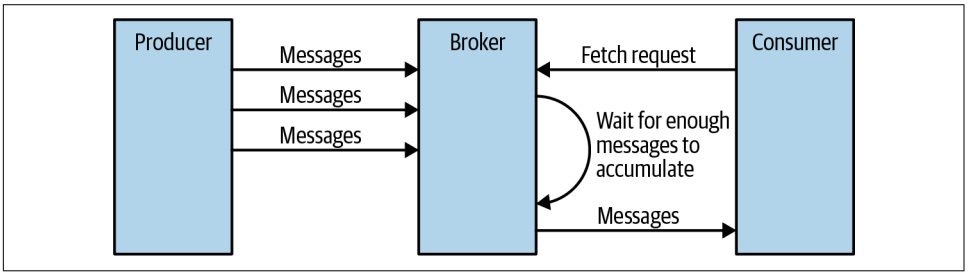

Figure 6-3. Broker delaying response until enough data accumulated

Of course, we wouldn't want clients to wait forever for the broker to have enough data. After a while, it makes sense to just take the data that exists and process that instead of waiting for more. Therefore, clients can also define a timeout to tell the broker, "If you didn't satisfy the minimum amount of data to send within *x* milliseconds, just send what you got."

It is interesting to note that not all the data that exists on the leader of the partition is available for clients to read. Most clients can only read messages that were written to all in-sync replicas (follower replicas, even though they are consumers, are exempt from this—otherwise replication would not work). We already discussed that the leader of the partition knows which messages were replicated to which replica, and until a message was written to all in-sync replicas, it will not be sent to consumers—attempts to fetch those messages will result in an empty response rather than an error.

The reason for this behavior is that messages not replicated to enough replicas yet are considered "unsafe"—if the leader crashes and another replica takes its place, these messages will no longer exist in Kafka. If we allowed clients to read messages that only exist on the leader, we could see inconsistent behavior. For example, if a

consumer reads a message and the leader crashed and no other broker contained this message, the message is gone. No other consumer will be able to read this message, which can cause inconsistency with the consumer who did read it. Instead, we wait until all the in-sync replicas get the message and only then allow consumers to read it (Figure 6-4). This behavior also means that if replication between brokers is slow for some reason, it will take longer for new messages to arrive to consumers (since we wait for the messages to replicate first). This delay is limited to `replica.lag.time.max.ms`—the amount of time a replica can be delayed in replicating new messages while still being considered in sync.

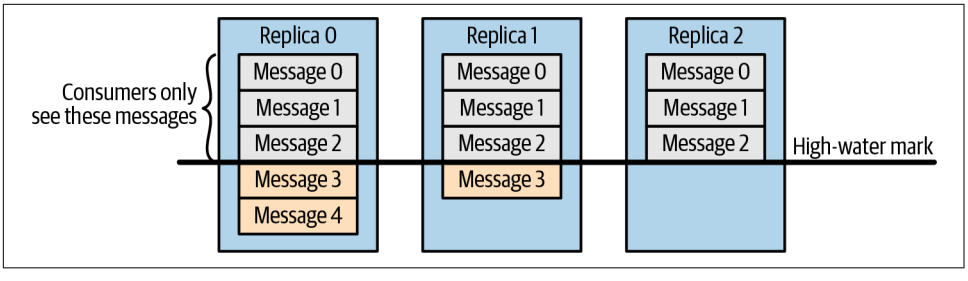

Figure 6-4. Consumers only see messages that were replicated to in-sync replicas

In some cases, a consumer consumes events from a large number of partitions. Sending the list of all the partitions it is interested in to the broker with every request and having the broker send all its metadata back can be very inefficient—the set of partitions rarely changes, their metadata rarely changes, and in many cases there isn't that much data to return. To minimize this overhead, Kafka has `fetch session cache`. Consumers can attempt to create a cached session that stores the list of partitions they are consuming from and its metadata. Once a session is created, consumers no longer need to specify all the partitions in each request and can use incremental fetch requests instead. Brokers will only include metadata in the response if there were any changes. The session cache has limited space, and Kafka prioritizes follower replicas and consumers with a large set of partitions, so in some cases a session will not be created or will be evicted. In both these cases the broker will return an appropriate error to the client, and the consumer will transparently resort to full fetch requests that include all the partition metadata.

Other Requests

We just discussed the most common types of requests used by Kafka clients: `Metadata`, `Produce`, and `Fetch`. The Kafka protocol currently handles 61 different request types, and more will be added. Consumers alone use 15 request types to form groups, coordinate consumption, and allow developers to manage the consumer groups. There are also large numbers of requests that are related to metadata management and security.

In addition, the same protocol is used to communicate between the Kafka brokers themselves. Those requests are internal and should not be used by clients. For example, when the controller announces that a partition has a new leader, it sends a `LeaderAndIsr` request to the new leader (so it will know to start accepting client requests) and to the followers (so they will know to follow the new leader).

The protocol is ever evolving—as the Kafka community adds more client capabilities, the protocol evolves to match. For example, in the past, Kafka consumers used Apache ZooKeeper to keep track of the offsets they receive from Kafka. So when a consumer is started, it can check ZooKeeper for the last offset that was read from its partitions and know where to start processing. For various reasons, the community decided to stop using ZooKeeper for this and instead stored those offsets in a special Kafka topic. To do this, the contributors had to add several requests to the protocol: `OffsetCommitRequest`, `OffsetFetchRequest`, and `ListOffsetsRequest`. Now when an application calls the client API to commit consumer offsets, the client no longer writes to ZooKeeper; instead, it sends `OffsetCommitRequest` to Kafka.

Topic creation used to be handled by command-line tools that directly update the list of topics in ZooKeeper. The Kafka community since added a `CreateTopicRequest`, and similar requests for managing Kafka's metadata. Java applications perform these metadata operations through Kafka's `AdminClient`, documented in depth in Chapter 5. Since these operations are now part of the Kafka protocol, it allows clients in languages that don't have a ZooKeeper library to create topics by asking Kafka brokers directly.

In addition to evolving the protocol by adding new request types, Kafka developers sometimes choose to modify existing requests to add some capabilities. For example, between Kafka 0.9.0 and Kafka 0.10.0, they've decided to let clients know who the current controller is by adding the information to the `Metadata` response. As a result, a new version was added to the `Metadata` request and response. Now, 0.9.0 clients send `Metadata` requests of version 0 (because version 1 did not exist in 0.9.0 clients), and the brokers, whether they are 0.9.0 or 0.10.0, know to respond with a version 0 response, which does not have the controller information. This is fine, because 0.9.0 clients don't expect the controller information and wouldn't know how to parse it anyway. If you have the 0.10.0 client, it will send a version 1 `Metadata` request, and 0.10.0 brokers will respond with a version 1 response that contains the controller information, which the 0.10.0 clients can use. If a 0.10.0 client sends a version 1 `Metadata` request to a 0.9.0 broker, the broker will not know how to handle the newer version of the request and will respond with an error. This is the reason we recommend upgrading the brokers before upgrading any of the clients—new brokers know how to handle old requests, but not vice versa.

In release 0.10.0, the Kafka community added `ApiVersionRequest`, which allows clients to ask the broker which versions of each request are supported and to use the

correct version accordingly. Clients that use this new capability correctly will be able to talk to older brokers by using a version of the protocol that is supported by the broker they are connecting to. There is currently ongoing work to add APIs that will allow clients to discover which features are supported by brokers and to allow brokers to gate features that exist in a specific version. This improvement was proposed in KIP-584, and at this time it seems likely to be part of version 3.0.0.

Physical Storage

The basic storage unit of Kafka is a partition replica. Partitions cannot be split between multiple brokers, and not even between multiple disks on the same broker. So the size of a partition is limited by the space available on a single mount point. (A mount point can be a single disk, if JBOD configuration is used, or multiple disks, if RAID is configured. See Chapter 2.)

When configuring Kafka, the administrator defines a list of directories in which the partitions will be stored—this is the `log.dirs` parameter (not to be confused with the location in which Kafka stores its error log, which is configured in the *log4j.properties* file). The usual configuration includes a directory for each mount point that Kafka will use.

Let's look at how Kafka uses the available directories to store data. First, we want to look at how data is allocated to the brokers in the cluster and the directories in the broker. Then we will look at how the broker manages the files—especially how the retention guarantees are handled. We will then dive inside the files and look at the file and index formats. Finally, we will look at log compaction, an advanced feature that allows you to turn Kafka into a long-term data store, and describe how it works.

Tiered Storage

Starting in late 2018, the Apache Kafka community began collaborating on an ambitious project to add tiered storage capabilities to Kafka. Work on the project is ongoing, and it is planned for the 3.0 release.

The motivation is fairly straightforward: Kafka is currently used to store large amounts of data, either due to high throughput or long retention periods. This introduces the following concerns:

- You are limited in how much data you can store in a partition. As a result, maximum retention and partition counts aren't simply driven by product requirements but also by the limits on physical disk sizes.
- Your choice of disk and cluster size is driven by storage requirements. Clusters often end up larger than they would if latency and throughput were the main considerations, which drives up costs.

- The time it takes to move partitions from one broker to another, for example, when expanding or shrinking the cluster, is driven by the size of the partitions. Large partitions make the cluster less elastic. These days, architectures are designed toward maximum elasticity, taking advantage of flexible cloud deployment options.

In the tiered storage approach, the Kafka cluster is configured with two tiers of storage: local and remote. The local tier is the same as the current Kafka storage tier—it uses the local disks on the Kafka brokers to store the log segments. The new remote tier uses dedicated storage systems, such as HDFS or S3, to store the completed log segments.

Kafka users can choose to set a separate storage retention policy for each tier. Since local storage is typically far more expensive than the remote tier, the retention period for the local tier is usually just a few hours or even shorter, and the retention period for the remote tier can be much longer—days, or even months.

Local storage is significantly lower latency than the remote storage. This works well because latency-sensitive applications perform tail reads and are served from the local tier, so they benefit from the existing Kafka mechanism of efficiently using the page cache to serve the data. Backfill and other applications recovering from a failure that needs data older than what is in the local tier are served from the remote tier.

The dual-tier architecture used in tiered storage allows scaling storage independent of memory and CPUs in a Kafka cluster. This enables Kafka to be a long-term storage solution. This also reduces the amount of data stored locally on Kafka brokers, and hence the amount of data that needs to be copied during recovery and rebalancing. Log segments that are available in the remote tier need not be restored on the broker or restored lazily and are served from the remote tier. Since not all data is stored on the brokers, increasing the retention period no longer requires scaling the Kafka cluster storage and adding new nodes. At the same time, the overall data retention can still be much longer, eliminating the need for separate data pipelines to copy the data from Kafka to external stores, as done currently in many deployments.

The design of tiered storage is documented in detail in KIP-405, including a new component—the `RemoteLogManager` and the interactions with existing functionality, such as replicas catching up to the leader and leader elections.

One interesting result that is documented in KIP-405 is the performance implications of tiered storage. The team implementing tiered storage measured performance in several use cases. The first was using Kafka's usual high-throughput workload. In that case, latency increased a bit (from 21 ms in p99 to 25 ms), since brokers also have to ship segments to remote storage. The second use case was when some consumers are reading old data. Without tiered storage, consumers reading old data have a large impact on latency (21 ms versus 60 ms p99), but with tiered storage enabled, the

impact is significantly lower (25 ms versus 42 ms p99); this is because tiered storage reads are read from HDFS or S3 via a network path. Network reads do not compete with local reads on disk I/O or page cache, and leave the page cache intact with fresh data.

This means that in addition to infinite storage, lower costs, and elasticity, tiered storage also delivers isolation between historical reads and real-time reads.

Partition Allocation

When you create a topic, Kafka first decides how to allocate the partitions between brokers. Suppose you have 6 brokers and you decide to create a topic with 10 partitions and a replication factor of 3. Kafka now has 30 partition replicas to allocate to 6 brokers. When doing the allocations, the goals are:

- To spread replicas evenly among brokers—in our example, to make sure we allocate five replicas per broker.
- To make sure that for each partition, each replica is on a different broker. If partition 0 has the leader on broker 2, we can place the followers on brokers 3 and 4, but not on 2 and not both on 3.
- If the brokers have rack information (available in Kafka release 0.10.0 and higher), then assign the replicas for each partition to different racks if possible. This ensures that an event that causes downtime for an entire rack does not cause complete unavailability for partitions.

To do this, we start with a random broker (let's say 4) and start assigning partitions to each broker in a round-robin manner to determine the location for the leaders. So partition 0 leader will be on broker 4, partition 1 leader will be on broker 5, partition 2 will be on broker 0 (because we only have 6 brokers), and so on. Then, for each partition, we place the replicas at increasing offsets from the leader. If the leader for partition 0 is on broker 4, the first follower will be on broker 5 and the second on broker 0. The leader for partition 1 is on broker 5, so the first replica is on broker 0 and the second on broker 1.

When rack awareness is taken into account, instead of picking brokers in numerical order, we prepare a rack-alternating broker list. Suppose that we know that brokers 0 and 1 are on the same rack, and brokers 2 and 3 are on a separate rack. Instead of picking brokers in the order of 0 to 3, we order them as 0, 2, 1, 3—each broker is followed by a broker from a different rack (Figure 6-5). In this case, if the leader for partition 0 is on broker 2, the first replica will be on broker 1, which is on a completely different rack. This is great, because if the first rack goes offline, we know that we still have a surviving replica, and therefore the partition is still available. This will

be true for all our replicas, so we have guaranteed availability in the case of rack failure.

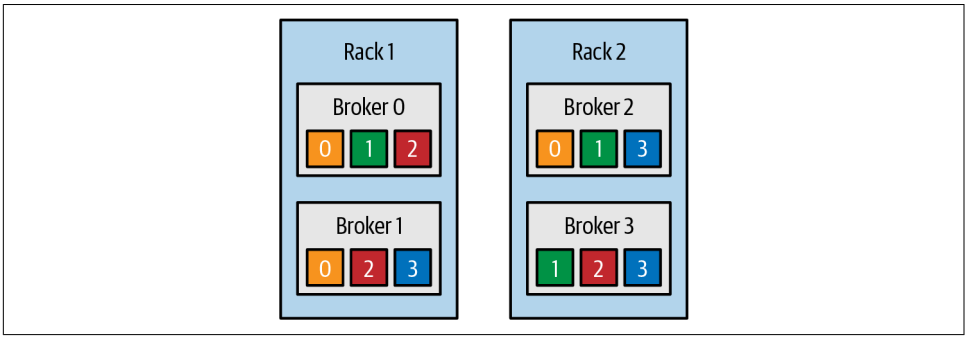

Figure 6-5. Partitions and replicas assigned to brokers on different racks

Once we choose the correct brokers for each partition and replica, it is time to decide which directory to use for the new partitions. We do this independently for each partition, and the rule is very simple: we count the number of partitions on each directory and add the new partition to the directory with the fewest partitions. This means that if you add a new disk, all the new partitions will be created on that disk. This is because, until things balance out, the new disk will always have the fewest partitions.

Mind the Disk Space

Note that the allocation of partitions to brokers does not take available space or existing load into account, and that allocation of partitions to disks takes the number of partitions into account but not the size of the partitions. This means that if some brokers have more disk space than others (perhaps because the cluster is a mix of older and newer servers), some partitions are abnormally large, or you have disks of different sizes on the same broker, you need to be careful with the partition allocation.

File Management

Retention is an important concept in Kafka—Kafka does not keep data forever, nor does it wait for all consumers to read a message before deleting it. Instead, the Kafka administrator configures a retention period for each topic—either the amount of time to store messages before deleting them or how much data to store before older messages are purged.

Because finding the messages that need purging in a large file and then deleting a portion of the file is both time-consuming and error prone, we instead split each partition into *segments*. By default, each segment contains either 1 GB of data or a week

of data, whichever is smaller. As a Kafka broker is writing to a partition, if the segment limit is reached, it closes the file and starts a new one.

The segment we are currently writing to is called an *active segment*. The active segment is never deleted, so if you set log retention to only store a day of data, but each segment contains five days of data, you will really keep data for five days because we can't delete the data before the segment is closed. If you choose to store data for a week and roll a new segment every day, you will see that every day we will roll a new segment while deleting the oldest segment—so most of the time the partition will have seven segments.

As you learned in Chapter 2, a Kafka broker will keep an open file handle to every segment in every partition—even inactive segments. This leads to an usually high number of open file handles, and the OS must be tuned accordingly.

File Format

Each segment is stored in a single data file. Inside the file, we store Kafka messages and their offsets. The format of the data on the disk is identical to the format of the messages that we send from the producer to the broker and later from the broker to the consumers. Using the same message format on disk and over the wire is what allows Kafka to use zero-copy optimization when sending messages to consumers, and also avoid decompressing and recompressing messages that the producer already compressed. As a result, if we decide to change the message format, both the wire protocol and the on-disk format need to change, and Kafka brokers need to know how to handle cases in which files contain messages of two formats due to upgrades.

Kafka messages consist of user payload and system headers. User payload includes an optional key, a value, and an optional collection of headers, where each header is its own key/value pair.

Starting with version 0.11 (and the v2 message format), Kafka producers always send messages in batches. If you send a single message, the batching adds a bit of overhead. But with two messages or more per batch, the batching saves space, which reduces network and disk usage. This is one of the reasons why Kafka performs better with `linger.ms=10`—the small delay increases the chance that more messages will be sent together. Since Kafka creates a separate batch per partition, producers that write to fewer partitions will be more efficient as well. Note that Kafka producers can include multiple batches in the same produce request. This means that if you are using compression on the producer (recommended!), sending larger batches means better compression both over the network and on the broker disks.

Message batch headers include:

- A magic number indicating the current version of the message format (here we're documenting v2).
- The offset of the first message in the batch and the difference from the offset of the last message—those are preserved even if the batch is later compacted and some messages are removed. The offset of the first message is set to 0 when the producer creates and sends the batch. The broker that first persists this batch (the partition leader) replaces this with the real offset.
- The timestamps of the first message and the highest timestamp in the batch. The timestamps can be set by the broker if the timestamp type is set to append time rather than create time.
- Size of the batch, in bytes.
- The epoch of the leader that received the batch (this is used when truncating messages after leader election; KIP-101 and KIP-279 explain the usage in detail).
- Checksum for validating that the batch is not corrupted.
- Sixteen bits indicating different attributes: compression type, timestamp type (timestamp can be set at the client or at the broker), and whether the batch is part of a transaction or is a control batch.
- Producer ID, producer epoch, and the first sequence in the batch—these are all used for exactly-once guarantees.
- And, of course, the set of messages that are part of the batch.

As you can see, the batch header includes a lot of information. The records themselves also have system headers (not to be confused with headers that can be set by users). Each record includes:

- Size of the record, in bytes
- Attributes—currently there are no record-level attributes, so this isn't used
- The difference between the offset of the current record and the first offset in the batch
- The difference, in milliseconds, between the timestamp of this record and the first timestamp in the batch
- The user payload: key, value, and headers

Note that there is very little overhead to each record, and most of the system information is at the batch level. Storing the first offset and timestamp of the batch in the header and only storing the difference in each record dramatically reduces the overhead of each record, making larger batches more efficient.

In addition to message batches that contain user data, Kafka also has control batches—indicating transactional commits, for instance. Those are handled by the consumer and not passed to the user application, and currently they include a version and a type indicator: 0 for an aborted transaction, 1 for a commit.

If you wish to see all this for yourself, Kafka brokers ship with the `DumpLogSegment` tool, which allows you to look at a partition segment in the filesystem and examine its contents. You can run the tool using:

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

If you choose the `--deep-iteration` parameter, it will show you information about messages compressed inside the wrapper messages.

Message Format Down Conversion

The message format documented earlier was introduced in version 0.11. Since Kafka supports upgrading brokers before all the clients are upgraded, it had to support any combination of versions between the broker, producer, and consumer. Most combinations work with no issues—new brokers will understand the old message format from producers, and new producers will know to send old format messages to old brokers. But there is a challenging situation when a new producer sends v2 messages to new brokers: the message is stored in v2 format, but an old consumer that doesn't support v2 format tries to read it. In this scenario, the broker will need to convert the message from v2 format to v1, so the consumer will be able to parse it. This conversion uses far more CPU and memory than normal consumption, so it is best avoided. KIP-188 introduced several important health metrics, among them `FetchMessageConversionsPerSec` and `MessageConversionsTimeMs`. If your organization is still using old clients, we recommend checking the metrics and upgrading the clients as soon as possible.

Indexes

Kafka allows consumers to start fetching messages from any available offset. This means that if a consumer asks for 1 MB messages starting at offset 100, the broker must be able to quickly locate the message for offset 100 (which can be in any of the segments for the partition) and start reading the messages from that offset on. In order to help brokers quickly locate the message for a given offset, Kafka maintains an index for each partition. The index maps offsets to segment files and positions within the file.

Similarly, Kafka has a second index that maps timestamps to message offsets. This index is used when searching for messages by timestamp. Kafka Streams uses this lookup extensively, and it is also useful in some failover scenarios.

Indexes are also broken into segments, so we can delete old index entries when the messages are purged. Kafka does not attempt to maintain checksums of the index. If the index becomes corrupted, it will get regenerated from the matching log segment simply by rereading the messages and recording the offsets and locations. It is also completely safe (albeit, it can cause a lengthy recovery) for an administrator to delete index segments if needed—they will be regenerated automatically.

Compaction

Normally, Kafka will store messages for a set amount of time and purge messages older than the retention period. However, imagine a case where you use Kafka to store shipping addresses for your customers. In that case, it makes more sense to store the last address for each customer rather than data for just the last week or year. This way, you don't have to worry about old addresses, and you still retain the address for customers who haven't moved in a while. Another use case can be an application that uses Kafka to store its current state. Every time the state changes, the application writes the new state into Kafka. When recovering from a crash, the application reads those messages from Kafka to recover its latest state. In this case, it only cares about the latest state before the crash, not all the changes that occurred while it was running.

Kafka supports such use cases by allowing the retention policy on a topic to be *delete*, which deletes events older than retention time, or to be *compact*, which only stores the most recent value for each key in the topic. Obviously, setting the policy to compact only makes sense on topics for which applications produce events that contain both a key and a value. If the topic contains *null* keys, compaction will fail.

Topics can also have a `delete.and.compact` policy that combines compaction with a retention period. Messages older than the retention period will be removed even if they are the most recent value for a key. This policy prevents compacted topics from growing overly large and is also used when the business requires removing records after a certain time period.

How Compaction Works

Each log is viewed as split into two portions (see Figure 6-6):

Clean
: Messages that have been compacted before. This section contains only one value for each key, which is the latest value at the time of the previous compaction.

Dirty
: Messages that were written after the last compaction.

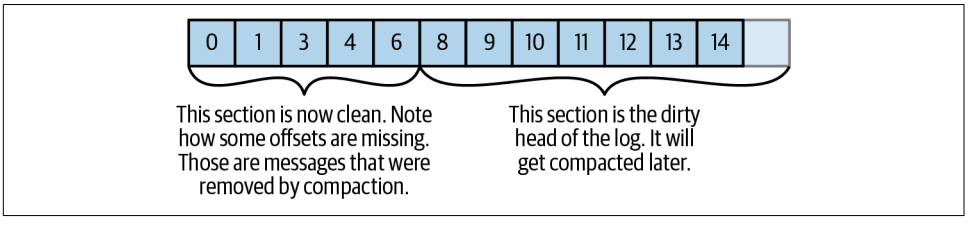

Figure 6-6. Partition with clean and dirty portions

If compaction is enabled when Kafka starts (using the awkwardly named `log.cleaner.enabled` configuration), each broker will start a compaction manager thread and a number of compaction threads. These are responsible for performing the compaction tasks. Each thread chooses the partition with the highest ratio of dirty messages to total partition size and cleans this partition.

To compact a partition, the cleaner thread reads the dirty section of the partition and creates an in-memory map. Each map entry is comprised of a 16-byte hash of a message key and the 8-byte offset of the previous message that had this same key. This means each map entry only uses 24 bytes. If we look at a 1 GB segment and assume that each message in the segment takes up 1 KB, the segment will contain 1 million such messages, and we will only need a 24 MB map to compact the segment (we may need a lot less—if the keys repeat themselves, we will reuse the same hash entries often and use less memory). This is quite efficient!

When configuring Kafka, the administrator configures how much memory compaction threads can use for this offset map. Even though each thread has its own map, the configuration is for total memory across all threads. If you configured 1 GB for the compaction offset map and you have 5 cleaner threads, each thread will get 200 MB for its own offset map. Kafka doesn't require the entire dirty section of the partition to fit into the size allocated for this map, but at least one full segment has to fit. If it doesn't, Kafka will log an error, and the administrator will need to either allocate more memory for the offset maps or use fewer cleaner threads. If only a few segments fit, Kafka will start by compacting the oldest segments that fit into the map. The rest will remain dirty and wait for the next compaction.

Once the cleaner thread builds the offset map, it will start reading off the clean segments, starting with the oldest, and check their contents against the offset map. For each message, it checks if the key of the message exists in the offset map. If the key does not exist in the map, the value of the message just read is still the latest, and the message is copied over to a replacement segment. If the key does exist in the map, the message is omitted because there is a message with an identical key but newer value later in the partition. Once all the messages that still contain the latest value for their

key are copied over, the replacement segment is swapped for the original and the thread on to the next segment. At the end of the process, we are left with one message per key—the one with the latest value. See Figure 6-7.

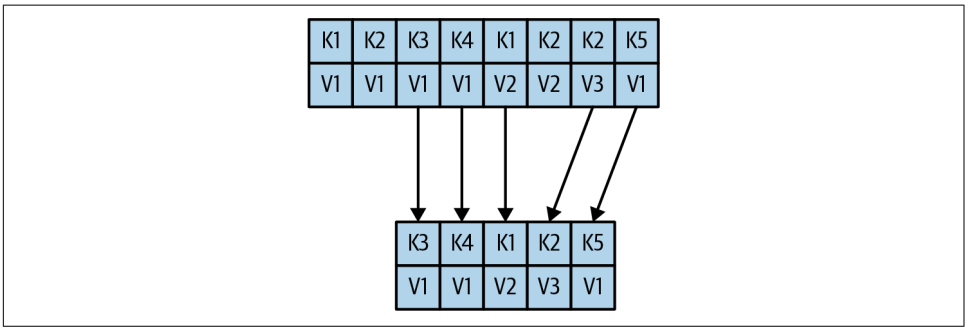

Figure 6-7. Partition segment before and after compaction

Deleted Events

If we always keep the latest message for each key, what do we do when we really want to delete all messages for a specific key, such as if a user left our service and we are legally obligated to remove all traces of that user from our system?

To delete a key from the system completely, not even saving the last message, the application must produce a message that contains that key and a null value. When the cleaner thread finds such a message, it will first do a normal compaction and retain only the message with the null value. It will keep this special message (known as a *tombstone*) around for a configurable amount of time. During this time, consumers will be able to see this message and know that the value is deleted. So if a consumer copies data from Kafka to a relational database, it will see the tombstone message and know to delete the user from the database. After this set amount of time, the cleaner thread will remove the tombstone message, and the key will be gone from the partition in Kafka. It is important to give consumers enough time to see the tombstone message, because if our consumer was down for a few hours and missed the tombstone message, it will simply not see the key when consuming and therefore not know that it was deleted from Kafka or that it needs to be deleted from the database.

It's worth remembering that Kafka's admin client also includes a `deleteRecords` method. This method deletes all records before a specified offset, and it uses a completely different mechanism. When this method is called, Kafka will move the low-water mark, its record of the first offset of a partition, to the specified offset. This will prevent consumers from consuming the records below the new low-water mark and effectively makes these records inaccessible until they get deleted by a cleaner thread. This method can be used on topics with a retention policy and on compacted topics.

When Are Topics Compacted?

In the same way that the `delete` policy never deletes the current active segments, the `compact` policy never compacts the current segment. Messages are eligible for compaction only on inactive segments.

By default, Kafka will start compacting when 50% of the topic contains dirty records. The goal is not to compact too often (since compaction can impact the read/write performance on a topic) but also not to leave too many dirty records around (since they consume disk space). Wasting 50% of the disk space used by a topic on dirty records and then compacting them in one go seems like a reasonable trade-off, and it can be tuned by the administrator.

In addition, administrators can control the timing of compaction with two configuration parameters:

- `min.compaction.lag.ms` can be used to guarantee the minimum length of time that must pass after a message is written before it could be compacted.
- `max.compaction.lag.ms` can be used to guarantee the maximum delay between the time a message is written and the time the message becomes eligible for compaction. This configuration is often used in situations where there is a business reason to guarantee compaction within a certain period; for example, GDPR requires that certain information will be deleted within 30 days after a request to delete has been made.

Summary

There is obviously more to Kafka than we could cover in this chapter, but we hope this gave you a taste of the kind of design decisions and optimizations the Kafka community made when working on the project and perhaps explained some of the more obscure behaviors and configurations you've run into while using Kafka.

If you are really interested in Kafka internals, there is no substitute for reading the code. The Kafka developer mailing list (*dev@kafka.apache.org*) is a very friendly community, and there is always someone willing to answer questions regarding how Kafka really works. And while you are reading the code, perhaps you can fix a bug or two—open source projects always welcome contributions.

CHAPTER 7
Reliable Data Delivery

Reliability is a property of a system—not of a single component—so when we are talking about the reliability guarantees of Apache Kafka, we will need to keep the entire system and its use cases in mind. When it comes to reliability, the systems that integrate with Kafka are as important as Kafka itself. And because reliability is a system concern, it cannot be the responsibility of just one person. Everyone—Kafka administrators, Linux administrators, network and storage administrators, and the application developers—must work together to build a reliable system.

Apache Kafka is very flexible about reliable data delivery. We understand that Kafka has many use cases, from tracking clicks in a website to credit card payments. Some of the use cases require utmost reliability, while others prioritize speed and simplicity over reliability. Kafka was written to be configurable enough, and its client API flexible enough, to allow all kinds of reliability trade-offs.

Because of its flexibility, it is also easy to accidentally shoot ourselves in the foot when using Kafka—believing that our system is reliable when in fact it is not. In this chapter, we will start by talking about different kinds of reliability and what they mean in the context of Apache Kafka. Then we will talk about Kafka's replication mechanism and how it contributes to the reliability of the system. We will then discuss Kafka's brokers and topics and how they should be configured for different use cases. Then we will discuss the clients, producer, and consumer, and how they should be used in different reliability scenarios. Last, we will discuss the topic of validating the system reliability, because it is not enough to believe a system is reliable—the assumption must be thoroughly tested.

Reliability Guarantees

When we talk about reliability, we usually talk in terms of *guarantees*, which are the behaviors a system is guaranteed to preserve under different circumstances.

Probably the best-known reliability guarantee is ACID, which is the standard reliability guarantee that relational databases universally support. ACID stands for *atomicity*, *consistency*, *isolation*, and *durability*. When a vendor explains that their database is ACID compliant, it means the database guarantees certain behaviors regarding transaction behavior.

Those guarantees are the reason people trust relational databases with their most critical applications—they know exactly what the system promises and how it will behave in different conditions. They understand the guarantees and can write safe applications by relying on those guarantees.

Understanding the guarantees Kafka provides is critical for those seeking to build reliable applications. This understanding allows the developers of the system to figure out how it will behave under different failure conditions. So, what does Apache Kafka guarantee?

- Kafka provides order guarantee of messages in a partition. If message B was written after message A, using the same producer in the same partition, then Kafka guarantees that the offset of message B will be higher than message A, and that consumers will read message B after message A.
- Produced messages are considered "committed" when they were written to the partition on all its in-sync replicas (but not necessarily flushed to disk). Producers can choose to receive acknowledgments of sent messages when the message was fully committed, when it was written to the leader, or when it was sent over the network.
- Messages that are committed will not be lost as long as at least one replica remains alive.
- Consumers can only read messages that are committed.

These basic guarantees can be used while building a reliable system, but in themselves, they don't make the system fully reliable. There are trade-offs involved in building a reliable system, and Kafka was built to allow administrators and developers to decide how much reliability they need by providing configuration parameters that allow controlling these trade-offs. The trade-offs usually involve how important it is to reliably and consistently store messages versus other important considerations, such as availability, high throughput, low latency, and hardware costs.

We next review Kafka's replication mechanism, introduce terminology, and discuss how reliability is built into Kafka. After that, we go over the configuration parameters we just mentioned.

Replication

Kafka's replication mechanism, with its multiple replicas per partition, is at the core of all of Kafka's reliability guarantees. Having a message written in multiple replicas is how Kafka provides durability of messages in the event of a crash.

We explained Kafka's replication mechanism in depth in Chapter 6, but let's recap the highlights here.

Each Kafka topic is broken down into *partitions*, which are the basic data building blocks. A partition is stored on a single disk. Kafka guarantees the order of events within a partition, and a partition can be either online (available) or offline (unavailable). Each partition can have multiple replicas, one of which is a designated leader. All events are produced to the leader replica and are usually consumed from the leader replica as well. Other replicas just need to stay in sync with the leader and replicate all the recent events on time. If the leader becomes unavailable, one of the in-sync replicas becomes the new leader (there is an exception to this rule, which we discussed in Chapter 6).

A replica is considered in sync if it is the leader for a partition, or if it is a follower that:

- Has an active session with ZooKeeper—meaning that it sent a heartbeat to ZooKeeper in the last 6 seconds (configurable).
- Fetched messages from the leader in the last 10 seconds (configurable).
- Fetched the most recent messages from the leader in the last 10 seconds. That is, it isn't enough that the follower is still getting messages from the leader; it must have had no lag at least once in the last 10 seconds (configurable).

If a replica loses connection to ZooKeeper, stops fetching new messages, or falls behind and can't catch up within 10 seconds, the replica is considered out of sync. An out-of-sync replica gets back into sync when it connects to ZooKeeper again and catches up to the most recent message written to the leader. This usually happens quickly after a temporary network glitch is healed but can take a while if the broker the replica is stored on was down for a longer period of time.

Out-of-Sync Replicas

In older versions of Kafka, it was not uncommon to see one or more replicas rapidly flip between in-sync and out-of-sync status. This was a sure sign that something was wrong with the cluster. A relatively common cause was a large maximum request size and large JVM heap that required tuning to prevent long garbage collection pauses that would cause the broker to temporarily disconnect from ZooKeeper. These days the problem is very rare, especially when using Apache Kafka release 2.5.0 and higher with its default configurations for ZooKeeper connection timeout and maximum replica lag. The use of JVM version 8 and above (now the minimum version supported by Kafka) with G1 garbage collector helped curb this problem, although tuning may still be required for large messages. Generally speaking, Kafka's replication protocol became significantly more reliable in the years since the first edition of the book was published. For details on the evolution of Kafka's replication protocol, refer to Jason Gustafson's excellent talk, "Hardening Apache Kafka Replication", and Gwen Shapira's overview of Kafka improvements, "Please Upgrade Apache Kafka Now".

An in-sync replica that is slightly behind can slow down producers and consumers—since they wait for all the in-sync replicas to get the message before it is *committed*. Once a replica falls out of sync, we no longer wait for it to get messages. It is still behind, but now there is no performance impact. The catch is that with fewer in-sync replicas, the effective replication factor of the partition is lower, and therefore there is a higher risk for downtime or data loss.

In the next section, we will look at what this means in practice.

Broker Configuration

There are three configuration parameters in the broker that change Kafka's behavior regarding reliable message storage. Like many broker configuration variables, these can apply at the broker level, controlling configuration for all topics in the system, and at the topic level, controlling behavior for a specific topic.

Being able to control reliability trade-offs at the topic level means that the same Kafka cluster can be used to host reliable and nonreliable topics. For example, at a bank, the administrator will probably want to set very reliable defaults for the entire cluster but make an exception to the topic that stores customer complaints where some data loss is acceptable.

Let's look at these configuration parameters one by one and see how they affect the reliability of message storage in Kafka and the trade-offs involved.

Replication Factor

The topic-level configuration is `replication.factor`. At the broker level, we control the `default.replication.factor` for automatically created topics.

Until this point in the book, we have assumed that topics had a replication factor of three, meaning that each partition is replicated three times on three different brokers. This was a reasonable assumption, as this is Kafka's default, but this is a configuration that users can modify. Even after a topic exists, we can choose to add or remove replicas and thereby modify the replication factor using Kafka's replica assignment tool.

A replication factor of *N* allows us to lose *N*-1 brokers while still being able to read and write data to the topic. So a higher replication factor leads to higher availability, higher reliability, and fewer disasters. On the flip side, for a replication factor of *N*, we will need at least *N* brokers and we will store *N* copies of the data, meaning we will need *N* times as much disk space. We are basically trading availability for hardware.

So how do we determine the right number of replicas for a topic? There are a few key considerations:

Availability
: A partition with just one replica will become unavailable even during a routine restart of a single broker. The more replicas we have, the higher availability we can expect.

Durability
: Each replica is a copy of all the data in a partition. If a partition has a single replica and the disk becomes unusable for any reason, we've lost all the data in the partition. With more copies, especially on different storage devices, the probability of losing all of them is reduced.

Throughput
: With each additional replica, we multiply the inter-broker traffic. If we produce to a partition at a rate of 10 MBps, then a single replica will not generate any replication traffic. If we have 2 replicas, then we'll have 10 MBps replication traffic, with 3 replicas it will be 20 MBps, and with 5 replicas it will be 40 MBps. We need to take this into account when planning the cluster size and capacity.

End-to-end latency
: Each produced record has to be replicated to all in-sync replicas before it is available for consumers. In theory, with more replicas, there is higher probability that one of these replicas is a bit slow and therefore will slow the consumers down. In practice, if one broker becomes slow for any reason, it will slow down every client that tries using it, regardless of replication factor.

Cost

This is the most common reason for using a replication factor lower than 3 for noncritical data. The more replicas we have of our data, the higher the storage and network costs. Since many storage systems already replicate each block 3 times, it sometimes makes sense to reduce costs by configuring Kafka with a replication factor of 2. Note that this will still reduce availability compared to a replication factor of 3, but durability will be guaranteed by the storage device.

Placement of replicas is also very important. Kafka will always make sure each replica for a partition is on a separate broker. In some cases, this is not safe enough. If all replicas for a partition are placed on brokers that are on the same rack, and the top-of-rack switch misbehaves, we will lose availability of the partition regardless of the replication factor. To protect against rack-level misfortune, we recommend placing brokers in multiple racks and using the `broker.rack` broker configuration parameter to configure the rack name for each broker. If rack names are configured, Kafka will make sure replicas for a partition are spread across multiple racks in order to guarantee even higher availability. When running Kafka in cloud environments, it is common to consider availability zones as separate racks. In Chapter 6, we provided details on how Kafka places replicas on brokers and racks.

Unclean Leader Election

This configuration is only available at the broker (and in practice, cluster-wide) level. The parameter name is `unclean.leader.election.enable`, and by default it is set to `false`.

As explained earlier, when the leader for a partition is no longer available, one of the in-sync replicas will be chosen as the new leader. This leader election is "clean" in the sense that it guarantees no loss of committed data—by definition, committed data exists on all in-sync replicas.

But what do we do when no in-sync replica exists except for the leader that just became unavailable?

This situation can happen in one of two scenarios:

- The partition had three replicas, and the two followers became unavailable (let's say two brokers crashed). In this situation, as producers continue writing to the leader, all the messages are acknowledged and committed (since the leader is the one and only in-sync replica). Now let's say that the leader becomes unavailable (oops, another broker crash). In this scenario, if one of the out-of-sync followers starts first, we have an out-of-sync replica as the only available replica for the partition.
- The partition had three replicas, and due to network issues, the two followers fell behind so that even though they are up and replicating, they are no longer in

sync. The leader keeps accepting messages as the only in-sync replica. Now if the leader becomes unavailable, there are only out-of-sync replicas available to become leaders.

In both these scenarios, we need to make a difficult decision:

- If we don't allow the out-of-sync replica to become the new leader, the partition will remain offline until we bring the old leader (and the last in-sync replica) back online. In some cases (e.g., memory chip needs replacement), this can take many hours.
- If we do allow the out-of-sync replica to become the new leader, we are going to lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers. Why? Imagine that while replicas 0 and 1 were not available, we wrote messages with offsets 100–200 to replica 2 (then the leader). Now replica 2 is unavailable and replica 0 is back online. Replica 0 only has messages 0–100 but not 100–200. If we allow replica 0 to become the new leader, it will allow producers to write new messages and allow consumers to read them. So, now the new leader has completely new messages 100–200. First, let's note that some consumers may have read the old messages 100–200, some consumers got the new 100–200, and some got a mix of both. This can lead to pretty bad consequences when looking at things like downstream reports. In addition, replica 2 will come back online and become a follower of the new leader. At that point, it will delete any messages it got that don't exist on the current leader. Those messages will not be available to any consumer in the future.

In summary, if we allow out-of-sync replicas to become leaders, we risk data loss and inconsistencies. If we don't allow them to become leaders, we face lower availability as we must wait for the original leader to become available before the partition is back online.

By default, `unclean.leader.election.enable` is set to false, which will not allow out-of-sync replicas to become leaders. This is the safest option since it provides the best guarantees against data loss. It does mean that in the extreme unavailability scenarios that we described previously, some partitions will remain unavailable until manually recovered. It is always possible for an administrator to look at the situation, decide to accept the data loss in order to make the partitions available, and switch this configuration to true before starting the cluster. Just don't forget to turn it back to false after the cluster recovered.

Minimum In-Sync Replicas

Both the topic and the broker-level configuration are called `min.insync.replicas`.

As we've seen, there are cases where even though we configured a topic to have three replicas, we may be left with a single in-sync replica. If this replica becomes unavailable, we may have to choose between availability and consistency. This is never an easy choice. Note that part of the problem is that, per Kafka reliability guarantees, data is considered committed when it is written to all in-sync replicas, even when `all` means just one replica and the data could be lost if that replica is unavailable.

When we want to be sure that committed data is written to more than one replica, we need to set the minimum number of in-sync replicas to a higher value. If a topic has three replicas and we set `min.insync.replicas` to 2, then producers can only write to a partition in the topic if at least two out of the three replicas are in sync.

When all three replicas are in sync, everything proceeds normally. This is also true if one of the replicas becomes unavailable. However, if two out of three replicas are not available, the brokers will no longer accept produce requests. Instead, producers that attempt to send data will receive `NotEnoughReplicasException`. Consumers can continue reading existing data. In effect, with this configuration, a single in-sync replica becomes read-only. This prevents the undesirable situation where data is produced and consumed, only to disappear when unclean election occurs. In order to recover from this read-only situation, we must make one of the two unavailable partitions available again (maybe restart the broker) and wait for it to catch up and get in sync.

Keeping Replicas In Sync

As mentioned earlier, out-of-sync replicas decrease the overall reliability, so it is important to avoid these as much as possible. We also explained that a replica can become out of sync in one of two ways: either it loses connectivity to ZooKeeper or it fails to keep up with the leader and builds up a replication lag. Kafka has two broker configurations that control the sensitivity of the cluster to these two conditions.

`zookeeper.session.timeout.ms` is the time interval during which a Kafka broker can stop sending heartbeats to ZooKeeper without ZooKeeper considering the broker dead and removing it from the cluster. In version 2.5.0, this value was increased from 6 seconds to 18 seconds, in order to increase the stability of Kafka clusters in cloud environments where network latencies show higher variance. In general, we want this time to be high enough to avoid random flapping caused by garbage collection or network conditions, but still low enough to make sure brokers that are actually frozen will be detected in a timely manner.

If a replica did not fetch from the leader or did not catch up to the latest messages on the leader for longer than `replica.lag.time.max.ms`, it will become out of sync. This was increased from 10 seconds to 30 seconds in release 2.5.0 to improve resilience of the cluster and avoid unnecessary flapping. Note that this higher value also impacts maximum latency for the consumer—with the higher value it can take up to 30

seconds until a message arrives to all replicas and the consumers are allowed to consume it.

Persisting to Disk

We've mentioned a few times that Kafka will acknowledge messages that were not persisted to disk, depending just on the number of replicas that received the message. Kafka will flush messages to disk when rotating segments (by default 1 GB in size) and before restarts but will otherwise rely on Linux page cache to flush messages when it becomes full. The idea behind this is that having three machines in separate racks or availability zones, each with a copy of the data, is safer than writing the messages to disk on the leader, because simultaneous failures on two different racks or zones are so unlikely. However, it is possible to configure the brokers to persist messages to disk more frequently. The configuration parameter `flush.messages` allows us to control the maximum number of messages not synced to disk, and `flush.ms` allows us to control the frequency of syncing to disk. Before using this feature, it is worth reading how `fsync` impacts Kafka's throughput and how to mitigate its drawbacks.

Using Producers in a Reliable System

Even if we configure the brokers in the most reliable configuration possible, the system as a whole can still potentially lose data if we don't configure the producers to be reliable as well.

Here are two example scenarios to demonstrate this:

- We configured the brokers with three replicas, and unclean leader election is disabled. So we should never lose a single message that was committed to the Kafka cluster. However, we configured the producer to send messages with `acks=1`. We sent a message from the producer, and it was written to the leader but not yet to the in-sync replicas. The leader sent back a response to the producer saying, "Message was written successfully" and immediately crashes before the data was replicated to the other replicas. The other replicas are still considered in sync (remember that it takes a while before we declare a replica out of sync), and one of them will become the leader. Since the message was not written to the replicas, it was lost. But the producing application thinks it was written successfully. The system is consistent because no consumer saw the message (it was never committed because the replicas never got it), but from the producer perspective, a message was lost.
- We configured the brokers with three replicas, and unclean leader election is disabled. We learned from our mistakes and started producing messages with `acks=all`. Suppose that we are attempting to write a message to Kafka, but the

leader for the partition we are writing to just crashed and a new one is still getting elected. Kafka will respond with "Leader not Available." At this point, if the producer doesn't handle the error correctly and doesn't retry until the write is successful, the message may be lost. Once again, this is not a broker reliability issue because the broker never got the message; and it is not a consistency issue because the consumers never got the message either. But if producers don't handle errors correctly, they may cause message loss.

As the examples show, there are two important things that everyone who writes applications that produce to Kafka must pay attention to:

- Use the correct `acks` configuration to match reliability requirements
- Handle errors correctly both in configuration and in code

We discussed producer configuration in depth in Chapter 3, but let's go over the important points again.

Send Acknowledgments

Producers can choose between three different acknowledgment modes:

`acks=0`
: This means that a message is considered to be written successfully to Kafka if the producer managed to send it over the network. We will still get errors if the object we are sending cannot be serialized or if the network card failed, but we won't get any error if the partition is offline, a leader election is in progress, or even if the entire Kafka cluster is unavailable. Running with `acks=0` has low produce latency (which is why we see a lot of benchmarks with this configuration), but it will not improve end-to-end latency (remember that consumers will not see messages until they are replicated to all available replicas).

`acks=1`
: This means that the leader will send either an acknowledgment or an error the moment it gets the message and writes it to the partition data file (but not necessarily synced to disk). We can lose data if the leader shuts down or crashes and some messages that were successfully written to the leader and acknowledged were not replicated to the followers before the crash. With this configuration, it is also possible to write to the leader faster than it can replicate messages and end up with under-replicated partitions, since the leader will acknowledge messages from the producer before replicating them.

`acks=all`
: This means that the leader will wait until all in-sync replicas get the message before sending back an acknowledgment or an error. In conjunction with the

`min.insync.replicas` configuration on the broker, this lets us control how many replicas get the message before it is acknowledged. This is the safest option—the producer won't stop trying to send the message before it is fully committed. This is also the option with the longest producer latency—the producer waits for all in-sync replicas to get all the messages before it can mark the message batch as "done" and carry on.

Configuring Producer Retries

There are two parts to handling errors in the producer: the errors that the producers handle automatically for us and the errors that we, as developers using the producer library, must handle.

The producer can handle *retriable* errors. When the producer sends messages to a broker, the broker can return either a success or an error code. Those error codes belong to two categories—errors that can be resolved after retrying and errors that won't be resolved. For example, if the broker returns the error code `LEADER_NOT_AVAILABLE`, the producer can try sending the message again—maybe a new broker was elected and the second attempt will succeed. This means that `LEADER_NOT_AVAILABLE` is a *retriable* error. On the other hand, if a broker returns an `INVALID_CONFIG` exception, trying the same message again will not change the configuration. This is an example of a *nonretriable error*.

In general, when our goal is to never lose a message, our best approach is to configure the producer to keep trying to send the messages when it encounters a retriable error. And the best approach to retries, as recommended in Chapter 3, is to leave the number of retries at its current default (`MAX_INT`, or effectively infinite) and use `delivery.timout.ms` to configure the maximum amount of time we are willing to wait until giving up on sending a message—the producer will retry sending the message as many times as possible within this time interval.

Retrying to send a failed message includes a risk that both messages were successfully written to the broker, leading to duplicates. Retries and careful error handling can guarantee that each message will be stored *at least once*, but not *exactly once*. Using `enable.idempotence=true` will cause the producer to include additional information in its records, which brokers will use to skip duplicate messages caused by retries. In Chapter 8, we discuss in detail how and when this works.

Additional Error Handling

Using the built-in producer retries is an easy way to correctly handle a large variety of errors without loss of messages, but as developers, we must still be able to handle other types of errors. These include:

- Nonretriable broker errors, such as errors regarding message size, authorization errors, etc.
- Errors that occur before the message was sent to the broker—for example, serialization errors
- Errors that occur when the producer exhausted all retry attempts or when the available memory used by the producer is filled to the limit due to using all of it to store messages while retrying
- Timeouts

In Chapter 3 we discussed how to write error handlers for both sync and async message-sending methods. The content of these error handlers is specific to the application and its goals—do we throw away "bad messages"? Log errors? Stop reading messages from the source system? Apply back pressure to the source system to stop sending messages for a while? Store these messages in a directory on the local disk? These decisions depend on the architecture and the product requirements. Just note that if all the error handler is doing is retrying to send the message, then we'll be better off relying on the producer's retry functionality.

Using Consumers in a Reliable System

Now that we have learned how to produce data while taking Kafka's reliability guarantees into account, it is time to see how to consume data.

As we saw in the first part of this chapter, data is only available to consumers after it has been committed to Kafka—meaning it was written to all in-sync replicas. This means that consumers get data that is guaranteed to be consistent. The only thing consumers are left to do is make sure they keep track of which messages they've read and which messages they haven't. This is key to not losing messages while consuming them.

When reading data from a partition, a consumer is fetching a batch of messages, checking the last offset in the batch, and then requesting another batch of messages starting from the last offset received. This guarantees that a Kafka consumer will always get new data in correct order without missing any messages.

When a consumer stops, another consumer needs to know where to pick up the work—what was the last offset that the previous consumer processed before it stopped? The "other" consumer can even be the original one after a restart. It doesn't really matter—some consumer is going to pick up consuming from that partition, and it needs to know at which offset to start. This is why consumers need to "commit" their offsets. For each partition it is consuming, the consumer stores its current location, so it or another consumer will know where to continue after a restart. The main way consumers can lose messages is when committing offsets for events they've read

but haven't completely processed yet. This way, when another consumer picks up the work, it will skip those messages and they will never get processed. This is why paying careful attention to when and how offsets get committed is critical.

Committed Messages Versus Committed Offsets

This is different from a *committed message*, which, as discussed previously, is a message that was written to all in-sync replicas and is available to consumers. *Committed offsets* are offsets the consumer sent to Kafka to acknowledge that it received and processed all the messages in a partition up to this specific offset.

In Chapter 4, we discussed the Consumer API in detail and covered the many methods for committing offsets. Here we will cover some important considerations and choices, but refer back to Chapter 4 for details on using the APIs.

Important Consumer Configuration Properties for Reliable Processing

There are four consumer configuration properties that are important to understand in order to configure our consumer for a desired reliability behavior.

The first is `group.id`, as explained in great detail in Chapter 4. The basic idea is that if two consumers have the same group ID and subscribe to the same topic, each will be assigned a subset of the partitions in the topic and will therefore only read a subset of the messages individually (but all the messages will be read by the group as a whole). If we need a consumer to see, on its own, every single message in the topics it is subscribed to, it will need a unique `group.id`.

The second relevant configuration is `auto.offset.reset`. This parameter controls what the consumer will do when no offsets were committed (e.g., when the consumer first starts) or when the consumer asks for offsets that don't exist in the broker (Chapter 4 explains how this can happen). There are only two options here. If we choose `earliest`, the consumer will start from the beginning of the partition whenever it doesn't have a valid offset. This can lead to the consumer processing a lot of messages twice, but it guarantees to minimize data loss. If we choose `latest`, the consumer will start at the end of the partition. This minimizes duplicate processing by the consumer but almost certainly leads to some messages getting missed by the consumer.

The third relevant configuration is `enable.auto.commit`. This is a big decision: are we going to let the consumer commit offsets for us based on schedule, or are we planning on committing offsets manually in our code? The main benefit of automatic offset commits is that it's one less thing to worry about when using consumers in our application. When we do all the processing of consumed records within the consumer poll loop, then the automatic offset commit guarantees we will never accidentally commit an offset that we didn't process. The main drawbacks of automatic offset

commits is that we have no control over the number of duplicate records the application may process because it was stopped after processing some records but before the automated commit kicked in. When the application has more complex processing, such as passing records to another thread to process in the background, there is no choice but to use manual offset commit since the automatic commit may commit offsets for records the consumer has read but perhaps has not processed yet.

The fourth relevant configuration, `auto.commit.interval.ms`, is tied to the third. If we choose to commit offsets automatically, this configuration lets us configure how frequently they will be committed. The default is every five seconds. In general, committing more frequently adds overhead but reduces the number of duplicates that can occur when a consumer stops.

While not directly related to reliable data processing, it is difficult to consider a consumer reliable if it frequently stops consuming in order to rebalance. Chapter 4 includes advice on how to configure consumers to minimize unnecessary rebalancing and to minimize pauses while rebalancing.

Explicitly Committing Offsets in Consumers

If we decide we need more control and choose to commit offsets manually, we need to be concerned about correctness and performance implications.

We will not go over the mechanics and APIs involved in committing offsets here, since they were covered in great depth in Chapter 4. Instead, we will review important considerations when developing a consumer to handle data reliably. We'll start with the simple and perhaps obvious points and move on to more complex patterns.

Always commit offsets after messages were processed

If we do all the processing within the poll loop and don't maintain state between poll loops (e.g., for aggregation), this should be easy. We can use the auto-commit configuration, commit offset at the end of the poll loop, or commit offset inside the loop at a frequency that balances requirements for both overhead and lack of duplicate processing. If there are additional threads or stateful processing involved, this becomes more complex, especially since the consumer object is not thread safe. In Chapter 4, we discussed how this can be done and provided references with additional examples.

Commit frequency is a trade-off between performance and number of duplicates in the event of a crash

Even in the simplest case where we do all the processing within the poll loop and don't maintain state between poll loops, we can choose to commit multiple times within a loop or choose to only commit every several loops. Committing has significant performance overhead. It is similar to produce with `acks=all`, but all offset commits of a single consumer group are produced to the same broker, which can

become overloaded. The commit frequency has to balance requirements for performance and lack of duplicates. Committing after every message should only ever be done on very low-throughput topics.

Commit the right offsets at the right time

A common pitfall when committing in the middle of the poll loop is accidentally committing the last offset read when polling and not the offset after the last offset processed. Remember that it is critical to always commit offsets for messages after they were processed—committing offsets for messages read but not processed can lead to the consumer missing messages. Chapter 4 has examples that show how to do just that.

Rebalances

When designing an application, we need to remember that consumer rebalances will happen, and we need to handle them properly. Chapter 4 contains a few examples. This usually involves committing offsets before partitions are revoked and cleaning any state the application maintains when it is assigned new partitions.

Consumers may need to retry

In some cases, after calling poll and processing records, some records are not fully processed and will need to be processed later. For example, we may try to write records from Kafka to a database but find that the database is not available at that moment and we need to retry later. Note that unlike traditional pub/sub messaging systems, Kafka consumers commit offsets and do not "ack" individual messages. This means that if we failed to process record #30 and succeeded in processing record #31, we should not commit offset #31—this would result in marking as processed all the records up to #31 including #30, which is usually not what we want. Instead, try following one of the following two patterns.

One option when we encounter a retriable error is to commit the last record we processed successfully. We'll then store the records that still need to be processed in a buffer (so the next poll won't override them), use the consumer `pause()` method to ensure that additional polls won't return data, and keep trying to process the records.

A second option when encountering a retriable error is to write it to a separate topic and continue. A separate consumer group can be used to handle retries from the retry topic, or one consumer can subscribe to both the main topic and to the retry topic but pause the retry topic between retries. This pattern is similar to the dead-letter-queue system used in many messaging systems.

Consumers may need to maintain state

In some applications, we need to maintain state across multiple calls to poll. For example, if we want to calculate moving average, we'll want to update the average after every time we poll Kafka for new messages. If our process is restarted, we will need to not just start consuming from the last offset, but we'll also need to recover the matching moving average. One way to do this is to write the latest accumulated value to a "results" topic at the same time the application is committing the offset. This means that when a thread is starting up, it can pick up the latest accumulated value when it starts and pick up right where it left off. In Chapter 8, we discuss how an application can write results and commit offsets in a single transaction. In general, this is a rather complex problem to solve, and we recommend looking at a library like Kafka Streams or Flink, which provides high-level DSL-like APIs for aggregation, joins, windows, and other complex analytics.

Validating System Reliability

Once we have gone through the process of figuring out our reliability requirements, configuring the brokers, configuring the clients, and using the APIs in the best way for our use case, we can just relax and run everything in production, confident that no event will ever be missed, right?

We recommend doing some validation first and suggest three layers of validation: validate the configuration, validate the application, and monitor the application in production. Let's look at each of these steps and see what we need to validate and how.

Validating Configuration

It is easy to test the broker and client configuration in isolation from the application logic, and it is recommended to do so for two reasons:

- It helps to test if the configuration we've chosen can meet our requirements.
- It is a good exercise to reason through the expected behavior of the system.

Kafka includes two important tools to help with this validation. The `org.apache.kafka.tools` package includes `VerifiableProducer` and `VerifiableConsumer` classes. These can run as command-line tools or be embedded in an automated testing framework.

The idea is that the verifiable producer produces a sequence of messages containing numbers from 1 to a value we choose. We can configure the verifiable producer the same way we configure our own producer, setting the right number of `acks`, `retries`, `delivery.timeout.ms`, and rate at which the messages will be produced. When we

run it, it will print success or error for each message sent to the broker, based on the `acks` received. The verifiable consumer performs the complementary check. It consumes events (usually those produced by the verifiable producer) and prints out the events it consumed in order. It also prints information regarding commits and rebalances.

It is important to consider which tests we want to run. For example:

- Leader election: what happens if we kill the leader? How long does it take the producer and consumer to start working as usual again?
- Controller election: how long does it take the system to resume after a restart of the controller?
- Rolling restart: can we restart the brokers one by one without losing any messages?
- Unclean leader election test: what happens when we kill all the replicas for a partition one by one (to make sure each goes out of sync) and then start a broker that was out of sync? What needs to happen in order to resume operations? Is this acceptable?

Then we pick a scenario, start the verifiable producer, start the verifiable consumer, and run through the scenario—for example, kill the leader of the partition we are producing data into. If we expected a short pause and then everything to resume normally with no message loss, we need to make sure the number of messages produced by the producer and the number of messages consumed by the consumer match.

The Apache Kafka source repository includes an extensive test suite. Many of the tests in the suite are based on the same principle and use the verifiable producer and consumer to make sure rolling upgrades work.

Validating Applications

Once we are sure the broker and client configuration meet our requirements, it is time to test whether the application provides the guarantees we need. This will check things like custom error-handling code, offset commits, and rebalance listeners and similar places where the application logic interacts with Kafka's client libraries.

Naturally, because application logic can vary considerably, there is only so much guidance we can provide on how to test it. We recommend integration tests for the application as part of any development process, and we recommend running tests under a variety of failure conditions:

- Clients lose connectivity to one of the brokers
- High latency between client and broker

- Disk full
- Hanging disk (also called "brown out")
- Leader election
- Rolling restart of brokers
- Rolling restart of consumers
- Rolling restart of producers

There are many tools that can be used to introduce network and disk faults, and many are excellent, so we will not attempt to make specific recommendations. Apache Kafka itself includes the Trogdor test framework for fault injection. For each scenario, we will have *expected behavior*, which is what we planned on seeing when we developed the application. Then we run the test to see what actually happens. For example, when planning for a rolling restart of consumers, we planned for a short pause as consumers rebalance and then continue consumption with no more than 1,000 duplicate values. Our test will show whether the way the application commits offsets and handles rebalances actually works this way.

Monitoring Reliability in Production

Testing the application is important, but it does not replace the need to continuously monitor production systems to make sure data is flowing as expected. Chapter 12 will cover detailed suggestions on how to monitor the Kafka cluster, but in addition to monitoring the health of the cluster, it is important to also monitor the clients and the flow of data through the system.

Kafka's Java clients include JMX metrics that allow monitoring client-side status and events. For the producers, the two metrics most important for reliability are error-rate and retry-rate per record (aggregated). Keep an eye on those, since error or retry rates going up can indicate an issue with the system. Also monitor the producer logs for errors that occur while sending events that are logged at `WARN` level, and say something along the lines of "Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (two attempts left). Error: …" When we see events with 0 attempts left, the producer is running out of retries. In Chapter 3 we discussed how to configure `delivery.timeout.ms` and `retries` to improve the error handling in the producer and avoid running out of retries prematurely. Of course, it is always better to solve the problem that caused the errors in the first place. `ERROR` level log messages on the producer are likely to indicate that sending the message failed completely due to nonretriable error, a retriable error that ran out of retries, or a timeout. When applicable, the exact error from the broker will be logged as well.

On the consumer side, the most important metric is consumer lag. This metric indicates how far the consumer is from the latest message committed to the partition on

the broker. Ideally, the lag would always be zero and the consumer will always read the latest message. In practice, because calling `poll()` returns multiple messages and then the consumer spends time processing them before fetching more messages, the lag will always fluctuate a bit. What is important is to make sure consumers do eventually catch up rather than fall further and further behind. Because of the expected fluctuation in consumer lag, setting traditional alerts on the metric can be challenging. Burrow is a consumer lag checker by LinkedIn and can make this easier.

Monitoring flow of data also means making sure all produced data is consumed in a timely manner ("timely manner" is usually based on business requirements). In order to make sure data is consumed in a timely manner, we need to know when the data was produced. Kafka assists in this: starting with version 0.10.0, all messages include a timestamp that indicates when the event was produced (although note that this can be overridden either by the application that is sending the events or by the brokers themselves if they are configured to do so).

To make sure all produced messages are consumed within a reasonable amount of time, we will need the application producing the messages to record the number of events produced (usually as events per second). The consumers need to record the number of events consumed per unit or time, and the lag from the time events were produced to the time they were consumed, using the event timestamp. Then we will need a system to reconcile the events per second numbers from both the producer and the consumer (to make sure no messages were lost on the way) and to make sure the interval between produce time and consume time is reasonable. This type of end-to-end monitoring systems can be challenging and time-consuming to implement. To the best of our knowledge, there is no open source implementation of this type of system, but Confluent provides a commercial implementation as part of the Confluent Control Center.

In addition to monitoring clients and the end-to-end flow of data, Kafka brokers include metrics that indicate the rate of error responses sent from the brokers to clients. We recommend collecting `kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec` and `kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec`. At times, some level of error responses is expected—for example, if we shut down a broker for maintenance and new leaders are elected on another broker, it is expected that producers will receive a `NOT_LEADER_FOR_PARTITION` error, which will cause them to request updated metadata before continuing to produce events as usual. Unexplained increases in failed requests should always be investigated. To assist in such investigations, the failed requests metrics are tagged with the specific error response that the broker sent.

Summary

As we said in the beginning of the chapter, reliability is not just a matter of specific Kafka features. We need to build an entire reliable system, including the application architecture, the way applications use the producer and Consumer APIs, producer and consumer configuration, topic configuration, and broker configuration. Making the system more reliable always has trade-offs in application complexity, performance, availability, or disk-space usage. By understanding all the options and common patterns and understanding requirements for each use case, we can make informed decisions regarding how reliable the application and Kafka deployment need to be and which trade-offs make sense.

CHAPTER 8
Exactly-Once Semantics

In Chapter 7 we discussed the configuration parameters and the best practices that allow Kafka users to control Kafka's reliability guarantees. We focused on at-least-once delivery—the guarantee that Kafka will not lose messages that it acknowledged as committed. This still leaves open the possibility of duplicate messages.

In simple systems where messages are produced and then consumed by various applications, duplicates are an annoyance that is fairly easy to handle. Most real-world applications contain unique identifiers that consuming applications can use to deduplicate the messages.

Things become more complicated when we look at stream processing applications that aggregate events. When inspecting an application that consumes events, computes an average, and produces the results, it is often impossible for those who check the results to detect that the average is incorrect because an event was processed twice while computing the average. In these cases, it is important to provide a stronger guarantee—exactly-once processing semantics.

In this chapter, we will discuss how to use Kafka with exactly-once semantics, the recommended use cases, and the limitations. As we did with at-least-once guarantees, we will dive a bit deeper and provide some insight and intuition into how this guarantee is implemented. These details can be skipped when you first read the chapter but will be useful to understand before using the feature—it will help clarify the meaning of the different configurations and APIs and how best to use them.

Exactly-once semantics in Kafka is a combination of two key features: idempotent producers, which help avoid duplicates caused by producer retries, and transactional semantics, which guarantee exactly-once processing in stream processing applications. We will discuss both, starting with the simpler and more generally useful idempotent producer.

Idempotent Producer

A service is called idempotent if performing the same operation multiple times has the same result as performing it a single time. In databases it is usually demonstrated as the difference between `UPDATE t SET x=x+1 where y=5` and `UPDATE t SET x=18 where y=5`. The first example is not idempotent; if we call it three times, we'll end up with a very different result than if we were to call it once. The second example is idempotent—no matter how many times we run this statement, `x` will be equal to 18.

How is this related to a Kafka producer? If we configure a producer to have at-least-once semantics rather than idempotent semantics, it means that in cases of uncertainty, the producer will retry sending the message so it will arrive at least once. These retries could lead to duplicates.

The classic case is when a partition leader received a record from the producer, replicated it successfully to the followers, and then the broker on which the leader resides crashed before it could send a response to the producer. The producer, after a certain time without a response, will resend the message. The message will arrive at the new leader, who already has a copy of the message from the previous attempt—resulting in a duplicate.

In some applications duplicates don't matter much, but in others they can lead to inventory miscounts, bad financial statements, or sending someone two umbrellas instead of the one they ordered.

Kafka's idempotent producer solves this problem by automatically detecting and resolving such duplicates.

How Does the Idempotent Producer Work?

When we enable the idempotent producer, each message will include a unique identified producer ID (PID) and a sequence number. These, together with the target topic and partition, uniquely identify each message. Brokers use these unique identifiers to track the last five messages produced to every partition on the broker. To limit the number of previous sequence numbers that have to be tracked for each partition, we also require that the producers will use `max.inflight.requests=5` or lower (the default is 5).

When a broker receives a message that it already accepted before, it will reject the duplicate with an appropriate error. This error is logged by the producer and is reflected in its metrics but does not cause any exception and should not cause any alarm. On the producer client, it will be added to the `record-error-rate` metric. On the broker, it will be part of the `ErrorsPerSec` metric of the `RequestMetrics` type, which includes a separate count for each type of error.

What if a broker receives a sequence number that is unexpectedly high? The broker expects message number 2 to be followed by message number 3; what happens if the broker receives message number 27 instead? In such cases the broker will respond with an "out of order sequence" error, but if we use an idempotent producer without using transactions, this error can be ignored.

While the producer will continue normally after encountering an "out of order sequence number" exception, this error typically indicates that messages were lost between the producer and the broker—if the broker received message number 2 followed by message number 27, something must have happened to messages 3 to 26. When encountering such an error in the logs, it is worth revisiting the producer and topic configuration and making sure the producer is configured with recommended values for high reliability and to check whether unclean leader election has occurred.

As is always the case with distributed systems, it is interesting to consider the behavior of an idempotent producer under failure conditions. Consider two cases: producer restart and broker failure.

Producer restart

When a producer fails, usually a new producer will be created to replace it—whether manually by a human rebooting a machine, or using a more sophisticated framework like Kubernetes that provides automated failure recovery. The key point is that when the producer starts, if the idempotent producer is enabled, the producer will initialize and reach out to a Kafka broker to generate a producer ID. Each initialization of a producer will result in a completely new ID (assuming that we did not enable transactions). This means that if a producer fails and the producer that replaces it sends a message that was previously sent by the old producer, the broker will not detect the duplicates—the two messages will have different producer IDs and different sequence numbers and will be considered as two different messages. Note that the same is true if the old producer froze and then came back to life after its replacement started—the original producer is not recognized as a zombie, because we have two totally different producers with different IDs.

Broker failure

When a broker fails, the controller elects new leaders for the partitions that had leaders on the failed broker. Say that we have a producer that produced messages to topic A, partition 0, which had its lead replica on broker 5 and a follower replica on broker 3. After broker 5 fails, broker 3 becomes the new leader. The producer will discover that the new leader is broker 3 via the metadata protocol and start producing to it.

But how will broker 3 know which sequences were already produced in order to reject duplicates?

The leader keeps updating its in-memory producer state with the five last sequence IDs every time a new message is produced. Follower replicas update their own in-memory buffers every time they replicate new messages from the leader. This means that when a follower becomes a leader, it already has the latest sequence numbers in memory, and validation of newly produced messages can continue without any issues or delays.

But what happens when the old leader comes back? After a restart, the old in-memory producer state will no longer be in memory. To assist in recovery, brokers take a snapshot of the producer state to a file when they shut down or every time a segment is created. When the broker starts, it reads the latest state from a file. The newly restarted broker then keeps updating the producer state as it catches up by replicating from the current leader, and it has the most current sequence IDs in memory when it is ready to become a leader again.

What if a broker crashed and the last snapshot is not updated? Producer ID and sequence ID are also part of the message format that is written to Kafka's logs. During crash recovery, the producer state will be recovered by reading the older snapshot and also messages from the latest segment of each partition. A new snapshot will be stored as soon as the recovery process completes.

An interesting question is what happens if there are no messages? Imagine that a certain topic has two hours of retention time, but no new messages arrived in the last two hours—there will be no messages to use to recover the state if a broker crashed. Luckily, no messages also means no duplicates. We will start accepting messages immediately (while logging a warning about the lack of state), and create the producer state from the new messages that arrive.

Limitations of the Idempotent Producer

Kafka's idempotent producer only prevents duplicates in case of retries that are caused by the producer's internal logic. Calling `producer.send()` twice with the same message will create a duplicate, and the idempotent producer won't prevent it. This is because the producer has no way of knowing that the two records that were sent are in fact the same record. It is always a good idea to use the built-in retry mechanism of the producer rather than catching producer exceptions and retrying from the application itself; the idempotent producer makes this pattern even more appealing—it is the easiest way to avoid duplicates when retrying.

It is also rather common to have applications that have multiple instances or even one instance with multiple producers. If two of these producers attempt to send identical messages, the idempotent producer will not detect the duplication. This scenario is fairly common in applications that get data from a source—a directory with files, for instance—and produce it to Kafka. If the application happened to have two instances reading the same file and producing records to Kafka, we will get multiple copies of the records in that file.

The idempotent producer will only prevent duplicates caused by the retry mechanism of the producer itself, whether the retry is caused by producer, network, or broker errors. But nothing else.

How Do I Use the Kafka Idempotent Producer?

This is the easy part. Add `enable.idempotence=true` to the producer configuration. If the producer is already configured with `acks=all`, there will be no difference in performance. By enabling idempotent producer, the following things will change:

- To retrieve a producer ID, the producer will make one extra API call when starting up.
- Each record batch sent will include the producer ID and the sequence ID for the first message in the batch (sequence IDs for each message in the batch are derived from the sequence ID of the first message plus a delta). These new fields add 96 bits to each record batch (producer ID is a long, and sequence is an integer), which is barely any overhead for most workloads.
- Brokers will validate the sequence numbers from any single producer instance and guarantee the lack of duplicate messages.
- The order of messages produced to each partition will be guaranteed, through all failure scenarios, even if `max.in.flight.requests.per.connection` is set to more than 1 (5 is the default and also the highest value supported by the idempotent producer).

Idempotent producer logic and error handling improved significantly in version 2.5 (both on the producer side and the broker side) as a result of KIP-360. Prior to release 2.5, the producer state was not always maintained for long enough, which resulted in fatal UNKNOWN_PRODUCER_ID errors in various scenarios (partition reassignment had a known edge case where the new replica became the leader before any writes happened from a specific producer, meaning that the new leader had no state for that partition). In addition, previous versions attempted to rewrite the sequence IDs in some error scenarios, which could lead to duplicates. In newer versions, if we encounter a fatal error for a record batch, this batch and all the batches that are in flight will be rejected. The user who writes the application can handle the exception and decide whether to skip those records or retry and risk duplicates and reordering.

Transactions

As we mentioned in the introduction to this chapter, transactions were added to Kafka to guarantee the correctness of applications developed using Kafka Streams. In order for a stream processing application to generate correct results, each input record must be processed exactly one time, and its processing result will be reflected exactly one time, even in case of failure. Transactions in Apache Kafka allow stream processing applications to generate accurate results. This, in turn, enables developers to use stream processing applications in use cases where accuracy is a key requirement.

It is important to keep in mind that transactions in Kafka were developed specifically for stream processing applications. And therefore they were built to work with the "consume-process-produce" pattern that forms the basis of stream processing applications. Use of transactions can guarantee exactly-once semantics in this context—the processing of each input record will be considered complete after the application's internal state has been updated and the results were successfully produced to output topics. In "What Problems Aren't Solved by Transactions?" on page 191, we'll explore a few scenarios where Kafka's exactly-once guarantees will not apply.

Transactions is the name of the underlying mechanism. Exactly-once semantics or exactly-once guarantees is the behavior of a stream processing application. Kafka Streams uses transactions to implement its exactly-once guarantees. Other stream processing frameworks, such as Spark Streaming or Flink, use different mechanisms to provide their users with exactly-once semantics.

Transactions Use Cases

Transactions are useful for any stream processing application where accuracy is important, and especially where stream processing includes aggregation and/or joins. If the stream processing application only performs single record transformation and filtering, there is no internal state to update, and even if duplicates were introduced in the process, it is fairly straightforward to filter them out of the output stream. When the stream processing application aggregates several records into one, it is much more difficult to check whether a result record is wrong because some input records were counted more than once; it is impossible to correct the result without reprocessing the input.

Financial applications are typical examples of complex stream processing applications where exactly-once capabilities are used to guarantee accurate aggregation. However, because it is rather trivial to configure any Kafka Streams application to provide exactly-once guarantees, we've seen it enabled in more mundane use cases, including, for instance, chatbots.

What Problems Do Transactions Solve?

Consider a simple stream processing application: it reads events from a source topic, maybe processes them, and writes results to another topic. We want to be sure that for each message we process, the results are written exactly once. What can possibly go wrong?

It turns out that quite a few things could go wrong. Let's look at two scenarios.

Reprocessing caused by application crashes

After consuming a message from the source cluster and processing it, the application has to do two things: produce the result to the output topic, and commit the offset of the message that we consumed. Suppose that these two separate actions happen in this order. What happens if the application crashes after the output was produced but before the offset of the input was committed?

In Chapter 4, we discussed what happens when a consumer crashes. After a few seconds, the lack of heartbeat will trigger a rebalance, and the partitions the consumer was consuming from will be reassigned to a different consumer. That consumer will begin consuming records from those partitions, starting at the last committed offset. This means that all the records that were processed by the application between the last committed offset and the crash will be processed again, and the results will be written to the output topic again—resulting in duplicates.

Reprocessing caused by zombie applications

What happens if our application just consumed a batch of records from Kafka and then froze or lost connectivity to Kafka before doing anything else with this batch of records?

Just like in the previous scenario, after several heartbeats are missed, the application will be assumed dead and its partitions reassigned to another consumer in the consumer group. That consumer will reread that batch of records, process it, produce the results to an output topic, and continue on.

Meanwhile, the first instance of the application—the one that froze—may resume its activity: process the batch of records it recently consumed, and produce the results to the output topic. It can do all that before it polls Kafka for records or sends a heartbeat and discovers that it is supposed to be dead and another instance now owns those partitions.

A consumer that is dead but doesn't know it is called a zombie. In this scenario, we can see that without additional guarantees, zombies can produce data to the output topic and cause duplicate results.

How Do Transactions Guarantee Exactly-Once?

Take our simple stream processing application. It reads data from one topic, processes it, and writes the result to another topic. Exactly-once processing means that consuming, processing, and producing are done *atomically*. Either the offset of the original message is committed and the result is successfully produced or neither of these things happen. We need to make sure that partial results—where the offset is committed but the result isn't produced, or vice versa—can't happen.

To support this behavior, Kafka transactions introduce the idea of *atomic multipartition writes*. The idea is that committing offsets and producing results both involve writing messages to partitions. However, the results are written to an output topic, and offsets are written to the `_consumer_offsets` topic. If we can open a transaction, write both messages, and commit if both were written successfully—or abort to retry if they were not—we will get the exactly-once semantics that we are after.

Figure 8-1 illustrates a simple stream processing application, performing an atomic multipartition write to two partitions while also committing offsets for the event it consumed.

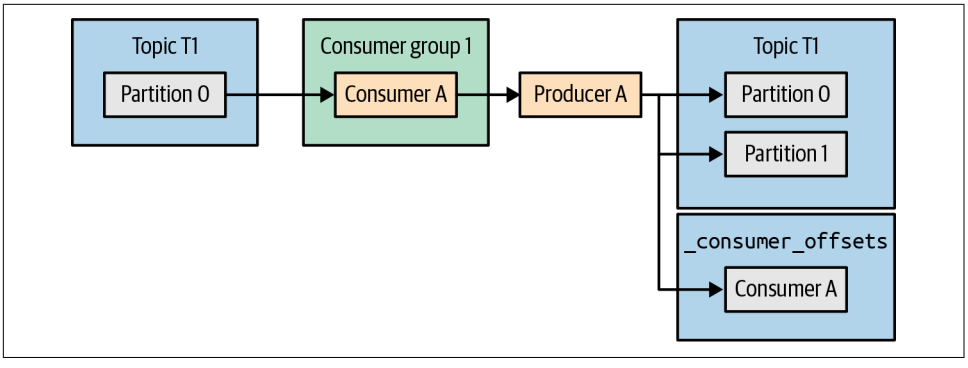

Figure 8-1. Transactional producer with atomic multipartition write

To use transactions and perform atomic multipartition writes, we use a *transactional producer*. A transactional producer is simply a Kafka producer that is configured with a `transactional.id` and has been initialized using `initTransactions()`. Unlike `producer.id`, which is generated automatically by Kafka brokers, `transactional.id` is part of the producer configuration and is expected to persist between restarts. In fact, the main role of the `transactional.id` is to identify the same producer across restarts. Kafka brokers maintain `transactional.id` to `producer.id` mapping, so if `initTransactions()` is called again with an existing `transactional.id`, the producer will also be assigned the same `producer.id` instead of a new random number.

Preventing zombie instances of the application from creating duplicates requires a mechanism for *zombie fencing*, or preventing zombie instances of the application from writing results to the output stream. The usual way of fencing zombies—using an epoch—is used here. Kafka increments the epoch number associated with a `transactional.id` when `initTransaction()` is invoked to initialize a transactional producer. Send, commit, and abort requests from producers with the same `transactional.id` but lower epochs will be rejected with the `FencedProducer` error. The older producer will not be able to write to the output stream and will be forced to `close()`, preventing the zombie from introducing duplicate records. In Apache Kafka 2.5 and later, there is also an option to add consumer group metadata to the transaction metadata. This metadata will also be used for fencing, which will allow producers with different transactional IDs to write to the same partitions while still fencing against zombie instances.

Transactions are a producer feature for the most part—we create a transactional producer, begin the transaction, write records to multiple partitions, produce offsets in order to mark records as already processed, and commit or abort the transaction. We do all this from the producer. However, this isn't quite enough—records written transactionally, even ones that are part of transactions that were eventually aborted, are written to partitions just like any other records. Consumers need to be configured

with the right isolation guarantees, otherwise we won't have the exactly-once guarantees we expected.

We control the consumption of messages that were written transactionally by setting the `isolation.level` configuration. If set to `read_committed`, calling `consumer.poll()` after subscribing to a set of topics will return messages that were either part of a successfully committed transaction or that were written nontransactionally; it will not return messages that were part of an aborted transaction or a transaction that is still open. The default `isolation.level` value, `read_uncommitted`, will return all records, including those that belong to open or aborted transactions. Configuring `read_committed` mode does not guarantee that the application will get all messages that are part of a specific transaction. It is possible to subscribe to only a subset of topics that were part of the transaction and therefore get a subset of the messages. In addition, the application can't know when transactions begin or end, or which messages are part of which transaction.

Figure 8-2 shows which records are visible to a consumer in `read_committed` mode compared to a consumer with the default `read_uncommitted` mode.

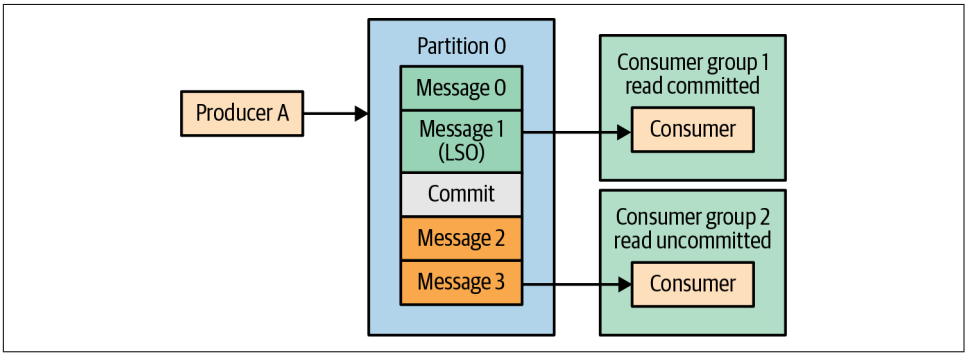

Figure 8-2. Consumers in `read_committed` mode will lag behind consumers with default configuration

To guarantee that messages will be read in order, `read_committed` mode will not return messages that were produced after the point when the first still-open transaction began (known as the Last Stable Offset, or LSO). Those messages will be withheld until that transaction is committed or aborted by the producer, or until they reach `transaction.timeout.ms` (default of 15 minutes) and are aborted by the broker. Holding a transaction open for a long duration will introduce higher end-to-end latency by delaying consumers.

Our simple stream processing job will have exactly-once guarantees on its output even if the input was written nontransactionally. The atomic multipartition produce guarantees that if the output records were committed to the output topic, the offset of

the input records was also committed for that consumer, and as a result the input records will not be processed again.

What Problems Aren't Solved by Transactions?

As explained earlier, transactions were added to Kafka to provide multipartition atomic writes (but not reads) and to fence zombie producers in stream processing applications. As a result, they provide exactly-once guarantees when used within chains of consume-process-produce stream processing tasks. In other contexts, transactions will either straight-out not work or will require additional effort in order to achieve the guarantees we want.

The two main mistakes are assuming that exactly-once guarantees apply on actions other than producing to Kafka, and that consumers always read entire transactions and have information about transaction boundaries.

The following are a few scenarios in which Kafka transactions won't help achieve exactly-once guarantees.

Side effects while stream processing

Let's say that the record processing step in our stream processing app includes sending email to users. Enabling exactly-once semantics in our app will not guarantee that the email will only be sent once. The guarantee only applies to records written to Kafka. Using sequence numbers to deduplicate records or using markers to abort or to cancel a transaction works within Kafka, but it will not un-send an email. The same is true for any action with external effects that is performed within the stream processing app: calling a REST API, writing to a file, etc.

Reading from a Kafka topic and writing to a database

In this case, the application is writing to an external database rather than to Kafka. In this scenario, there is no producer involved—records are written to the database using a database driver (likely JDBC) and offsets are committed to Kafka within the consumer. There is no mechanism that allows writing results to an external database and committing offsets to Kafka within a single transaction. Instead, we could manage offsets in the database (as explained in Chapter 4) and commit both data and offsets to the database in a single transaction—this would rely on the database's transactional guarantees rather than Kafka's.

Microservices often need to update the database *and* publish a message to Kafka within a single atomic transaction, so either both will happen or neither will. As we've just explained in the last two examples, Kafka transactions will not do this.

A common solution to this common problem is known as the *outbox pattern*. The microservice only publishes the message to a Kafka topic (the "outbox"), and a separate message relay service reads the event from Kafka and updates the database. Because, as we've just seen, Kafka won't guarantee an exactly-once update to the database, it is important to make sure the update is idempotent.

Using this pattern guarantees that the message will eventually make it to Kafka, the topic consumers, and the database—or to none of those.

The inverse pattern—where a database table serves as the outbox and a relay service makes sure updates to the table will also arrive to Kafka as messages—is also used. This pattern is preferred when built-in RDBMS constraints, such as uniqueness and foreign keys, are useful. The Debezium project published an in-depth blog post on the outbox pattern with detailed examples.

Reading data from a database, writing to Kafka, and from there writing to another database

It is very tempting to believe that we can build an app that will read data from a database, identify database transactions, write the records to Kafka, and from there write records to another database, still maintaining the original transactions from the source database.

Unfortunately, Kafka transactions don't have the necessary functionality to support these kinds of end-to-end guarantees. In addition to the problem with committing both records and offsets within the same transaction, there is another difficulty: `read_committed` guarantees in Kafka consumers are too weak to preserve database transactions. Yes, a consumer will not see records that were not committed. But it is not guaranteed to have seen all the records that were committed within the transaction because it could be lagging on some topics; it has no information to identify transaction boundaries, so it can't know when a transaction began and ended, and whether it has seen some, none, or all of its records.

Copying data from one Kafka cluster to another

This one is more subtle—it is possible to support exactly-once guarantees when copying data from one Kafka cluster to another. There is a description of how this is done in the Kafka improvement proposal for adding exactly-once capabilities in MirrorMaker 2.0. At the time of this writing, the proposal is still in draft, but the algorithm is clearly described. This proposal includes the guarantee that each record in the source cluster will be copied to the destination cluster exactly once.

However, this does not guarantee that transactions will be atomic. If an app produces several records and offsets transactionally, and then MirrorMaker 2.0 copies them to another Kafka cluster, the transactional properties and guarantees will be lost during the copy process. They are lost for the same reason when copying data from Kafka to a relational database: the consumer reading data from Kafka can't know or guarantee that it is getting all the events in a transaction. For example, it can replicate part of a transaction if it is only subscribed to a subset of the topics.

Publish/subscribe pattern

Here's a slightly more subtle case. We've discussed exactly-once in the context of the consume-process-produce pattern, but the publish/subscribe pattern is a very common use case. Using transactions in a publish/subscribe use case provides some guarantees: consumers configured with `read_committed` mode will not see records that were published as part of a transaction that was aborted. But those guarantees fall short of exactly-once. Consumers may process a message more than once, depending on their own offset commit logic.

The guarantees Kafka provides in this case are similar to those provided by JMS transactions but depend on consumers in `read_committed` mode to guarantee that uncommitted transactions will remain invisible. JMS brokers withhold uncommitted transactions from all consumers.

An important pattern to avoid is publishing a message and then waiting for another application to respond before committing the transaction. The other application will not receive the message until after the transaction was committed, resulting in a deadlock.

How Do I Use Transactions?

Transactions are a broker feature and part of the Kafka protocol, so there are multiple clients that support transactions.

The most common and most recommended way to use transactions is to enable exactly-once guarantees in Kafka Streams. This way, we will not use transactions directly at all, but rather Kafka Streams will use them for us behind the scenes to provide the guarantees we need. Transactions were designed with this use case in mind, so using them via Kafka Streams is the easiest and most likely to work as expected.

To enable exactly-once guarantees for a Kafka Streams application, we simply set the `processing.guarantee` configuration to either `exactly_once` or `exactly_once_beta`. That's it.

`exactly_once_beta` is a slightly different method of handling application instances that crash or hang with in-flight transactions. This was introduced in release 2.5 to Kafka brokers, and in release 2.6 to Kafka Streams. The main benefit of this method is the ability to handle many partitions with a single transactional producer and therefore create more scalable Kafka Streams applications. There is more information about the changes in the Kafka improvement proposal where they were first discussed.

But what if we want exactly-once guarantees without using Kafka Streams? In this case we will use transactional APIs directly. Here's a snippet showing how this will work. There is a full example in the Apache Kafka GitHub, which includes a demo driver and a simple exactly-once processor that runs in separate threads:

```
Properties producerProps = new Properties();
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
producerProps.put(ProducerConfig.CLIENT_ID_CONFIG, "DemoProducer");
producerProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionalId); ❶

producer = new KafkaProducer<>(producerProps);

Properties consumerProps = new Properties();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); ❷
consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed"); ❸

consumer = new KafkaConsumer<>(consumerProps);

producer.initTransactions(); ❹

consumer.subscribe(Collections.singleton(inputTopic)); ❺

while (true) {
  try {
    ConsumerRecords<Integer, String> records =
      consumer.poll(Duration.ofMillis(200));
    if (records.count() > 0) {
      producer.beginTransaction(); ❻
      for (ConsumerRecord<Integer, String> record : records) {
        ProducerRecord<Integer, String> customizedRecord = transform(record); ❼
        producer.send(customizedRecord);
      }
      Map<TopicPartition, OffsetAndMetadata> offsets = consumerOffsets();
      producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata()); ❽
      producer.commitTransaction(); ❾
    }
  } catch (ProducerFencedException|InvalidProducerEpochException e) { ❿
    throw new KafkaException(String.format(
      "The transactional.id %s is used by another process", transactionalId));
```

```
    } catch (KafkaException e) {
      producer.abortTransaction(); ⓫
      resetToLastCommittedPositions(consumer);
    }}
```

❶ Configuring a producer with `transactional.id` makes it a transactional producer capable of producing atomic multipartition writes. The transactional ID must be unique and long-lived. Essentially it defines an instance of the application.

❷ Consumers that are part of the transactions don't commit their own offsets—the producer writes offsets as part of the transaction. So offset commit should be disabled.

❸ In this example, the consumer reads from an input topic. We will assume that the records in the input topic were also written by a transactional producer (just for fun—there is no such requirement for the input). To read transactions cleanly (i.e., ignore in-flight and aborted transactions), we will set the consumer isolation level to `read_committed`. Note that the consumer will still read nontransactional writes, in addition to reading committed transactions.

❹ The first thing a transactional producer must do is initialize. This registers the transactional ID, bumps up the epoch to guarantee that other producers with the same ID will be considered zombies, and aborts older in-flight transactions from the same transactional ID.

❺ Here we are using the `subscribe` consumer API, which means that partitions assigned to this instance of the application can change at any point as a result of rebalance. Prior to release 2.5, which introduced API changes from KIP-447, this was much more challenging. Transactional producers had to be statically assigned a set of partitions, because the transaction fencing mechanism relied on the same transactional ID being used for the same partitions (there was no zombie fencing protection if the transactional ID changed). KIP-447 added new APIs, used in this example, that attach consumer-group information to the transaction, and this information is used for fencing. When using this method, it also makes sense to commit transactions whenever the related partitions are revoked.

❻ We consumed records, and now we want to process them and produce results. This method guarantees that everything that is produced from the time it was called, until the transaction is either committed or aborted, is part of a single atomic transaction.

❼ This is where we process the records—all our business logic goes here.

❽ As we explained earlier in the chapter, it is important to commit the offsets as part of the transaction. This guarantees that if we fail to produce results, we won't commit the offsets for records that were not, in fact, processed. This method commits offsets as part of the transaction. Note that it is important not to commit offsets in any other way—disable offset auto-commit, and don't call any of the consumer commit APIs. Committing offsets by any other method does not provide transactional guarantees.

❾ We produced everything we needed, we committed offsets as part of the transaction, and it is time to commit the transaction and seal the deal. Once this method returns successfully, the entire transaction has made it through, and we can continue to read and process the next batch of events.

❿ If we got this exception, it means we are the zombie. Somehow our application froze or disconnected, and there is a newer instance of the app with our transactional ID running. Most likely the transaction we started has already been aborted and someone else is processing those records. Nothing to do but die gracefully.

⓫ If we got an error while writing a transaction, we can abort the transaction, set the consumer position back, and try again.

Transactional IDs and Fencing

Choosing the transactional ID for producers is important and a bit more challenging than it seems. Assigning the transactional ID incorrectly can lead to either application errors or loss of exactly-once guarantees. The key requirements are that the transactional ID will be consistent for the same instance of the application between restarts and is different for different instances of the application, otherwise the brokers will not be able to fence off zombie instances.

Until release 2.5, the only way to guarantee fencing was to statically map the transactional ID to partitions. This guaranteed that each partition will always be consumed with the same transactional ID. If a producer with transactional ID A processed messages from topic T and lost connectivity, and the new producer that replaces it has transactional ID B, and later producer A comes back as a zombie, zombie A will not be fenced because the ID doesn't match that of the new producer B. We want producer A to always be replaced by producer A, and the new producer A will have a higher epoch number and zombie A will be properly fenced away. In those releases, the previous example would be incorrect—transactional IDs are assigned randomly to threads without making sure the same transactional ID is always used to write to the same partition.

In Apache Kafka 2.5, KIP-447 introduced a second method of fencing based on consumer group metadata for fencing in addition to transactional IDs. We use the producer offset commit method and pass as an argument the consumer group metadata rather than just the consumer group ID.

Let's say that we have topic T1 with two partitions, t-0 and t-1. Each is consumed by a separate consumer in the same group; each consumer passes records to a matching transactional producer—one with transactional ID A and the other with transactional ID B; and they are writing output to topic T2 partitions 0 and 1, respectively. Figure 8-3 illustrates this scenario.

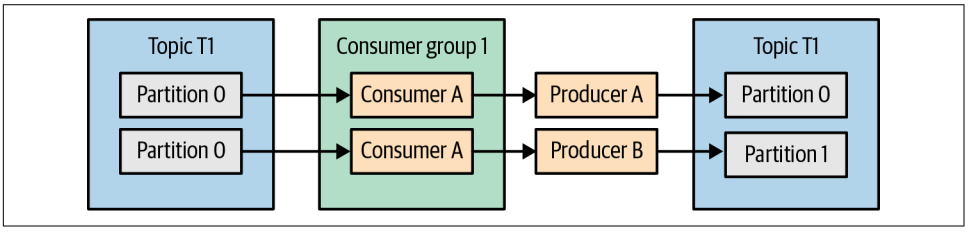

Figure 8-3. Transactional record processor

As illustrated in Figure 8-4, if the application instance with consumer A and producer A becomes a zombie, consumer B will start processing records from both partitions. If we want to guarantee that no zombies write to partition 0, consumer B can't just start reading from partition 0 and writing to partition 0 with transactional ID B. Instead the application will need to instantiate a new producer, with transactional ID A, to safely write to partition 0 and fence the old transactional ID A. This is wasteful. Instead, we include the consumer group information in the transactions. Transactions from producer B will show that they are from a newer generation of the consumer group, and therefore they will go through, while transactions from the now-zombie producer A will show an old generation of the consumer group and will be fenced.

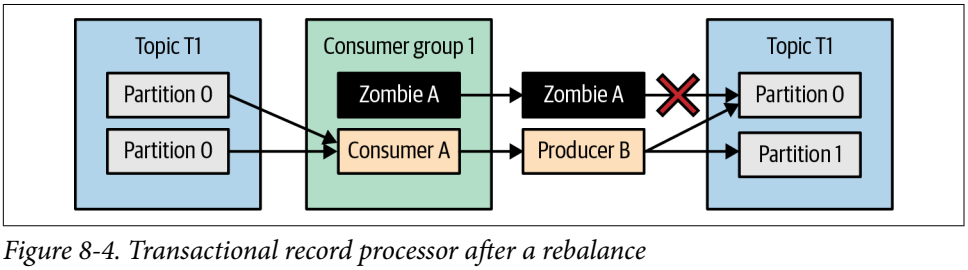

Figure 8-4. Transactional record processor after a rebalance

How Transactions Work

We can use transactions by calling the APIs without understanding how they work. But having some mental model of what is going on under the hood will help us troubleshoot applications that do not behave as expected.

The basic algorithm for transactions in Kafka was inspired by Chandy-Lamport snapshots, in which "marker" control messages are sent into communication channels, and consistent state is determined based on the arrival of the marker. Kafka transactions use marker messages to indicate that transactions are committed or aborted across multiple partitions—when the producer decides to commit a transaction, it sends a "commit" message to the transaction coordinator, which then writes commit markers to all partitions involved in a transaction. But what happens if the producer crashes after only writing commit messages to a subset of the partitions? Kafka transactions solve this by using two-phase commit and a transaction log. At a high level, the algorithm will:

1. Log the existence of an ongoing transaction, including the partitions involved
2. Log the intent to commit or abort—once this is logged, we are doomed to commit or abort eventually
3. Write all the transaction markers to all the partitions
4. Log the completion of the transaction

To implement this basic algorithm, Kafka needs a transaction log. We use an internal topic called `__transaction_state`.

Let's see how this algorithm works in practice by going through the inner workings of the transactional API calls we've used in the preceding code snippet.

Before we begin the first transaction, producers need to register as transactional by calling `initTransaction()`. This request is sent to a broker that will be the *transaction coordinator* for this transactional producer. Each broker is the transactional coordinator for a subset of the producers, just like each broker is the consumer group coordinator for a subset of the consumer groups. The transaction coordinator for each transactional ID is the leader of the partition of the transaction log the transactional ID is mapped to.

The `initTransaction()` API registers a new transactional ID with the coordinator, or increments the epoch of an existing transactional ID in order to fence off previous producers that may have become zombies. When the epoch is incremented, pending transactions will be aborted.

The next step for the producer is to call `beginTransaction()`. This API call isn't part of the protocol—it simply tells the producer that there is now a transaction in progress. The transaction coordinator on the broker side is still unaware that the transaction began. However, once the producer starts sending records, each time the producer detects that it is sending records to a new partition, it will also send `AddPartitionsToTxnRequest` to the broker informing it that there is a transaction in progress for this producer, and that additional partitions are part of the transaction. This information will be recorded in the transaction log.

When we are done producing results and are ready to commit, we start by committing offsets for the records we've processed in this transaction. Committing offsets can be done at any time but must be done before the transaction is committed. Calling `sendOffsetsToTransaction()` will send a request to the transaction coordinator that includes the offsets and also the consumer group ID. The transaction coordinator will use the consumer group ID to find the group coordinator and commit the offsets as a consumer group normally would.

Now it is time to commit—or abort. Calling `commitTransaction()` or `abortTransaction()` will send an `EndTransactionRequest` to the transaction coordinator. The transaction coordinator will log the commit or abort intention to the transaction log. Once this step is successful, it is the transaction coordinator's responsibility to complete the commit (or abort) process. It writes a commit marker to all the partitions involved in the transaction, then writes to the transaction log that the commit completed successfully. Note that if the transaction coordinator shuts down or crashes after logging the intention to commit and before completing the process, a new transaction coordinator will be elected, pick up the intent to commit from the transaction log, and complete the process.

If a transaction is not committed or aborted within `transaction.timeout.ms`, the transaction coordinator will abort it automatically.

Each broker that receives records from transactional or idempotent producers will store the producer/transactional IDs in memory, together with related state for each of the last five batches sent by the producer: sequence numbers, offsets, and such. This state is stored for `transactional.id.expiration.ms` milliseconds after the producer stopped being active (seven days by default). This allows the producer to resume activity without running into `UNKNOWN_PRODUCER_ID` errors. It is possible to cause something similar to a memory leak in the broker by creating new idempotent producers or new transactional IDs at a very high rate but never reusing them. Three new idempotent producers per second, accumulated over the course of a week, will result in 1.8 million producer state entries with a total of 9 million batch metadata stored, using around 5 GB RAM. This can cause out-of-memory or severe garbage collection issues on the broker. We recommend architecting the application to initialize a few long-lived producers when the application starts up, and then reuse them for the lifetime of the application. If this isn't possible (Function as a Service makes this difficult), we recommend lowering `transactional.id.expiration.ms` so the IDs will expire faster, and therefore old state that will never be reused won't take up a significant part of the broker memory.

Performance of Transactions

Transactions add moderate overhead to the producer. The request to register transactional ID occurs once in the producer lifecycle. Additional calls to register partitions as part of a transaction happen at most one per partition for each transaction, then each transaction sends a commit request, which causes an extra commit marker to be written on each partition. The transactional initialization and transaction commit requests are synchronous, so no data will be sent until they complete successfully, fail, or time out, which further increases the overhead.

Note that the overhead of transactions on the producer is independent of the number of messages in a transaction. So a larger number of messages per transaction will both reduce the relative overhead and reduce the number of synchronous stops, resulting in higher throughput overall.

On the consumer side, there is some overhead involved in reading commit markers. The key impact that transactions have on consumer performance is introduced by the fact that consumers in `read_committed` mode will not return records that are part of an open transaction. Long intervals between transaction commits mean that the consumer will need to wait longer before returning messages, and as a result, end-to-end latency will increase.

Note, however, that the consumer does not need to buffer messages that belong to open transactions. The broker will not return those in response to fetch requests from the consumer. Since there is no extra work for the consumer when reading transactions, there is no decrease in throughput either.

Summary

Exactly-once semantics in Kafka is the opposite of chess: it is challenging to understand but easy to use.

This chapter covered the two key mechanisms that provide exactly-once guarantees in Kafka: idempotent producer, which avoids duplicates that are caused by the retry mechanism, and transactions, which form the basis of exactly-once semantics in Kafka Streams.

Both can be enabled in a single configuration and allow us to use Kafka for applications that require fewer duplicates and stronger correctness guarantees.

We discussed in depth specific scenarios and use cases to show the expected behavior, and even looked at some of the implementation details. Those details are important when troubleshooting applications or when using transactional APIs directly.

By understanding what Kafka's exactly-once semantics guarantee in which use case, we can design applications that will use exactly-once when necessary. Application behavior should not be surprising, and the information in this chapter will help us avoid surprises.

CHAPTER 9
Building Data Pipelines

When people discuss building data pipelines using Apache Kafka, they are usually referring to a couple of use cases. The first is building a data pipeline where Apache Kafka is one of the two end points—for example, getting data from Kafka to S3 or getting data from MongoDB into Kafka. The second use case involves building a pipeline between two different systems but using Kafka as an intermediary. An example of this is getting data from Twitter to Elasticsearch by sending the data first from Twitter to Kafka and then from Kafka to Elasticsearch.

When we added Kafka Connect to Apache Kafka in version 0.9, it was after we saw Kafka used in both use cases at LinkedIn and other large organizations. We noticed that there were specific challenges in integrating Kafka into data pipelines that every organization had to solve, and decided to add APIs to Kafka that solve some of those challenges rather than force every organization to figure them out from scratch.

The main value Kafka provides to data pipelines is its ability to serve as a very large, reliable buffer between various stages in the pipeline. This effectively decouples producers and consumers of data within the pipeline and allows use of the same data from the source in multiple target applications and systems, all with different timeliness and availability requirements. This decoupling, combined with reliability, security, and efficiency, makes Kafka a good fit for most data pipelines.

Putting Data Integration in Context

Some organizations think of Kafka as an *end point* of a pipeline. They look at questions such as "How do I get data from Kafka to Elastic?" This is a valid question to ask—especially if there is data you need in Elastic and it is currently in Kafka—and we will look at ways to do exactly this. But we are going to start the discussion by looking at the use of Kafka within a larger context that includes at least two (and possibly many more) end points that are not Kafka itself. We encourage anyone faced with a data-integration problem to consider the bigger picture and not focus only on the immediate end points. Focusing on short-term integrations is how you end up with a complex and expensive-to-maintain data integration mess.

In this chapter, we'll discuss some of the common issues that you need to take into account when building data pipelines. Those challenges are not specific to Kafka but are general data integration problems. Nonetheless, we will show why Kafka is a good fit for data integration use cases and how it addresses many of those challenges. We will discuss how the Kafka Connect API is different from the normal producer and consumer clients, and when each client type should be used. Then we'll jump into some details of Kafka Connect. While a full discussion of Kafka Connect is outside the scope of this chapter, we will show examples of basic usage to get you started and give you pointers on where to learn more. Finally, we'll discuss other data integration systems and how they integrate with Kafka.

Considerations When Building Data Pipelines

While we won't get into all the details on building data pipelines here, we would like to highlight some of the most important things to take into account when designing software architectures with the intent of integrating multiple systems.

Timeliness

Some systems expect their data to arrive in large bulks once a day; others expect the data to arrive a few milliseconds after it is generated. Most data pipelines fit somewhere in between these two extremes. Good data integration systems can support different timeliness requirements for different pipelines and also make the migration between different timetables easier as business requirements change. Kafka, being a streaming data platform with scalable and reliable storage, can be used to support anything from near-real-time pipelines to daily batches. Producers can write to Kafka as frequently and infrequently as needed, and consumers can also read and deliver the latest events as they arrive. Or consumers can work in batches: run every hour, connect to Kafka, and read the events that accumulated during the previous hour.

A useful way to look at Kafka in this context is that it acts as a giant buffer that decouples the time-sensitivity requirements between producers and consumers. Producers can write events in real time, while consumers process batches of events, or vice versa. This also makes it trivial to apply back pressure—Kafka itself applies back pressure on producers (by delaying acks when needed) since consumption rate is driven entirely by the consumers.

Reliability

We want to avoid single points of failure and allow for fast and automatic recovery from all sorts of failure events. Data pipelines are often the way data arrives to business-critical systems; failure for more than a few seconds can be hugely disruptive, especially when the timeliness requirement is closer to the few milliseconds end of the spectrum. Another important consideration for reliability is delivery guarantees—some systems can afford to lose data, but most of the time there is a requirement for *at-least-once* delivery, which means every event from the source system will reach its destination, but sometimes retries will cause duplicates. Often, there is even a requirement for *exactly-once* delivery—every event from the source system will reach the destination with no possibility for loss or duplication.

We discussed Kafka's availability and reliability guarantees in depth in Chapter 7. As we discussed, Kafka can provide at-least-once on its own, and exactly-once when combined with an external data store that has a transactional model or unique keys. Since many of the end points are data stores that provide the right semantics for exactly-once delivery, a Kafka-based pipeline can often be implemented as exactly-once. It is worth highlighting that Kafka's Connect API makes it easier for connectors to build an end-to-end exactly-once pipeline by providing an API for integrating with the external systems when handling offsets. Indeed, many of the available open source connectors support exactly-once delivery.

High and Varying Throughput

The data pipelines we are building should be able to scale to very high throughputs, as is often required in modern data systems. Even more importantly, they should be able to adapt if throughput suddenly increases.

With Kafka acting as a buffer between producers and consumers, we no longer need to couple consumer throughput to the producer throughput. We no longer need to implement a complex back-pressure mechanism because if producer throughput exceeds that of the consumer, data will accumulate in Kafka until the consumer can catch up. Kafka's ability to scale by adding consumers or producers independently allows us to scale either side of the pipeline dynamically and independently to match the changing requirements.

Kafka is a high-throughput distributed system—capable of processing hundreds of megabytes per second on even modest clusters—so there is no concern that our pipeline will not scale as demand grows. In addition, the Kafka Connect API focuses on parallelizing the work and can do this on a single node as well as by scaling out, depending on system requirements. We'll describe in the following sections how the platform allows data sources and sinks to split the work among multiple threads of execution and use the available CPU resources even when running on a single machine.

Kafka also supports several types of compression, allowing users and admins to control the use of network and storage resources as the throughput requirements increase.

Data Formats

One of the most important considerations in a data pipeline is reconciling different data formats and data types. The data types supported vary among different databases and other storage systems. You may be loading XMLs and relational data into Kafka, using Avro within Kafka, and then need to convert data to JSON when writing it to Elasticsearch, to Parquet when writing to HDFS, and to CSV when writing to S3.

Kafka itself and the Connect API are completely agnostic when it comes to data formats. As we've seen in previous chapters, producers and consumers can use any serializer to represent data in any format that works for you. Kafka Connect has its own in-memory objects that include data types and schemas, but as we'll soon discuss, it allows for pluggable converters to allow storing these records in any format. This means that no matter which data format you use for Kafka, it does not restrict your choice of connectors.

Many sources and sinks have a schema; we can read the schema from the source with the data, store it, and use it to validate compatibility or even update the schema in the sink database. A classic example is a data pipeline from MySQL to Snowflake. If someone added a column in MySQL, a great pipeline will make sure the column gets added to Snowflake too as we are loading new data into it.

In addition, when writing data from Kafka to external systems, sink connectors are responsible for the format in which the data is written to the external system. Some connectors choose to make this format pluggable. For example, the S3 connector allows a choice between Avro and Parquet formats.

It is not enough to support different types of data. A generic data integration framework should also handle differences in behavior between various sources and sinks. For example, Syslog is a source that pushes data, while relational databases require the framework to pull data out. HDFS is append-only and we can only write data to it, while most systems allow us to both append data and update existing records.

Transformations

Transformations are more controversial than other requirements. There are generally two approaches to building data pipelines: ETL and ELT. ETL, which stands for *Extract-Transform-Load*, means that the data pipeline is responsible for making modifications to the data as it passes through. It has the perceived benefit of saving time and storage because you don't need to store the data, modify it, and store it again. Depending on the transformations, this benefit is sometimes real, but sometimes it shifts the burden of computation and storage to the data pipeline itself, which may or may not be desirable. The main drawback of this approach is that the transformations that happen to the data in the pipeline may tie the hands of those who wish to process the data further down the pipe. If the person who built the pipeline between MongoDB and MySQL decided to filter certain events or remove fields from records, all the users and applications who access the data in MySQL will only have access to partial data. If they require access to the missing fields, the pipeline needs to be rebuilt, and historical data will require reprocessing (assuming it is available).

ELT stands for *Extract-Load-Transform* and means that the data pipeline does only minimal transformation (mostly around data type conversion), with the goal of making sure the data that arrives at the target is as similar as possible to the source data. In these systems, the target system collects "raw data" and all required processing is done at the target system. The benefit here is that the system provides maximum flexibility to users of the target system, since they have access to all the data. These systems also tend to be easier to troubleshoot since all data processing is limited to one system rather than split between the pipeline and additional applications. The drawback is that the transformations take CPU and storage resources at the target system. In some cases, these systems are expensive and there is strong motivation to move computation off those systems when possible.

Kafka Connect includes the Single Message Transformation feature, which transforms records while they are being copied from a source to Kafka, or from Kafka to a target. This includes routing messages to different topics, filtering messages, changing data types, redacting specific fields, and more. More complex transformations that involve joins and aggregations are typically done using Kafka Streams, and we will explore those in detail in a separate chapter.

When building an ETL system with Kafka, keep in mind that Kafka allows you to build one-to-many pipelines, where the source data is written to Kafka once and then consumed by multiple applications and written to multiple target systems. Some preprocessing and cleanup is expected, such as standardizing timestamps and data types, adding lineage, and perhaps removing personal information—transformations that will benefit all consumers of the data. But don't prematurely clean and optimize the data on ingest because it might be needed less refined elsewhere.

Security

Security should always be a concern. In terms of data pipelines, the main security concerns are usually:

- Who has access to the data that is ingested into Kafka?
- Can we make sure the data going through the pipe is encrypted? This is mainly a concern for data pipelines that cross datacenter boundaries.
- Who is allowed to make modifications to the pipelines?
- If the data pipeline needs to read or write from access-controlled locations, can it authenticate properly?
- Is our PII (Personally Identifiable Information) handling compliant with laws and regulations regarding its storage, access and use?

Kafka allows encrypting data on the wire, as it is piped from sources to Kafka and from Kafka to sinks. It also supports authentication (via SASL) and authorization—so you can be sure that if a topic contains sensitive information, it can't be piped into less secured systems by someone unauthorized. Kafka also provides an audit log to track access—unauthorized and authorized. With some extra coding, it is also possible to track where the events in each topic came from and who modified them, so you can provide the entire lineage for each record.

Kafka security is discussed in detail in Chapter 11. However, Kafka Connect and its connectors need to be able to connect to, and authenticate with, external data systems, and configuration of connectors will include credentials for authenticating with external data systems.

These days it is not recommended to store credentials in configuration files, since this means that the configuration files have to be handled with extra care and have restricted access. A common solution is to use an external secret management system such as HashiCorp Vault. Kafka Connect includes support for external secret configuration. Apache Kafka only includes the framework that allows introduction of pluggable external config providers, an example provider that reads configuration from a

file, and there are community-developed external config providers that integrate with Vault, AWS, and Azure.

Failure Handling

Assuming that all data will be perfect all the time is dangerous. It is important to plan for failure handling in advance. Can we prevent faulty records from ever making it into the pipeline? Can we recover from records that cannot be parsed? Can bad records get fixed (perhaps by a human) and reprocessed? What if the bad event looks exactly like a normal event and you only discover the problem a few days later?

Because Kafka can be configured to store all events for long periods of time, it is possible to go back in time and recover from errors when needed. This also allows replaying the events stored in Kafka to the target system if they were lost.

Coupling and Agility

A desirable characteristic of data pipeline implementation is to decouple the data sources and data targets. There are multiple ways accidental coupling can happen:

Ad hoc pipelines
: Some companies end up building a custom pipeline for each pair of applications they want to connect. For example, they use Logstash to dump logs to Elasticsearch, Flume to dump logs to HDFS, Oracle GoldenGate to get data from Oracle to HDFS, Informatica to get data from MySQL and XML to Oracle, and so on. This tightly couples the data pipeline to the specific end points and creates a mess of integration points that requires significant effort to deploy, maintain, and monitor. It also means that every new system the company adopts will require building additional pipelines, increasing the cost of adopting new technology, and inhibiting innovation.

Loss of metadata
: If the data pipeline doesn't preserve schema metadata and does not allow for schema evolution, you end up tightly coupling the software producing the data at the source and the software that uses it at the destination. Without schema information, both software products need to include information on how to parse the data and interpret it. If data flows from Oracle to HDFS and a DBA added a new field in Oracle without preserving schema information and allowing schema evolution, either every app that reads data from HDFS will break or all the developers will need to upgrade their applications at the same time. Neither option is agile. With support for schema evolution in the pipeline, each team can modify their applications at their own pace without worrying that things will break down the line.

Extreme processing

As we mentioned when discussing data transformations, some processing of data is inherent to data pipelines. After all, we are moving data between different systems where different data formats make sense and different use cases are supported. However, too much processing ties all the downstream systems to decisions made when building the pipelines about which fields to preserve, how to aggregate data, etc. This often leads to constant changes to the pipeline as requirements of downstream applications change, which isn't agile, efficient, or safe. The more agile way is to preserve as much of the raw data as possible and allow downstream apps, including Kafka Streams apps, to make their own decisions regarding data processing and aggregation.

When to Use Kafka Connect Versus Producer and Consumer

When writing to Kafka or reading from Kafka, you have the choice between using traditional producer and consumer clients, as described in Chapters 3 and 4, or using the Kafka Connect API and the connectors, as we'll describe in the following sections. Before we start diving into the details of Kafka Connect, you may already be wondering, "When do I use which?"

As we've seen, Kafka clients are clients embedded in your own application. It allows your application to write data to Kafka or to read data from Kafka. Use Kafka clients when you can modify the code of the application that you want to connect an application to and when you want to either push data into Kafka or pull data from Kafka.

You will use Connect to connect Kafka to datastores that you did not write and whose code or APIs you cannot or will not modify. Connect will be used to pull data from the external datastore into Kafka or push data from Kafka to an external store. To use Kafka Connect, you need a connector for the datastore to which you want to connect, and nowadays these connectors are plentiful. This means that in practice, users of Kafka Connect only need to write configuration files.

If you need to connect Kafka to a datastore and a connector does not exist yet, you can choose between writing an app using the Kafka clients or the Connect API. Connect is recommended because it provides out-of-the-box features like configuration management, offset storage, parallelization, error handling, support for different data types, and standard management REST APIs. Writing a small app that connects Kafka to a datastore sounds simple, but there are many little details you will need to handle concerning data types and configuration that make the task nontrivial. What's more, you will need to maintain this pipeline app and document it, and your teammates will need to learn how to use it. Kafka Connect is a standard part of the

Kafka ecosystem, and it handles most of this for you, allowing you to focus on transporting data to and from the external stores.

Kafka Connect

Kafka Connect is a part of Apache Kafka and provides a scalable and reliable way to copy data between Kafka and other datastores. It provides APIs and a runtime to develop and run *connector plug-ins*—libraries that Kafka Connect executes and that are responsible for moving the data. Kafka Connect runs as a cluster of *worker processes*. You install the connector plug-ins on the workers and then use a REST API to configure and manage *connectors*, which run with a specific configuration. *Connectors* start additional *tasks* to move large amounts of data in parallel and use the available resources on the worker nodes more efficiently. Source connector tasks just need to read data from the source system and provide Connect data objects to the worker processes. Sink connector tasks get connector data objects from the workers and are responsible for writing them to the target data system. Kafka Connect uses *convertors* to support storing those data objects in Kafka in different formats—JSON format support is part of Apache Kafka, and the Confluent Schema Registry provides Avro, Protobuf, and JSON Schema converters. This allows users to choose the format in which data is stored in Kafka independent of the connectors they use, as well as how the schema of the data is handled (if at all).

This chapter cannot possibly get into all the details of Kafka Connect and its many connectors. This could fill an entire book on its own. We will, however, give an overview of Kafka Connect and how to use it, and point to additional resources for reference.

Running Kafka Connect

Kafka Connect ships with Apache Kafka, so there is no need to install it separately. For production use, especially if you are planning to use Connect to move large amounts of data or run many connectors, you should run Connect on separate servers from your Kafka brokers. In this case, install Apache Kafka on all the machines, and simply start the brokers on some servers and start Connect on other servers.

Starting a Connect worker is very similar to starting a broker—you call the start script with a properties file:

```
bin/connect-distributed.sh config/connect-distributed.properties
```

There are a few key configurations for Connect workers:

`bootstrap.servers`
: A list of Kafka brokers that Connect will work with. `Connectors` will pipe their data either to or from those brokers. You don't need to specify every broker in the cluster, but it's recommended to specify at least three.

`group.id`
: All workers with the same group ID are part of the same Connect cluster. A connector started on the cluster will run on any worker, and so will its tasks.

`plugin.path`
: Kafka Connect uses a pluggable architecture where connectors, converters, transformations, and secret providers can be downloaded and added to the platform. In order to do this, Kafka Connect has to be able to find and load those plug-ins.

  We can configure one or more directories as locations where connectors and their dependencies can be found. For example, we can configure `plugin.path=/opt/connectors,/home/gwenshap/connectors`. Inside one of these directories, we will typically create a subdirectory for each connector, so in the previous example, we'll create `/opt/connectors/jdbc` and `/opt/connectors/elastic`. Inside each subdirectory, we'll place the connector jar itself and all its dependencies. If the connector ships as an `uberJar` and has no dependencies, it can be placed directly in `plugin.path` and doesn't require a subdirectory. But note that placing dependencies in the top-level path will not work.

  An alternative is to add the connectors and all their dependencies to the Kafka Connect classpath, but this is not recommended and can introduce errors if you use a connector that brings a dependency that conflicts with one of Kafka's dependencies. The recommended approach is to use `plugin.path` configuration.

`key.converter` *and* `value.converter`
: Connect can handle multiple data formats stored in Kafka. The two configurations set the converter for the key and value part of the message that will be stored in Kafka. The default is JSON format using the `JSONConverter` included in Apache Kafka. These configurations can also be set to `AvroConverter`, `ProtobufConverter`, or `JscoSchemaConverter`, which are part of the Confluent Schema Registry.

  Some converters include converter-specific configuration parameters. You need to prefix these parameters with `key.converter.` or `value.converter.`, depending on whether you want to apply them to the key or value converter. For example, JSON messages can include a schema or be schema-less. To support either, you can set `key.converter.schemas.enable=true` or `false`, respectively. The same configuration can be used for the value converter by setting `value.converter.schemas.enable` to `true` or `false`. Avro messages also

contain a schema, but you need to configure the location of the Schema Registry using `key.converter.schema.registry.url` and `value.converter.schema.registry.url`.

`rest.host.name` *and* `rest.port`
: Connectors are typically configured and monitored through the REST API of Kafka Connect. You can configure the specific port for the REST API.

Once the workers are up and you have a cluster, make sure it is up and running by checking the REST API:

```
$ curl http://localhost:8083/
{"version":"3.0.0-
SNAPSHOT","commit":"fae0784ce32a448a","kafka_cluster_id":"pfkYIGZQSXm8Ryl-
vACQHdg"}%
```

Accessing the base REST URI should return the current version you are running. We are running a snapshot of Kafka 3.0.0 (prerelease). We can also check which connector plug-ins are available:

```
$ curl http://localhost:8083/connector-plugins

[
  {
    "class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type": "sink",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type": "source",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorHeartbeatConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "type": "source",
    "version": "1"
  }
]
```

We are running plain Apache Kafka, so the only available connector plug-ins are the file source, file sink, and the connectors that are part of MirrorMaker 2.0.

Let's see how to configure and use these example connectors, and then we'll dive into more advanced examples that require setting up external data systems to connect to.

Standalone Mode

Take note that Kafka Connect also has a standalone mode. It is similar to distributed mode—you just run `bin/connect-standalone.sh` instead of `bin/connect-distributed.sh`. You can also pass in a connector configuration file on the command line instead of through the REST API. In this mode, all the connectors and tasks run on the one standalone worker. It is used in cases where connectors and tasks need to run on a specific machine (e.g., the `syslog` connector listens on a port, so you need to know which machines it is running on).

Connector Example: File Source and File Sink

This example will use the file connectors and JSON converter that are part of Apache Kafka. To follow along, make sure you have ZooKeeper and Kafka up and running.

To start, let's run a distributed Connect worker. In a real production environment, you'll want at least two or three of these running to provide high availability. In this example, we'll only start one:

```
bin/connect-distributed.sh config/connect-distributed.properties &
```

Now it's time to start a file source. As an example, we will configure it to read the Kafka configuration file—basically piping Kafka's configuration into a Kafka topic:

```
echo '{"name":"load-kafka-config", "config":{"connector.class":
"FileStreamSource","file":"config/server.properties","topic":
"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:8083/connectors
-H "Content-Type: application/json"

{
  "name": "load-kafka-config",
  "config": {
    "connector.class": "FileStreamSource",
    "file": "config/server.properties",
    "topic": "kafka-config-topic",
    "name": "load-kafka-config"
  },
  "tasks": [
    {
      "connector": "load-kafka-config",
      "task": 0
    }
```

```
  ],
  "type": "source"
}
```

To create a connector, we wrote a JSON that includes a connector name, `load-kafka-config`, and a connector configuration map, which includes the connector class, the file we want to load, and the topic we want to load the file into.

Let's use the Kafka Console consumer to check that we have loaded the configuration into a topic:

```
gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092
--topic kafka-config-topic --from-beginning
```

If all went well, you should see something along the lines of:

```
{"schema":{"type":"string","optional":false},"payload":"# Licensed to the
Apache Software Foundation (ASF) under one or more"}

<more stuff here>

{"schema":{"type":"string","optional":false},"pay-
load":"############################ Server Basics
############################"}
{"schema":{"type":"string","optional":false},"payload":""}
{"schema":{"type":"string","optional":false},"payload":"# The id of the broker.
This must be set to a unique integer for each broker."}
{"schema":{"type":"string","optional":false},"payload":"broker.id=0"}
{"schema":{"type":"string","optional":false},"payload":""}

<more stuff here>
```

This is literally the contents of the *config/server.properties* file, as it was converted to JSON line by line and placed in `kafka-config-topic` by our connector. Note that by default, the JSON converter places a schema in each record. In this specific case, the schema is very simple—there is only a single column, named `payload` of type `string`, and it contains a single line from the file for each record.

Now let's use the file sink converter to dump the contents of that topic into a file. The resulting file should be completely identical to the original *server.properties* file, as the JSON converter will convert the JSON records back into simple text lines:

```
echo '{"name":"dump-kafka-config", "config":
{"connector.class":"FileStreamSink","file":"copy-of-server-
properties","topics":"kafka-config-topic"}}' | curl -X POST -d @- http://local-
host:8083/connectors --header "content-Type:application/json"

{"name":"dump-kafka-config","config":
{"connector.class":"FileStreamSink","file":"copy-of-server-
properties","topics":"kafka-config-topic","name":"dump-kafka-config"},"tasks":
[]}
```

Note the changes from the source configuration: the class we are using is now `FileStreamSink` rather than `FileStreamSource`. We still have a file property, but now it refers to the destination file rather than the source of the records, and instead of specifying a *topic*, you specify *topics*. Note the plurality—you can write multiple topics into one file with the sink, while the source only allows writing into one topic.

If all went well, you should have a file named *copy-of-server-properties*, which is completely identical to the *config/server.properties* we used to populate `kafka-config-topic`.

To delete a connector, you can run:

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```

This example uses FileStream connectors because they are simple and built into Kafka, allowing you to create your first pipeline without installing anything except Kafka. These should not be used for actual production pipelines, as they have many limitations and no reliability guarantees. There are several alternatives you can use if you want to ingest data from files: FilePulse Connector, FileSystem Connector, or SpoolDir.

Connector Example: MySQL to Elasticsearch

Now that we have a simple example working, let's do something more useful. Let's take a MySQL table, stream it to a Kafka topic, and from there load it to Elasticsearch and index its content.

We are running tests on a MacBook. To install MySQL and Elasticsearch, simply run:

```
brew install mysql
brew install elasticsearch
```

The next step is to make sure you have the connectors. There are a few options:

1. Download and install using Confluent Hub client.
2. Download from the Confluent Hub website (or from any other website where the connector you are interested in is hosted).
3. Build from source code. To do this, you'll need to:
   a. Clone the connector source:

      ```
      git clone https://github.com/confluentinc/kafka-connect-elasticsearch
      ```

   b. Run `mvn install -DskipTests` to build the project.
   c. Repeat with the JDBC connector.

Now we need to load these connectors. Create a directory, such as */opt/connectors* and update *config/connect-distributed.properties* to include `plugin.path=/opt/connectors`.

Then take the jars that were created under the `target` directory where you built each connector and copy each one, plus their dependencies, to the appropriate subdirectories of `plugin.path`:

```
gwen$ mkdir /opt/connectors/jdbc
gwen$ mkdir /opt/connectors/elastic
gwen$ cp .../kafka-connect-jdbc/target/kafka-connect-jdbc-10.3.x-
SNAPSHOT.jar /opt/connectors/jdbc
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-
elasticsearch-11.1.0-SNAPSHOT.jar /opt/connectors/elastic
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-
elasticsearch-11.1.0-SNAPSHOT-package/share/java/kafka-connect-
elasticsearch/* /opt/connectors/elastic
```

In addition, since we need to connect not just to any database but specifically to MySQL, you'll need to download and install a MySQL JDBC driver. The driver doesn't ship with the connector for license reasons. You can download the driver from the MySQL website and then place the jar in */opt/connectors/jdbc*.

Restart the Kafka Connect workers and check that the new connector plug-ins are listed:

```
gwen$  bin/connect-distributed.sh config/connect-distributed.properties &

gwen$  curl http://localhost:8083/connector-plugins
[
  {
    "class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "type": "sink",
    "version": "11.1.0-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "type": "sink",
    "version": "10.3.x-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "type": "source",
    "version": "10.3.x-SNAPSHOT"
  }
```

We can see that we now have additional connector plug-ins available in our `Connect` cluster.

The next step is to create a table in MySQL that we can stream into Kafka using our JDBC connector:

```
gwen$ mysql.server restart
gwen$  mysql --user=root

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)
```

As you can see, we created a database and a table, and inserted a few rows as an example.

The next step is to configure our JDBC source connector. We can find out which configuration options are available by looking at the documentation, but we can also use the REST API to find the available configuration options:

```
gwen$ curl -X PUT -d '{"connector.class":"JdbcSource"}' localhost:8083/
connector-plugins/JdbcSourceConnector/config/validate/ --header "content-
Type:application/json"

{
    "configs": [
        {
            "definition": {
                "default_value": "",
                "dependents": [],
                "display_name": "Timestamp Column Name",
                "documentation": "The name of the timestamp column to use
                to detect new or modified rows. This column may not be
                nullable.",
                "group": "Mode",
                "importance": "MEDIUM",
                "name": "timestamp.column.name",
                "order": 3,
                "required": false,
                "type": "STRING",
                "width": "MEDIUM"
            },
            <more stuff>
```

We asked the REST API to validate configuration for a connector and sent it a configuration with just the class name (this is the bare minimum configuration necessary). As a response, we got the JSON definition of all available configurations.

With this information in mind, it's time to create and configure our JDBC connector:

```
echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSource-
Connector","connection.url":"jdbc:mysql://127.0.0.1:3306/test?
user=root","mode":"timestamp","table.whitelist":"login","vali-
date.non.null":false,"timestamp.column.name":"login_time","topic.pre-
fix":"mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header
"content-Type:application/json"


{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector"
  },
  "tasks": []
}
```

Let's make sure it worked by reading data from the `mysql.login` topic:

```
gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning
```

If you get errors saying the topic doesn't exist or you see no data, check the Connect worker logs for errors such as:

```
[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-login-
connector (org.apache.kafka.connect.runtime.WorkerConnector:108)
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: Access
denied for user 'root;'@'localhost' (using password: NO)
        at io.confluent.connect.jdbc.JdbcSourceConnector.start(JdbcSourceConnec-
tor.java:78)
```

Other issues can involve the existence of the driver in the classpath or permissions to read the table.

Once the connector is running, if you insert additional rows in the `login` table, you should immediately see them reflected in the `mysql.login` topic.

Change Data Capture and Debezium Project

The JDBC connector that we are using uses JDBC and SQL to scan database tables for new records. It detects new records by using timestamp fields or an incrementing primary key. This is a relatively inefficient and at times inaccurate process. All relational databases have a transaction log (also called redo log, binlog, or write-ahead log) as part of their implementation, and many allow external systems to read data directly from their transaction log—a far more accurate and efficient process known as `change data capture`. Most modern ETL systems depend on change data capture as a data source. The Debezium Project provides a collection of high-quality, open source, change capture connectors for a variety of databases. If you are planning on streaming data from a relational database to Kafka, we highly recommend using a Debezium change capture connector if one exists for your database. In addition, the Debezium documentation is one of the best we've seen—in addition to documenting the connectors themselves, it covers useful design patterns and use cases related to change data capture, especially in the context of microservices.

Getting MySQL data to Kafka is useful in itself, but let's make things more fun by writing the data to Elasticsearch.

First, we start Elasticsearch and verify it is up by accessing its local port:

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Chens-MBP",
  "cluster_name" : "elasticsearch_gwenshap",
  "cluster_uuid" : "X69zu3_sQNGb7zbMh7NDVw",
  "version" : {
    "number" : "7.5.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8bec50e1e0ad29dad5653712cf3bb580cd1afcdf",
    "build_date" : "2020-01-15T12:11:52.313576Z",
    "build_snapshot" : false,
    "lucene_version" : "8.3.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Now create and start the connector:

```
echo '{"name":"elastic-login-connector", "config":{"connector.class":"Elastic-
searchSinkConnector","connection.url":"http://localhost:
9200","type.name":"mysql-data","topics":"mysql.login","key.ignore":true}}' |
curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"

{
  "name": "elastic-login-connector",
  "config": {
    "connector.class": "ElasticsearchSinkConnector",
    "connection.url": "http://localhost:9200",
    "topics": "mysql.login",
    "key.ignore": "true",
    "name": "elastic-login-connector"
  },
  "tasks": [
    {
      "connector": "elastic-login-connector",
      "task": 0
    }
  ]
}
```

There are a few configurations we need to explain here. The `connection.url` is simply the URL of the local Elasticsearch server we configured earlier. Each topic in Kafka will become, by default, a separate Elasticsearch index, with the same name as the topic. The only topic we are writing to Elasticsearch is `mysql.login`. The JDBC connector does not populate the message key. As a result, the events in Kafka have null keys. Because the events in Kafka lack keys, we need to tell the Elasticsearch connector to use the topic name, partition ID, and offset as the key for each event. This is done by setting `key.ignore` configuration to `true`.

Let's check that the index with `mysql.login` data was created:

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index       uuid                   pri rep docs.count
docs.deleted store.size pri.store.size
yellow open   mysql.login wkeyk9-bQea6NJmAFjv4hw   1   1          2
0      3.9kb          3.9kb
```

If the index isn't there, look for errors in the Connect worker log. Missing configurations or libraries are common causes for errors. If all is well, we can search the index for our records:

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=true"
{
  "took" : 40,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
```

```
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "mysql.login",
        "_type" : "_doc",
        "_id" : "mysql.login+0+0",
        "_score" : 1.0,
        "_source" : {
          "username" : "gwenshap",
          "login_time" : 1621699811000
        }
      },
      {
        "_index" : "mysql.login",
        "_type" : "_doc",
        "_id" : "mysql.login+0+1",
        "_score" : 1.0,
        "_source" : {
          "username" : "tpalino",
          "login_time" : 1621699816000
        }
      }
    ]
  }
}
```

If you add new records to the table in MySQL, they will automatically appear in the `mysql.login` topic in Kafka and in the corresponding Elasticsearch index.

Now that we've seen how to build and install the JDBC source and Elasticsearch sink, we can build and use any pair of connectors that suits our use case. Confluent maintains a set of their own prebuilt connectors, as well as some from across the community and other vendors, at Confluent Hub. You can pick any connector on the list that you wish to try out, download it, configure it—either based on the documentation or by pulling the configuration from the REST API—and run it on your Connect worker cluster.

Build Your Own Connectors

The Connector API is public and anyone can create a new connector. So if the datastore you wish to integrate with does not have an existing connector, we encourage you to write your own. You can then contribute it to Confluent Hub so others can discover and use it. It is beyond the scope of this chapter to discuss all the details involved in building a connector, but there are multiple blog posts that explain how to do so, and good talks from Kafka Summit NY 2019, Kafka Summit London 2018, and ApacheCon. We also recommend looking at the existing connectors as a starting point and perhaps jump-starting using an Apache Maven archtype. We always encourage you to ask for assistance or show off your latest connectors on the Apache Kafka community mailing list (*users@kafka.apache.org*) or submit them to Confluent Hub so they can be easily found.

Single Message Transformations

Copying records from MySQL to Kafka and from there to Elastic is rather useful on its own, but ETL pipelines typically involve a transformation step. In the Kafka ecosystem we separate transformations to single message transformations (SMTs), which are stateless, and stream processing, which can be stateful. SMTs can be done within Kafka Connect transforming messages while they are being copied, often without writing any code. More complex transformations, which typically involve joins or aggregation, will require the stateful Kafka Streams framework. We'll discuss Kafka Streams in a later chapter.

Apache Kafka includes the following SMTs:

Cast
: Change data type of a field.

MaskField
: Replace the contents of a field with null. This is useful for removing sensitive or personally identifying data.

Filter
: Drop or include all messages that match a specific condition. Built-in conditions include matching on a topic name, a particular header, or whether the message is a tombstone (that is, has a null value).

Flatten
: Transform a nested data structure to a flat one. This is done by concatenating all the names of all fields in the path to a specific value.

HeaderFrom
: Move or copy fields from the message into the header.

InsertHeader
: Add a static string to the header of each message.

InsertField
: Add a new field to a message, either using values from its metadata such as offset, or with a static value.

RegexRouter
: Change the destination topic using a regular expression and a replacement string.

ReplaceField
: Remove or rename a field in the message.

TimestampConverter
: Modify the time format of a field—for example, from Unix Epoch to a String.

TimestampRouter
: Modify the topic based on the message timestamp. This is mostly useful in sink connectors when we want to copy messages to specific table partitions based on their timestamp and the topic field is used to find an equivalent dataset in the destination system.

In addition, transformations are available from contributors outside the main Apache Kafka code base. Those can be found on GitHub (Lenses.io, Aiven, and Jeremy Custenborder have useful collections) or on Confluent Hub.

To learn more about Kafka Connect SMTs, you can read detailed examples of many transformations in the "Twelve Days of SMT" blog series. In addition, you can learn how to write your own transformations by following a tutorial and deep dive.

As an example, let's say that we want to add a record header to each record produced by the MySQL connector we created previously. The header will indicate that the record was created by this MySQL connector, which is useful in case auditors want to examine the lineage of these records.

To do this, we'll replace the previous MySQL connector configuration with the following:

```
echo '{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
```

```
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector",
    "transforms": "InsertHeader",
    "transforms.InsertHeader.type":
      "org.apache.kafka.connect.transforms.InsertHeader",
    "transforms.InsertHeader.header": "MessageSource",
    "transforms.InsertHeader.value.literal": "mysql-login-connector"
  }}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
```

Now, if you insert a few more records into the MySQL table that we created in the previous example, you'll be able to see that the new messages in the `mysql.login` topic have headers (note that you'll need Apache Kafka 2.7 or higher to print headers in the console consumer):

```
bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic
mysql.login --from-beginning --property print.headers=true

NO_HEADERS      {"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time-
stamp","version":1,"field":"login_time"}],"optional":false,"name":"login"},"pay-
load":{"username":"tpalino","login_time":1621699816000}}
MessageSource:mysql-login-connector     {"schema":{"type":"struct","fields":

[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time-
stamp","version":1,"field":"login_time"}],"optional":false,"name":"login"},"pay-
load":{"username":"rajini","login_time":1621803287000}}
```

As you can see, the old records show `NO_HEADERS`, but the new records show `MessageSource:mysql-login-connector`.

Error Handling and Dead Letter Queues

Transforms is an example of a connector config that isn't specific to one connector but can be used in the configuration of any connector. Another very useful connector configuration that can be used in any sink connector is `error.tolerance`—you can configure any connector to silently drop corrupt messages, or to route them to a special topic called a "dead letter queue." You can find more details in the "Kafka Connect Deep Dive—Error Handling and Dead Letter Queues" blog post.

A Deeper Look at Kafka Connect

To understand how Kafka Connect works, you need to understand three basic concepts and how they interact. As we explained earlier and demonstrated with

examples, to use Kafka Connect, you need to run a cluster of workers and create/remove connectors. An additional detail we did not dive into before is the handling of data by converters—these are the components that convert MySQL rows to JSON records, which the connector wrote into Kafka.

Let's look a bit deeper into each system and how they interact with one another.

Connectors and tasks

Connector plug-ins implement the Connector API, which includes two parts:

Connectors
: The connector is responsible for three important things:

  - Determining how many tasks will run for the connector
  - Deciding how to split the data-copying work between the tasks
  - Getting configurations for the tasks from the workers and passing them along

  For example, the JDBC source connector will connect to the database, discover the existing tables to copy, and based on that decide how many tasks are needed—choosing the lower of `tasks.max` configuration and the number of tables. Once it decides how many tasks will run, it will generate a configuration for each task—using both the connector configuration (e.g., `connection.url`) and a list of tables it assigns for each task to copy. The `taskConfigs()` method returns a list of maps (i.e., a configuration for each task we want to run). The workers are then responsible for starting the tasks and giving each one its own unique configuration so that it will copy a unique subset of tables from the database. Note that when you start the connector via the REST API, it may start on any node, and subsequently the tasks it starts may also execute on any node.

Tasks
: Tasks are responsible for actually getting the data in and out of Kafka. All tasks are initialized by receiving a context from the worker. Source context includes an object that allows the source task to store the offsets of source records (e.g., in the file connector, the offsets are positions in the file; in the JDBC source connector, the offsets can be a timestamp column in a table). Context for the sink connector includes methods that allow the connector to control the records it receives from Kafka—this is used for things like applying back pressure and retrying and storing offsets externally for exactly-once delivery. After tasks are initialized, they are started with a `Properties` object that contains the configuration the `Connector` created for the task. Once tasks are started, source tasks poll an external system and return lists of records that the worker sends to Kafka brokers. Sink tasks receive records from Kafka through the worker and are responsible for writing the records to an external system.

Workers

Kafka Connect's worker processes are the "container" processes that execute the connectors and tasks. They are responsible for handling the HTTP requests that define connectors and their configuration, as well as for storing the connector configuration in an internal Kafka topic, starting the connectors and their tasks, and passing the appropriate configurations along. If a worker process is stopped or crashes, other workers in a Connect cluster will recognize that (using the heartbeats in Kafka's consumer protocol) and reassign the connectors and tasks that ran on that worker to the remaining workers. If a new worker joins a Connect cluster, other workers will notice that and assign connectors or tasks to it to make sure load is balanced among all workers fairly. Workers are also responsible for automatically committing offsets for both source and sink connectors into internal Kafka topics and for handling retries when tasks throw errors.

The best way to understand workers is to realize that connectors and tasks are responsible for the "moving data" part of data integration, while the workers are responsible for the REST API, configuration management, reliability, high availability, scaling, and load balancing.

This separation of concerns is the main benefit of using the Connect API versus the classic consumer/producer APIs. Experienced developers know that writing code that reads data from Kafka and inserts it into a database takes maybe a day or two, but if you need to handle configuration, errors, REST APIs, monitoring, deployment, scaling up and down, and handling failures, it can take a few months to get everything right. And most data integration pipelines involve more than just the one source or target. So now consider that effort spent on bespoke code for just a database integration, repeated many times for other technologies. If you implement data copying with a connector, your connector plugs into workers that handle a bunch of complicated operational issues that you don't need to worry about.

Converters and Connect's data model

The last piece of the Connect API puzzle is the connector data model and the converters. Kafka's Connect API includes a data API, which includes both data objects and a schema that describes that data. For example, the JDBC source reads a column from a database and constructs a `Connect Schema` object based on the data types of the columns returned by the database. It then uses the schema to construct a `Struct` that contains all the fields in the database record. For each column, we store the column name and the value in that column. Every source connector does something similar—read an event from the source system and generate a `Schema` and `Value` pair. Sink connectors do the opposite—get a `Schema` and `Value` pair and use the `Schema` to parse the values and insert them into the target system.

Though source connectors know how to generate objects based on the Data API, there is still a question of how `Connect` workers store these objects in Kafka. This is where the converters come in. When users configure the worker (or the connector), they choose which converter they want to use to store data in Kafka. At the moment, the available choices are primitive types, byte arrays, strings, Avro, JSON, JSON schemas, or Protobufs. The JSON converter can be configured to either include a schema in the result record or not include one—so we can support both structured and semi-structured data. When the connector returns a Data API record to the worker, the worker then uses the configured converter to convert the record to an Avro object, a JSON object, or a string, and the result is then stored into Kafka.

The opposite process happens for sink connectors. When the Connect worker reads a record from Kafka, it uses the configured converter to convert the record from the format in Kafka (i.e., primitive types, byte arrays, strings, Avro, JSON, JSON schema, or Protobufs) to the Connect Data API record and then passes it to the sink connector, which inserts it into the destination system.

This allows the Connect API to support different types of data stored in Kafka, independent of the connector implementation (i.e., any connector can be used with any record type, as long as a converter is available).

Offset management

Offset management is one of the convenient services the workers perform for the connectors (in addition to deployment and configuration management via the REST API). The idea is that connectors need to know which data they have already processed, and they can use APIs provided by Kafka to maintain information on which events were already processed.

For source connectors, this means that the records the connector returns to the Connect workers include a logical partition and a logical offset. Those are not Kafka partitions and Kafka offsets but rather partitions and offsets as needed in the source system. For example, in the file source, a partition can be a file and an offset can be a line number or character number in the file. In a JDBC source, a partition can be a database table and the offset can be an ID or timestamp of a record in the table. One of the most important design decisions involved in writing a source connector is deciding on a good way to partition the data in the source system and to track offsets—this will impact the level of parallelism the connector can achieve and whether it can deliver at-least-once or exactly-once semantics.

When the source connector returns a list of records, which includes the source partition and offset for each record, the worker sends the records to Kafka brokers. If the brokers successfully acknowledge the records, the worker then stores the offsets of the records it sent to Kafka. This allows connectors to start processing events from the most recently stored offset after a restart or a crash. The storage mechanism is

pluggable and is usually a Kafka topic; you can control the topic name with the `offset.storage.topic` configuration. In addition, Connect uses Kafka topics to store the configuration of all the connectors we've created and the status of each connector—these use names configured by `config.storage.topic` and `status.storage.topic`, respectively.

Sink connectors have an opposite but similar workflow: they read Kafka records, which already have a topic, partition, and offset identifiers. Then they call the `connector put()` method that should store those records in the destination system. If the connector reports success, they commit the offsets they've given to the connector back to Kafka, using the usual consumer commit methods.

Offset tracking provided by the framework itself should make it easier for developers to write connectors and guarantee some level of consistent behavior when using different connectors.

Alternatives to Kafka Connect

So far we've looked at Kafka's Connect API in great detail. While we love the convenience and reliability the Connect API provides, it is not the only method for getting data in and out of Kafka. Let's look at other alternatives and when they are commonly used.

Ingest Frameworks for Other Datastores

While we like to think that Kafka is the center of the universe, some people disagree. Some people build most of their data architectures around systems like Hadoop or Elasticsearch. Those systems have their own data ingestion tools—Flume for Hadoop, and Logstash or Fluentd for Elasticsearch. We recommend Kafka's Connect API when Kafka is an integral part of the architecture and when the goal is to connect large numbers of sources and sinks. If you are actually building a Hadoop-centric or Elastic-centric system and Kafka is just one of many inputs into that system, then using Flume or Logstash makes sense.

GUI-Based ETL Tools

Old-school systems like Informatica, open source alternatives like Talend and Pentaho, and even newer alternatives such as Apache NiFi and StreamSets, support Apache Kafka as both a data source and a destination. If you are already using these systems—if you already do everything using Pentaho, for example—you may not be interested in adding another data integration system just for Kafka. They also make sense if you are using a GUI-based approach to building ETL pipelines. The main drawback of these systems is that they are usually built for involved workflows and will be a somewhat heavy and involved solution if all you want to do is get data in and

out of Kafka. We believe that data integration should focus on faithful delivery of messages under all conditions, while most ETL tools add unnecessary complexity.

We do encourage you to look at Kafka as a platform that can handle data integration (with Connect), application integration (with producers and consumers), and stream processing. Kafka could be a viable replacement for an ETL tool that only integrates data stores.

Stream Processing Frameworks

Almost all stream processing frameworks include the ability to read events from Kafka and write them to a few other systems. If your destination system is supported and you already intend to use that stream processing framework to process events from Kafka, it seems reasonable to use the same framework for data integration as well. This often saves a step in the stream processing workflow (no need to store processed events in Kafka—just read them out and write them to another system), with the drawback that it can be more difficult to troubleshoot things like lost and corrupted messages.

Summary

In this chapter we discussed the use of Kafka for data integration. Starting with reasons to use Kafka for data integration, we covered general considerations for data integration solutions. We showed why we think Kafka and its Connect API are a good fit. We then gave several examples of how to use Kafka Connect in different scenarios, spent some time looking at how Connect works, and then discussed a few alternatives to Kafka Connect.

Whatever data integration solution you eventually land on, the most important feature will always be its ability to deliver all messages under all failure conditions. We believe that Kafka Connect is extremely reliable—based on its integration with Kafka's tried-and-true reliability features—but it is important that you test the system of your choice, just like we do. Make sure your data integration system of choice can survive stopped processes, crashed machines, network delays, and high loads without missing a message. After all, at their heart, data integration systems only have one job—delivering those messages.

Of course, while reliability is usually the most important requirement when integrating data systems, it is only one requirement. When choosing a data system, it is important to first review your requirements (refer to "Considerations When Building Data Pipelines" on page 204 for examples) and then make sure your system of choice satisfies them. But this isn't enough—you must also learn your data integration solution well enough to be certain that you are using it in a way that supports your requirements. It isn't enough that Kafka supports at-least-once semantics; you must

be sure you aren't accidentally configuring it in a way that may end up with less than complete reliability.