

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

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

DATA | DATABASES



9 781492 087366

"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.

Twitter: @oreillymedia
facebook.com/oreilly



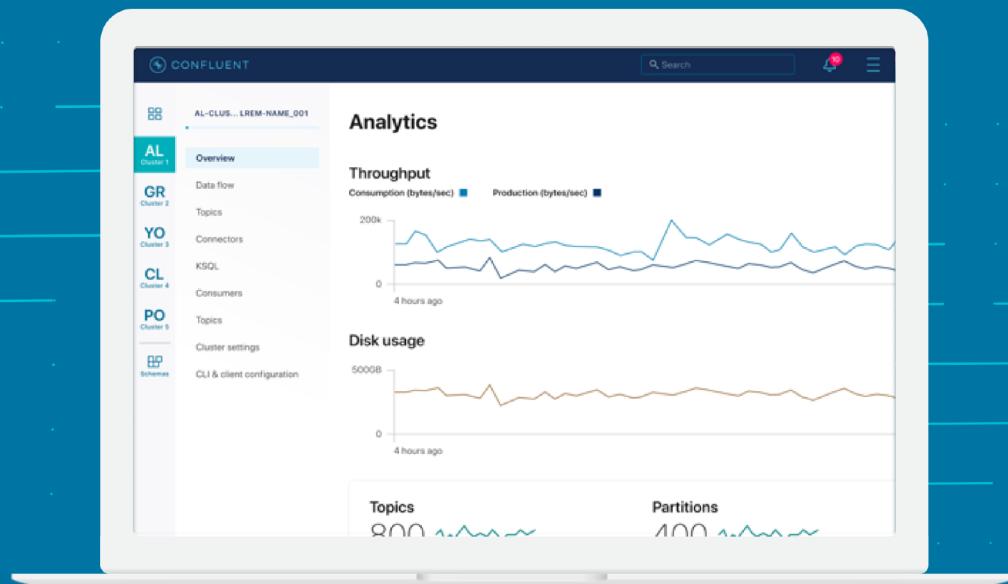
Fully-Managed Apache Kafka® Service

✓ Kafka made serverless with elastic scalability and infinite retention

✓ Complete event streaming platform with 100+ connectors and ksqlDB

✓ Start streaming in minutes with self-serve provisioning

USE ANY POPULAR CLOUD PROVIDER



Try Confluent Cloud **Free for 90 Days**

New signups get \$200 per month for your first 3 months, plus use promo code **KTDG2021** for an additional \$200 credit!

TRY FREE

Claim your promo code in-product

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.

—Suman 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

Copyright © 2022 Chen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty. All rights reserved.

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

Indexer: Ellen Troutman-Zaig

Development Editor: Gary O'Brien

Interior Designer: David Futato

Production Editor: Kate Galloway

Cover Designer: Karen Montgomery

Copyeditor: Sonia Saruba

Illustrator: Kate Dullea

Proofreader: Piper Editorial Consulting, LLC

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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Kafka: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Confluent. See our [statement of editorial independence](#).

978-1-492-08736-6

[LSI]

Table of Contents

Foreword to the Second Edition.....	xiii
Foreword to the First Edition.....	xv
Preface.....	xix
1. Meet Kafka.....	1
Publish/Subscribe Messaging	1
How It Starts	2
Individual Queue Systems	3
Enter Kafka	4
Messages and Batches	4
Schemas	5
Topics and Partitions	5
Producers and Consumers	6
Brokers and Clusters	8
Multiple Clusters	9
Why Kafka?	10
Multiple Producers	10
Multiple Consumers	10
Disk-Based Retention	11
Scalable	11
High Performance	11
Platform Features	11
The Data Ecosystem	12
Use Cases	12
Kafka's Origin	14

LinkedIn's Problem	14
The Birth of Kafka	15
Open Source	16
Commercial Engagement	16
The Name	17
Getting Started with Kafka	17
2. Installing Kafka.....	19
Environment Setup	19
Choosing an Operating System	19
Installing Java	19
Installing ZooKeeper	20
Installing a Kafka Broker	23
Configuring the Broker	24
General Broker Parameters	25
Topic Defaults	27
Selecting Hardware	33
Disk Throughput	33
Disk Capacity	34
Memory	34
Networking	35
CPU	35
Kafka in the Cloud	35
Microsoft Azure	36
Amazon Web Services	36
Configuring Kafka Clusters	36
How Many Brokers?	37
Broker Configuration	38
OS Tuning	38
Production Concerns	42
Garbage Collector Options	42
Datacenter Layout	43
Colocating Applications on ZooKeeper	44
Summary	46
3. Kafka Producers: Writing Messages to Kafka.....	47
Producer Overview	48
Constructing a Kafka Producer	50
Sending a Message to Kafka	52
Sending a Message Synchronously	52
Sending a Message Asynchronously	53

Configuring Producers	54
client.id	55
acks	55
Message Delivery Time	56
linger.ms	59
buffer.memory	59
compression.type	59
batch.size	59
max.in.flight.requests.per.connection	60
max.request.size	60
receive.buffer.bytes and send.buffer.bytes	61
enable.idempotence	61
Serializers	61
Custom Serializers	62
Serializing Using Apache Avro	64
Using Avro Records with Kafka	65
Partitions	68
Headers	71
Interceptors	71
Quotas and Throttling	73
Summary	75
4. Kafka Consumers: Reading Data from Kafka.....	77
Kafka Consumer Concepts	77
Consumers and Consumer Groups	77
Consumer Groups and Partition Rebalance	80
Static Group Membership	83
Creating a Kafka Consumer	84
Subscribing to Topics	85
The Poll Loop	86
Thread Safety	87
Configuring Consumers	88
fetch.min.bytes	88
fetch.max.wait.ms	88
fetch.max.bytes	89
max.poll.records	89
max.partition.fetch.bytes	89
session.timeout.ms and heartbeat.interval.ms	89
max.poll.interval.ms	90
default.api.timeout.ms	90
request.timeout.ms	90

auto.offset.reset	91
enable.auto.commit	91
partition.assignment.strategy	91
client.id	93
client.rack	93
group.instance.id	93
receive.buffer.bytes and send.buffer.bytes	93
offsets.retention.minutes	93
Commits and Offsets	94
Automatic Commit	95
Commit Current Offset	96
Asynchronous Commit	97
Combining Synchronous and Asynchronous Commits	99
Committing a Specified Offset	100
Rebalance Listeners	101
Consuming Records with Specific Offsets	104
But How Do We Exit?	105
Deserializers	106
Custom Deserializers	107
Using Avro Deserialization with Kafka Consumer	109
Standalone Consumer: Why and How to Use a Consumer Without a Group	110
Summary	111
5. Managing Apache Kafka Programmatically.....	113
AdminClient Overview	114
Asynchronous and Eventually Consistent API	114
Options	114
Flat Hierarchy	115
Additional Notes	115
AdminClient Lifecycle: Creating, Configuring, and Closing	115
client.dns.lookup	116
request.timeout.ms	117
Essential Topic Management	118
Configuration Management	121
Consumer Group Management	123
Exploring Consumer Groups	123
Modifying Consumer Groups	125
Cluster Metadata	127
Advanced Admin Operations	127
Adding Partitions to a Topic	127
Deleting Records from a Topic	128

Leader Election	128
Reassigning Replicas	129
Testing	131
Summary	133
6. Kafka Internals.....	135
Cluster Membership	135
The Controller	136
KRaft: Kafka's New Raft-Based Controller	137
Replication	139
Request Processing	142
Produce Requests	144
Fetch Requests	145
Other Requests	147
Physical Storage	149
Tiered Storage	149
Partition Allocation	151
File Management	152
File Format	153
Indexes	155
Compaction	156
How Compaction Works	156
Deleted Events	158
When Are Topics Compacted?	159
Summary	159
7. Reliable Data Delivery.....	161
Reliability Guarantees	162
Replication	163
Broker Configuration	164
Replication Factor	165
Unclean Leader Election	166
Minimum In-Sync Replicas	167
Keeping Replicas In Sync	168
Persisting to Disk	169
Using Producers in a Reliable System	169
Send Acknowledgments	170
Configuring Producer Retries	171
Additional Error Handling	171
Using Consumers in a Reliable System	172
Important Consumer Configuration Properties for Reliable Processing	173

Explicitly Committing Offsets in Consumers	174
Validating System Reliability	176
Validating Configuration	176
Validating Applications	177
Monitoring Reliability in Production	178
Summary	180
8. Exactly-Once Semantics.....	181
Idempotent Producer	182
How Does the Idempotent Producer Work?	182
Limitations of the Idempotent Producer	184
How Do I Use the Kafka Idempotent Producer?	185
Transactions	186
Transactions Use Cases	187
What Problems Do Transactions Solve?	187
How Do Transactions Guarantee Exactly-Once?	188
What Problems Aren't Solved by Transactions?	191
How Do I Use Transactions?	193
Transactional IDs and Fencing	196
How Transactions Work	198
Performance of Transactions	200
Summary	201
9. Building Data Pipelines.....	203
Considerations When Building Data Pipelines	204
Timeliness	204
Reliability	205
High and Varying Throughput	205
Data Formats	206
Transformations	207
Security	208
Failure Handling	209
Coupling and Agility	209
When to Use Kafka Connect Versus Producer and Consumer	210
Kafka Connect	211
Running Kafka Connect	211
Connector Example: File Source and File Sink	214
Connector Example: MySQL to Elasticsearch	216
Single Message Transformations	223
A Deeper Look at Kafka Connect	225
Alternatives to Kafka Connect	229

Ingest Frameworks for Other Datastores	229
GUI-Based ETL Tools	229
Stream Processing Frameworks	230
Summary	230
10. Cross-Cluster Data Mirroring.....	233
Use Cases of Cross-Cluster Mirroring	234
Multicloud Architectures	235
Some Realities of Cross-Datacenter Communication	235
Hub-and-Spoke Architecture	236
Active-Active Architecture	238
Active-Standby Architecture	240
Stretch Clusters	246
Apache Kafka's MirrorMaker	247
Configuring MirrorMaker	249
Multicloud Replication Topology	251
Securing MirrorMaker	252
Deploying MirrorMaker in Production	253
Tuning MirrorMaker	257
Other Cross-Cluster Mirroring Solutions	259
Uber uReplicator	259
LinkedIn Brooklin	260
Confluent Cross-Datacenter Mirroring Solutions	261
Summary	263
11. Securing Kafka.....	265
Locking Down Kafka	265
Security Protocols	268
Authentication	269
SSL	270
SASL	275
Reauthentication	286
Security Updates Without Downtime	288
Encryption	289
End-to-End Encryption	289
Authorization	291
AclAuthorizer	292
Customizing Authorization	295
Security Considerations	297
Auditing	298
Securing ZooKeeper	299

SASL	299
SSL	300
Authorization	301
Securing the Platform	301
Password Protection	301
Summary	303
12. Administering Kafka.....	305
Topic Operations	305
Creating a New Topic	306
Listing All Topics in a Cluster	308
Describing Topic Details	308
Adding Partitions	310
Reducing Partitions	311
Deleting a Topic	311
Consumer Groups	312
List and Describe Groups	312
Delete Group	313
Offset Management	314
Dynamic Configuration Changes	315
Overriding Topic Configuration Defaults	315
Overriding Client and User Configuration Defaults	317
Overriding Broker Configuration Defaults	318
Describing Configuration Overrides	319
Removing Configuration Overrides	319
Producing and Consuming	320
Console Producer	320
Console Consumer	322
Partition Management	326
Preferred Replica Election	326
Changing a Partition's Replicas	327
Dumping Log Segments	332
Replica Verification	334
Other Tools	334
Unsafe Operations	335
Moving the Cluster Controller	335
Removing Topics to Be Deleted	336
Deleting Topics Manually	336
Summary	337

13. Monitoring Kafka.....	339
Metric Basics	339
Where Are the Metrics?	339
What Metrics Do I Need?	341
Application Health Checks	343
Service-Level Objectives	343
Service-Level Definitions	343
What Metrics Make Good SLIs?	344
Using SLOs in Alerting	345
Kafka Broker Metrics	346
Diagnosing Cluster Problems	347
The Art of Under-Replicated Partitions	348
Broker Metrics	354
Topic and Partition Metrics	364
JVM Monitoring	366
OS Monitoring	367
Logging	369
Client Monitoring	370
Producer Metrics	370
Consumer Metrics	373
Quotas	376
Lag Monitoring	377
End-to-End Monitoring	378
Summary	378
14. Stream Processing.....	381
What Is Stream Processing?	382
Stream Processing Concepts	385
Topology	385
Time	386
State	388
Stream-Table Duality	389
Time Windows	390
Processing Guarantees	392
Stream Processing Design Patterns	392
Single-Event Processing	392
Processing with Local State	393
Multiphase Processing/Repartitioning	395
Processing with External Lookup: Stream-Table Join	396
Table-Table Join	398
Streaming Join	398

Out-of-Sequence Events	399
Reprocessing	400
Interactive Queries	401
Kafka Streams by Example	402
Word Count	402
Stock Market Statistics	405
ClickStream Enrichment	408
Kafka Streams: Architecture Overview	410
Building a Topology	410
Optimizing a Topology	411
Testing a Topology	411
Scaling a Topology	412
Surviving Failures	415
Stream Processing Use Cases	416
How to Choose a Stream Processing Framework	417
Summary	419
A. Installing Kafka on Other Operating Systems.....	421
B. Additional Kafka Tools.....	427
Index.....	433

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 trade-offs 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 Nar-khede, 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](#).

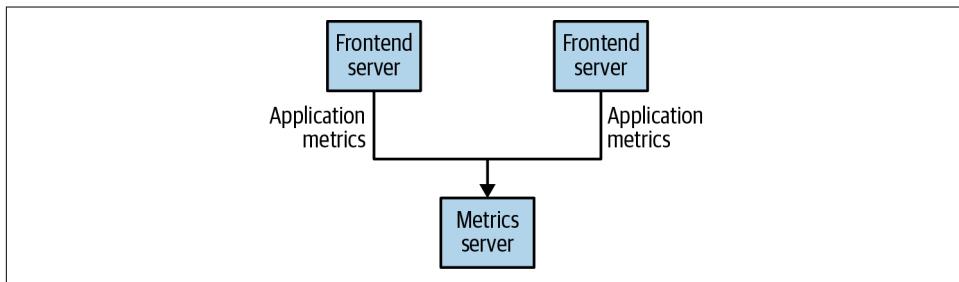


Figure 1-1. A single, direct metrics publisher

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.

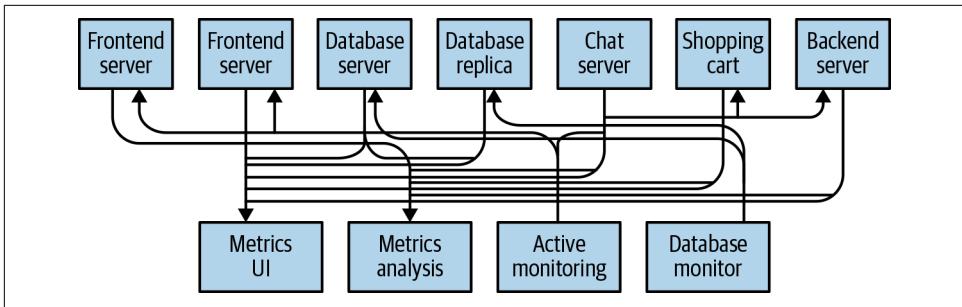


Figure 1-2. Many metrics publishers, using direct connections

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!

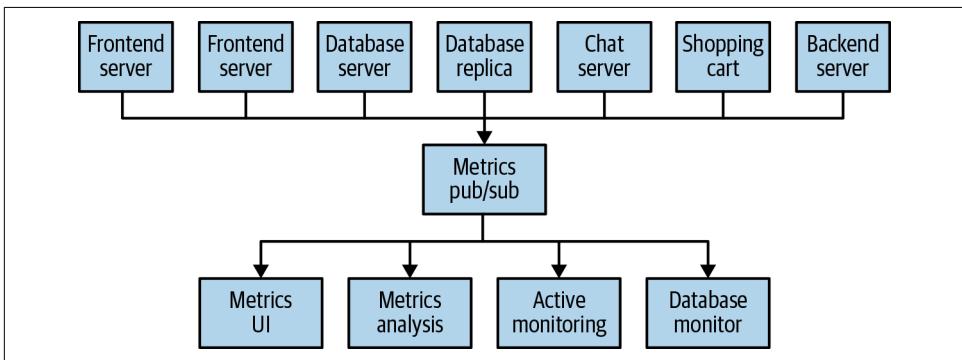


Figure 1-3. A metrics publish/subscribe system

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.

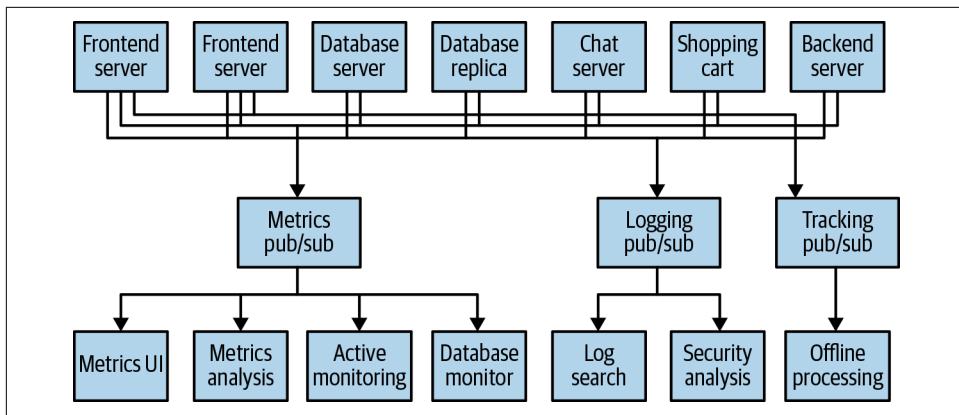


Figure 1-4. Multiple publish/subscribe systems

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.

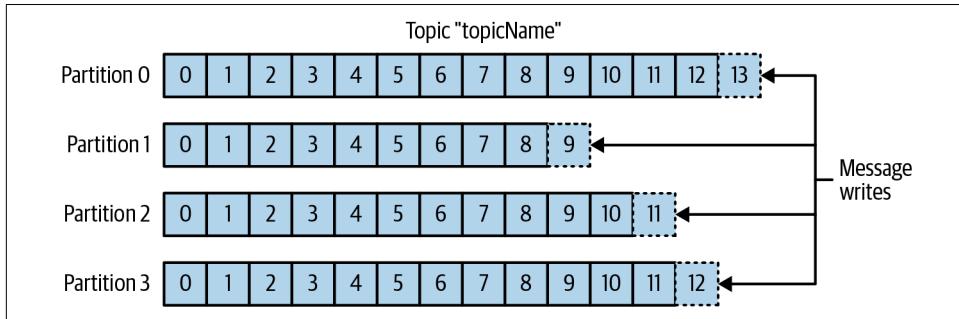


Figure 1-5. Representation of a topic with multiple partitions

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 reassigned the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in [Chapter 4](#).

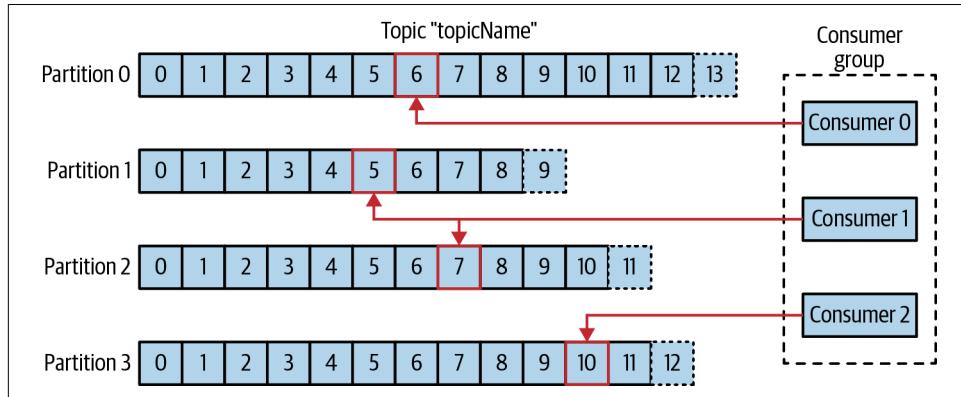


Figure 1-6. A consumer group reading from a topic

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](#).

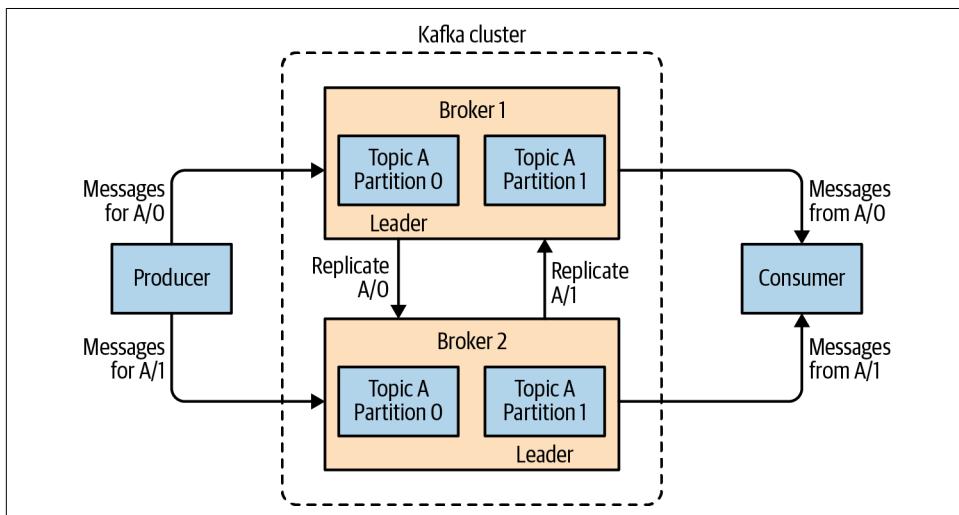


Figure 1-7. Replication of partitions in a cluster

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](#).

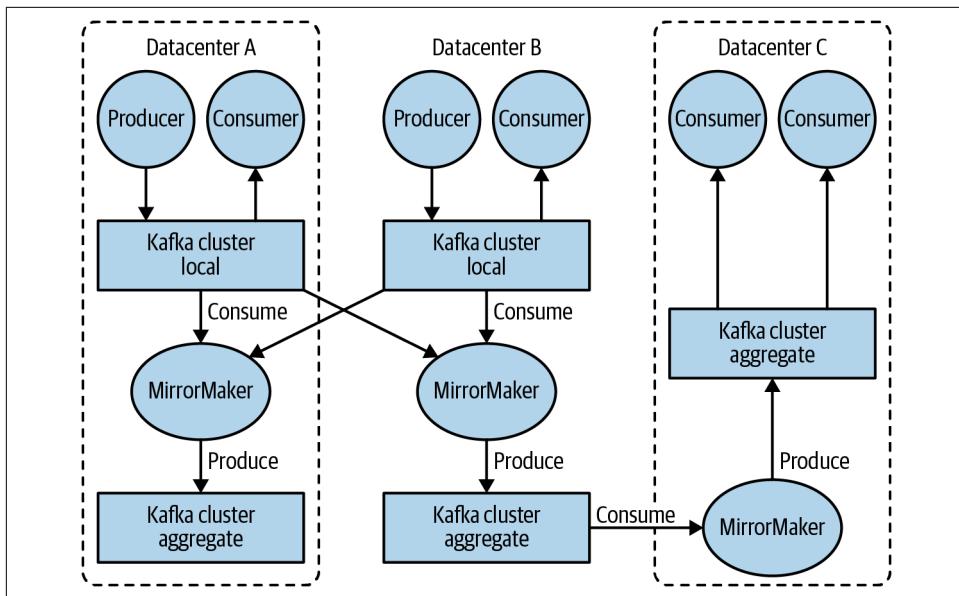


Figure 1-8. Multiple datacenters architecture

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.

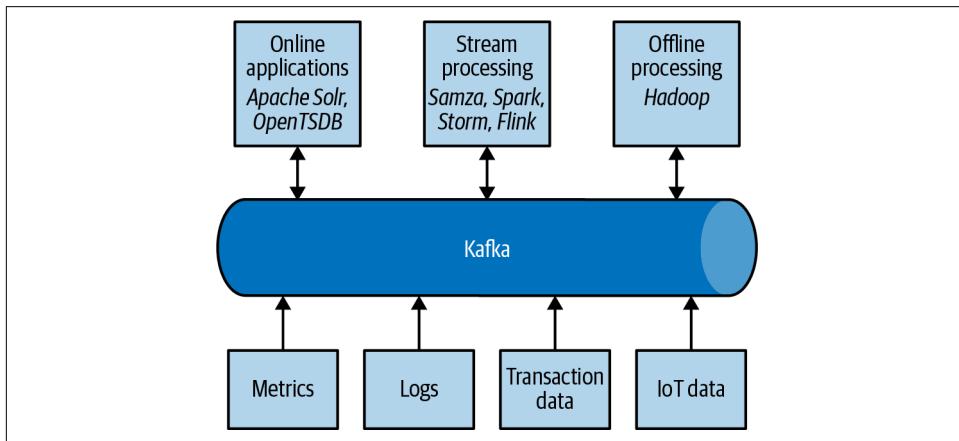


Figure 1-9. A big data ecosystem

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.

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.

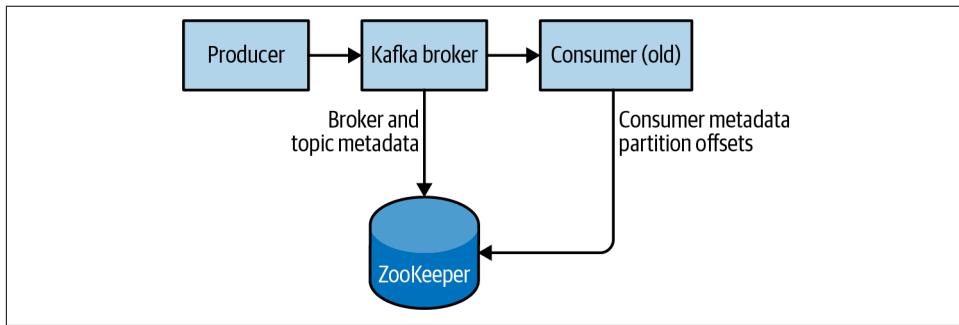


Figure 2-1. Kafka and ZooKeeper

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` $20 \times 2,000$ 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.

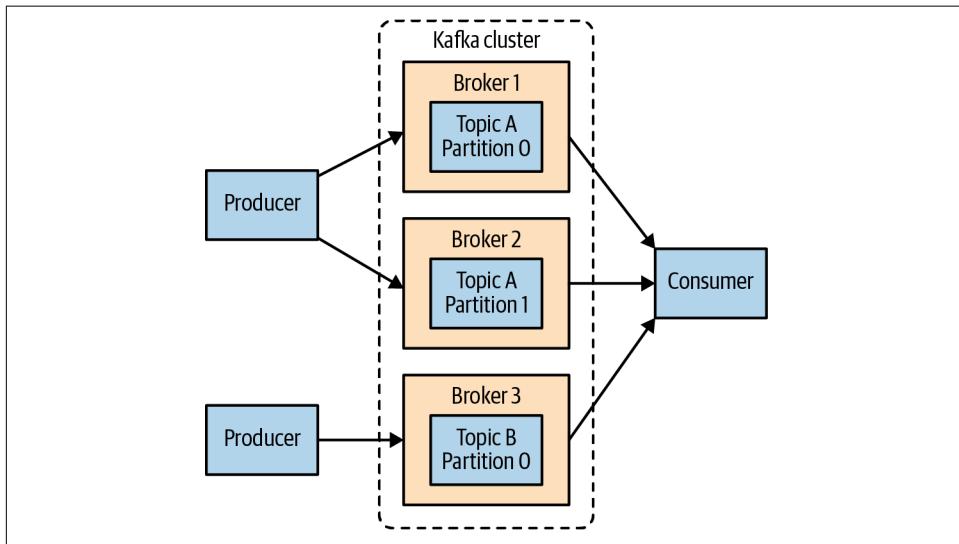


Figure 2-2. A simple Kafka cluster

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 $(\text{number_of_partitions}) \times (\text{partition_size}/\text{segment_size})$ 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 recommended 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](#)).

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.

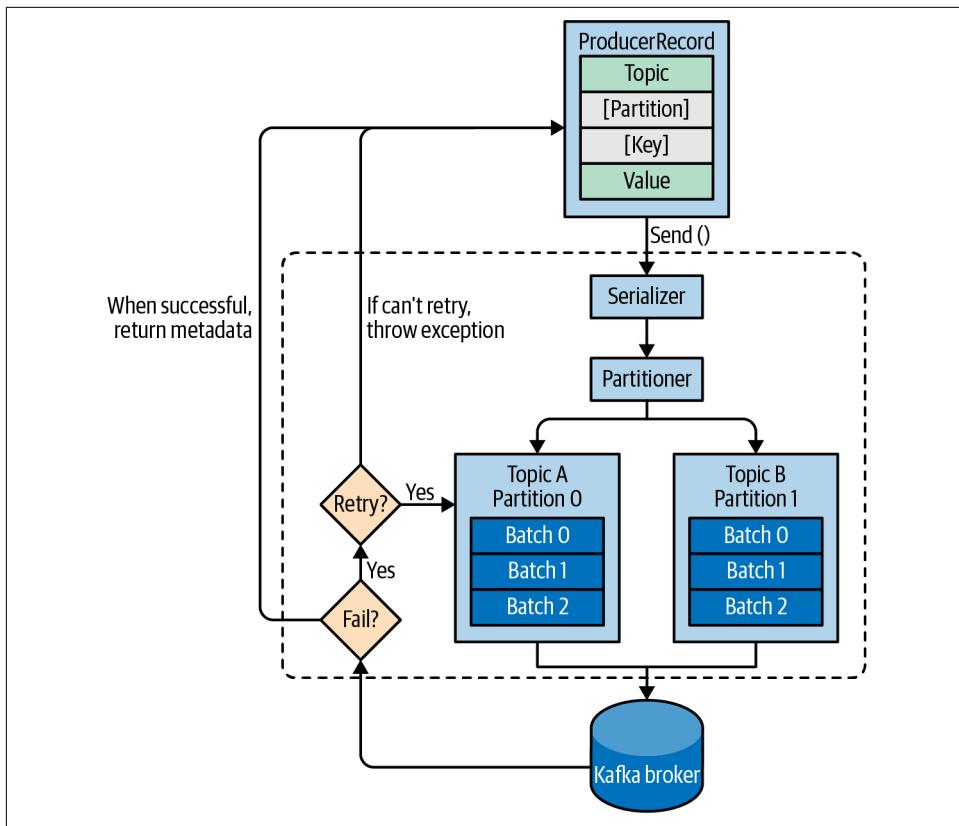


Figure 3-1. High-level overview of Kafka producer components

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(); ①
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

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

```
    "org.apache.kafka.common.serialization.StringSerializer"); ②  
kafkaProps.put("value.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
  
producer = new KafkaProducer<String, String>(kafkaProps); ③
```

- ① We start with a `Properties` object.
- ② Since we plan on using strings for message key and value, we use the built-in `StringSerializer`.
- ③ 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 nonretryable 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 `InterruptedException` 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(); ①  
} catch (Exception e) {  
    e.printStackTrace(); ②  
}
```

- ① 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.
- ② 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-retrievable 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.

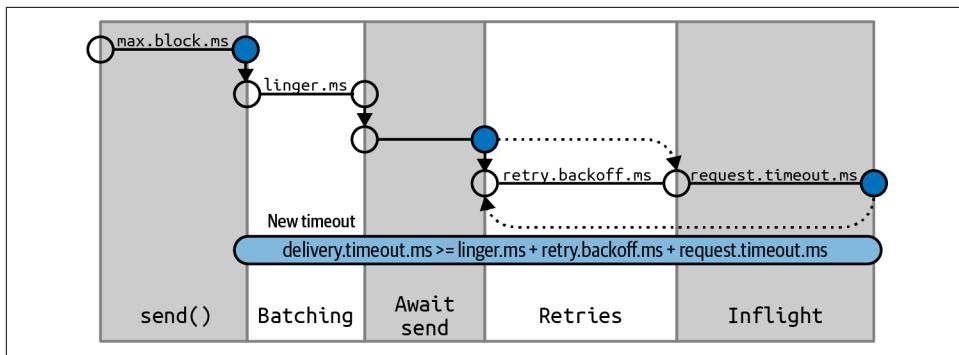


Figure 3-2. Sequence diagram of delivery time breakdown inside Kafka producer

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 `delivery.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 nonretryable 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(Customer record, byte[] buffer) {  
        // Implementation  
    }  
}
```

```

 */
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.

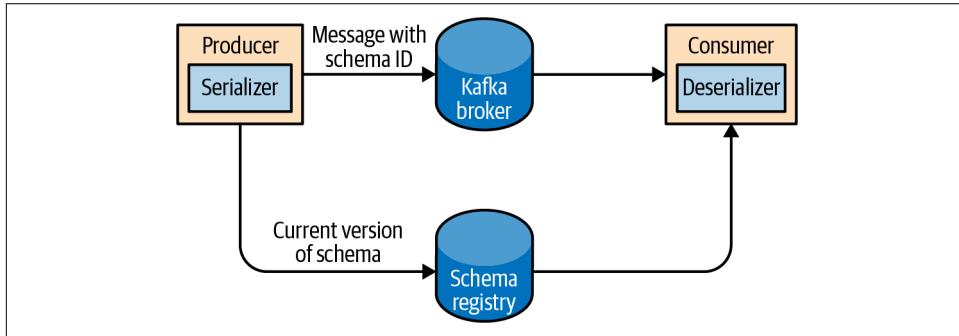


Figure 3-3. Flow diagram of serialization and deserialization of Avro records

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"); ①
props.put("schema.registry.url", schemaUrl); ②

String topic = "customerContacts";

Producer<String, Customer> producer = new KafkaProducer<>(props); ③

// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext(); ④
    System.out.println("Generated customer " +
        customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getName(), customer); ⑤
    producer.send(record); ⑥
}
  
```

- ① 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.

- ② `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.
- ③ `Customer` is our generated object. We tell the producer that our records will contain `Customer` as the value.
- ④ `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.
- ⑤ We also instantiate `ProducerRecord` with `Customer` as the value type, and pass a `Customer` object when creating the new record.
- ⑥ 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"); ①
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ②

String schemaString =
    "{\"namespace\": \"customerManagement.avro\",
     \"type\": \"record\", " + ③
     \"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); ④

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); ⑤
    customer.put("id", nCustomers);
    customer.put("name", name);
    customer.put("email", email);

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

```

- ① We still use the same `KafkaAvroSerializer`.
- ② And we provide the URI of the same Schema Registry.
- ③ But now we also need to provide the Avro schema, since it is not provided by an Avro-generated object.
- ④ Our object type is an Avro `GenericRecord`, which we initialize with our schema and the data we want to write.
- ⑤ 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"); ①
```

- ① 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) {} ①

    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))) ②
            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() {}
}
```

- ① 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`.

- ② 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 tracking 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(); ❷  
        return producerRecord; ❸  
    }  
  
    public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {  
        numAcked.incrementAndGet(); ❹  
    }  
  
    public void close() {  
        executorService.shutdownNow(); ❺  
    }  
  
    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.

- ③ When Kafka responds with an ack, we increment the acknowledgment count and don't need to return anything.
- ④ 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.CountProducer
Interceptor 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 ①
```

```
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 ②
```

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'con-  
sumer_byte_rate=2048' --entity-type users ③
```

- ① Limiting clientC (identified by client-id) to produce only 1024 bytes per second
- ② Limiting user1 (identified by authenticated principal) to produce only 1024 bytes per second and consume only 2048 bytes per second.
- ③ 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.

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](#).

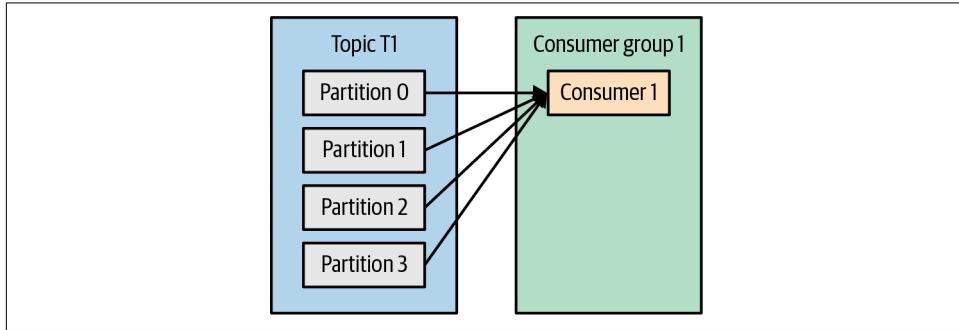


Figure 4-1. One consumer group with four partitions

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](#).

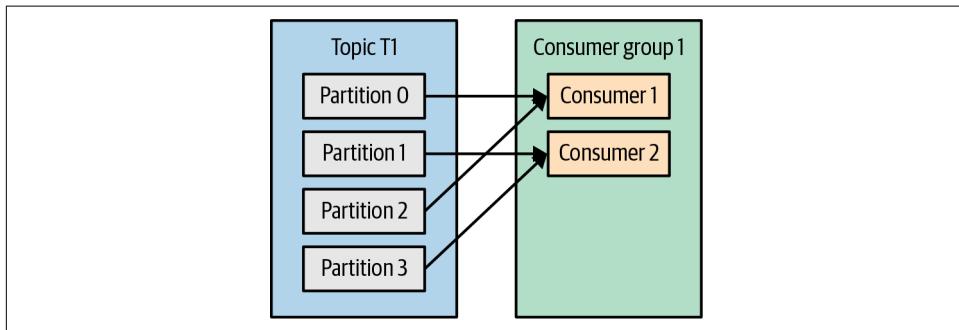


Figure 4-2. Four partitions split to two consumers in a group

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

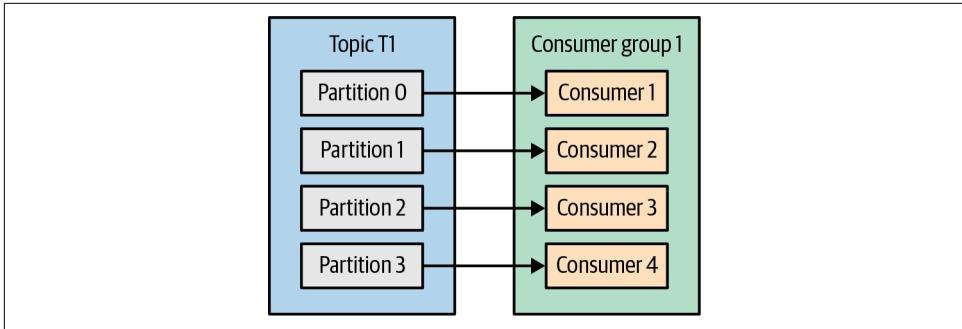


Figure 4-3. Four consumers in a group with one partition each

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](#).

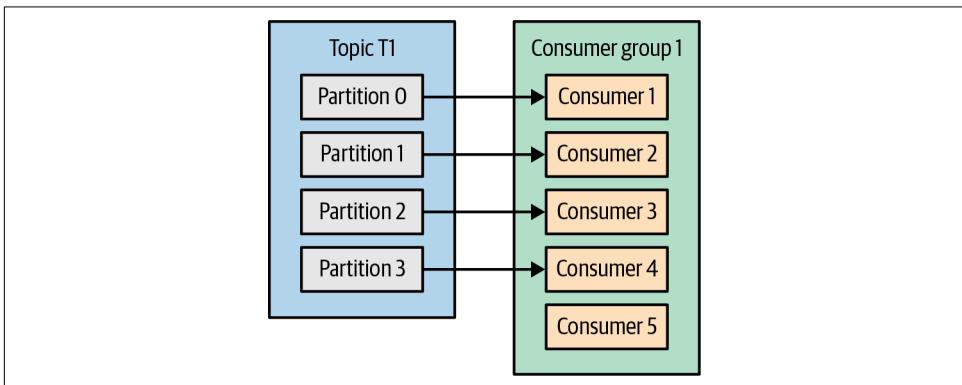


Figure 4-4. More consumers in a group than partitions means idle consumers

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](#).

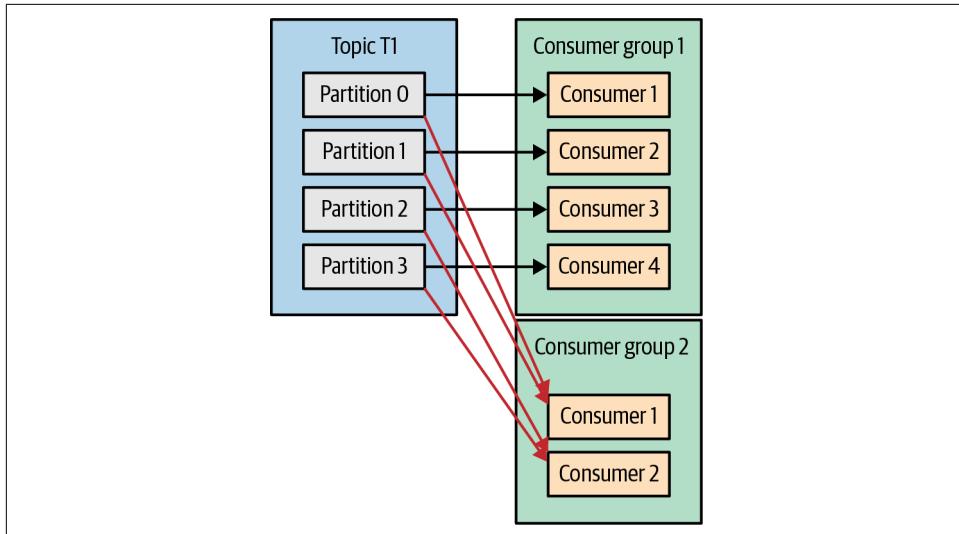


Figure 4-5. Adding a new consumer group, both groups receive all messages

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.

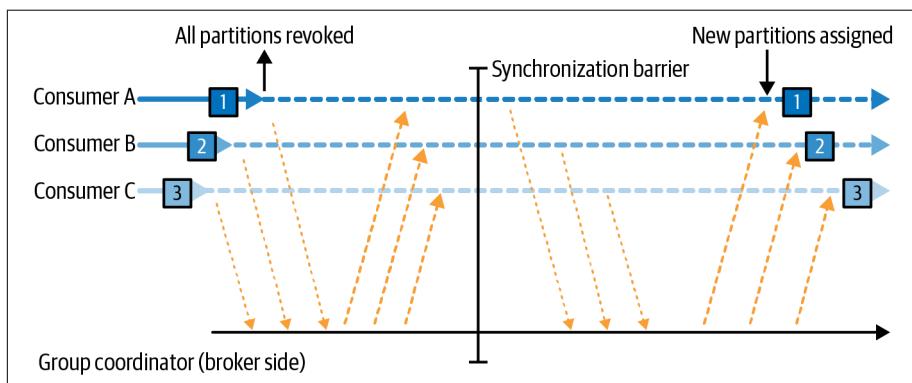


Figure 4-6. Eager rebalance revokes all partitions, pauses consumption, and reassigns them

Cooperative rebalances

Cooperative rebalances (also called *incremental rebalances*) typically involve reassigned 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.

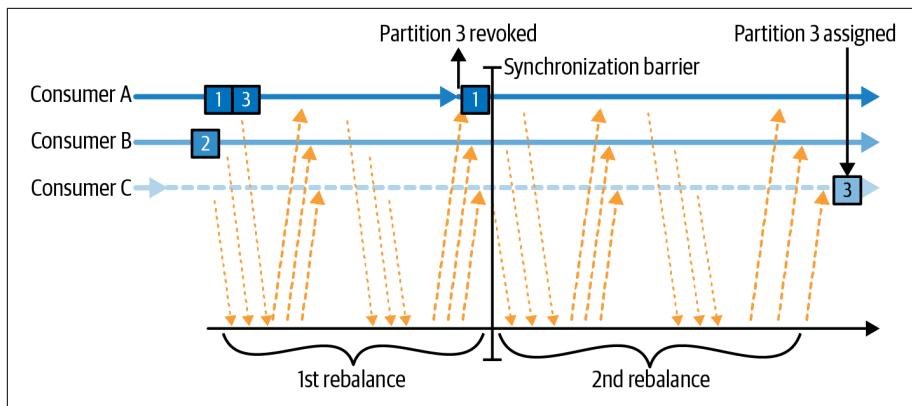


Figure 4-7. Cooperative rebalance only pauses consumption for the subset of partitions that will be reassigned

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 `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")); ①
```

- ① 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) { ①
    ConsumerRecords<String, String> records = consumer.poll(timeout); ②

    for (ConsumerRecord<String, String> record : records) { ③
        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()); ④
    }
}
```

- ① 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.
- ② 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.
- ③ `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.
- ④ 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](#).

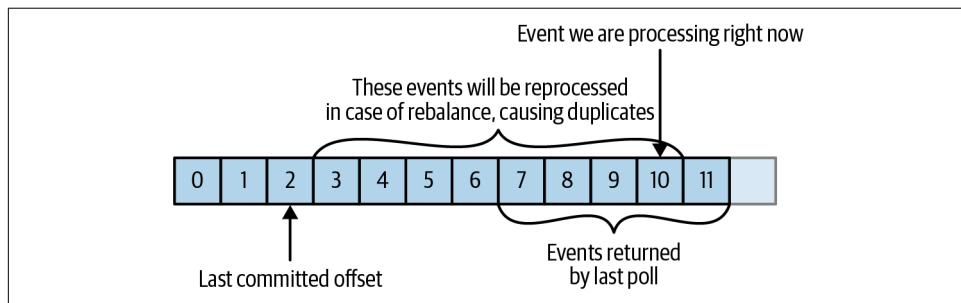


Figure 4-8. Reprocessed messages

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](#).

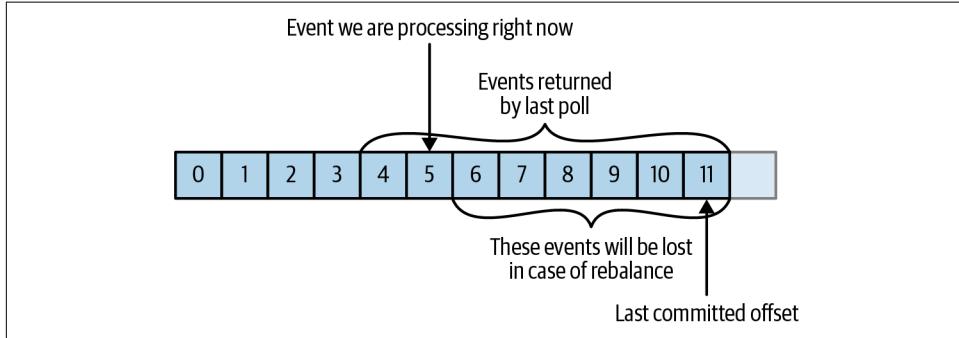


Figure 4-9. Missed messages between offsets

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(); ①
}

```

- ① Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a nonretryable 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);
        }
    }); ①
}

```

- ① 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(); ①
    }
    consumer.commitSync(); ②
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    consumer.close();
}
```

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

- ③ 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.
- ④ 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.
- ⑤ 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()); ④

        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");
        }
    }
}

```

- ➊ We start by implementing a `ConsumerRebalanceListener`.
- ➋ In this example we don't need to do anything when we get a new partition; we'll just start consuming messages.
- ➌ 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.
- ➍ 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)); ①
Map<TopicPartition, OffsetAndTimestamp> offsetMap
    = consumer.offsetsForTimes(partitionTimestampMap); ②
for(Map.Entry<TopicPartition,OffsetAndTimestamp> entry: offsetMap.entrySet()) {
    consumer.seek(entry.getKey(), entry.getValue().offset()); ③
}
```

- ① 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.
- ② 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.
- ③ 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 ③
} finally {
    consumer.close(); ④
    System.out.println("Closed consumer and we are done");
}

```

- ➊ 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.
- ➋ 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.
- ➌ 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.
- ➍ 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> { ①
```

```

@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); ②
    } 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"); ①
props.put("specific.avro.reader","true");
props.put("schema.registry.url", schemaUrl); ②
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); ③

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
            record.value().getName()); ④
    }
    consumer.commitSync();
}

```

- ➊ We use `KafkaAvroDeserializer` to deserialize the Avro messages.
- ➋ `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.
- ➌ We specify the generated class, `Customer`, as the type for the record value.
- ➍ `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"); ①

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ②

    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();
    }
}

```

- ① 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.
- ② 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.

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, `KafkaAdminClient.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 "expedite" 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))); ⑤

// Check that the topic was created correctly:
if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ⑥
    System.out.println("Topic has wrong number of partitions.");
    System.exit(-1);
}
}

```

- ➊ 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.
- ➋ 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.
- ➌ 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.
- ➍ 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.
- ➎ 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.
- ➏ 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 -> { ①  
    String topic = request.getParam("topic"); ②  
    String timeout = request.getParam("timeout");  
    int timeoutMs = NumberUtils.toInt(timeout, 1000);  
  
    DescribeTopicsResult demoTopic = admin.describeTopics( ③  
        Collections.singletonList(topic),  
        new DescribeTopicsOptions().timeoutMs(timeoutMs));  
  
    demoTopic.values().get(topic).whenComplete( ④  
        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()); ⑤
        } else {
            request.response().end(topicDescription.toString()); ⑥
        }
    }
});

}).listen(8080);

```

- ➊ 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.
- ➋ The request includes a topic name as a parameter, and we'll respond with a description of this topic.
- ➌ We call `AdminClient.describeTopics` as usual and get a wrapped `Future` in response.
- ➍ Instead of using the blocking `get()` call, we construct a function that will be called when the `Future` completes.
- ➎ If the `Future` completes with an exception, we send the error to the HTTP client.
- ➏ 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(); ①

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new HashMap<>();

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ②
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> latestOffsets =
    admin.listOffsets(requestLatestOffsets).all().get();

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ③
    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(); ①  
  
Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();  
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:  
    earliestOffsets.entrySet()) {  
    resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ②  
}  
  
try {  
    admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ③  
} 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."); ④  
}
```

- ➊ 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.
- ➋ 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`.
- ➌ After calling `alterConsumerGroupOffsets`, we are waiting on the `Future` to complete so we can see if it completed successfully.
- ➍ 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()); ①
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());
```

- ① 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)); ①
admin.createPartitions(newPartitions).all().get();
```

- ① 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)))); ①
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ②
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ③
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ④

admin.alterPartitionReassignments(reassignment).all().get();

System.out.println("currently reassigning: " +
    admin.listPartitionReassignments().reassignments().get()); ⑤
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Description of demo topic:" + topicDescription); ⑥
```

- ① 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.
- ② 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.
- ③ 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.
- ④ 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.
- ⑤ We can list the ongoing reassignments.
- ⑥ 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 Mock `AdminClient` 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)); ①  
  
    // 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()); ②  
}
```

- ➊ `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.
- ➋ 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()); ①  
}
```

```
@Test
public void testNotTopic() throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ②
}
```

- 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.

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 `Update Metadata` 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 delayed—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.

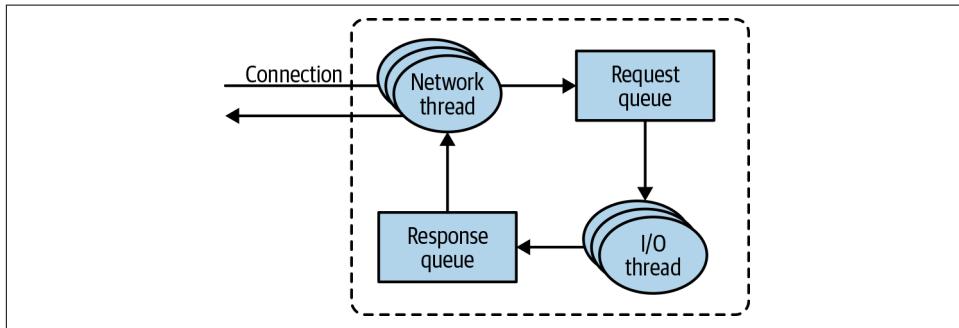


Figure 6-1. Request processing inside Apache Kafka

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 `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.

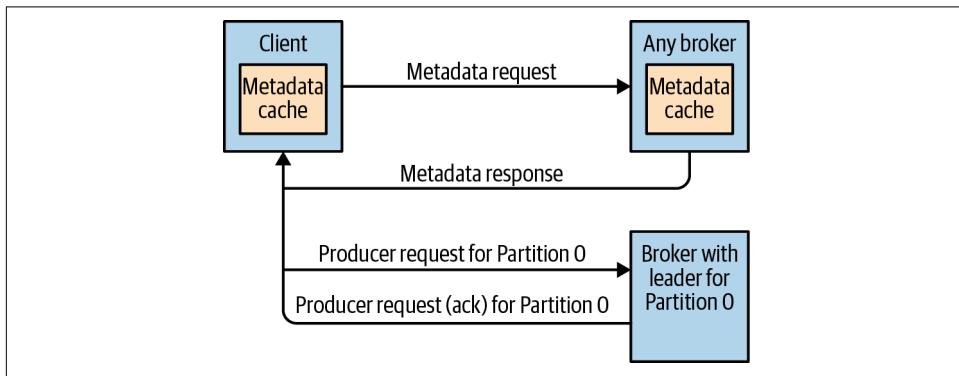


Figure 6-2. Client routing requests

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.

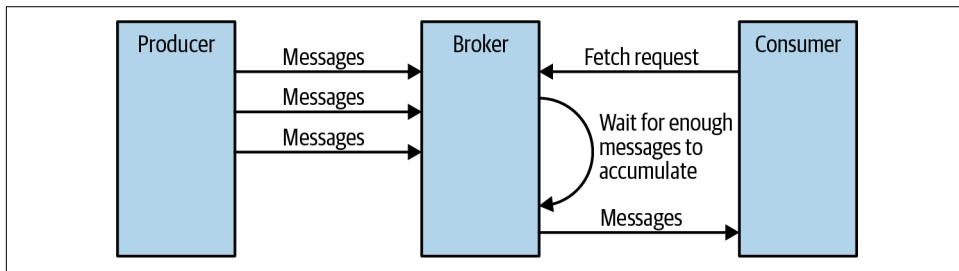


Figure 6-3. Broker delaying response until enough data accumulated

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.

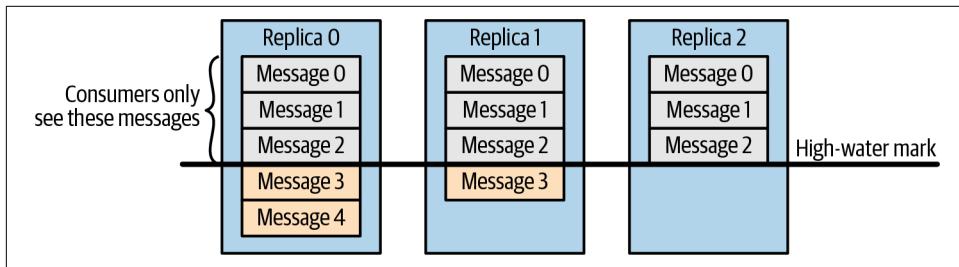


Figure 6-4. Consumers only see messages that were replicated to in-sync replicas

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.

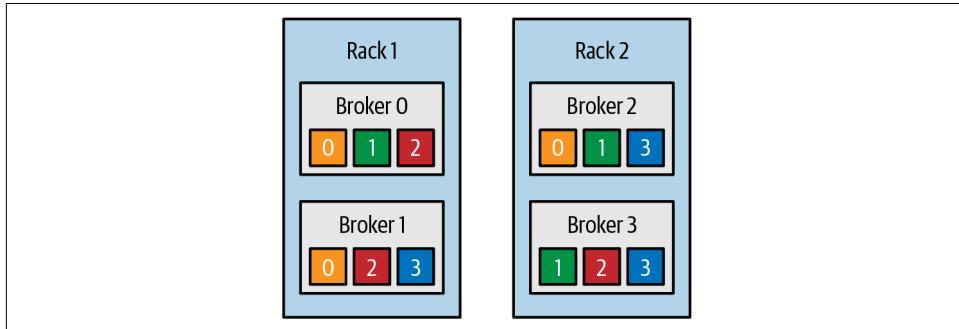


Figure 6-5. Partitions and replicas assigned to brokers on different racks

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.

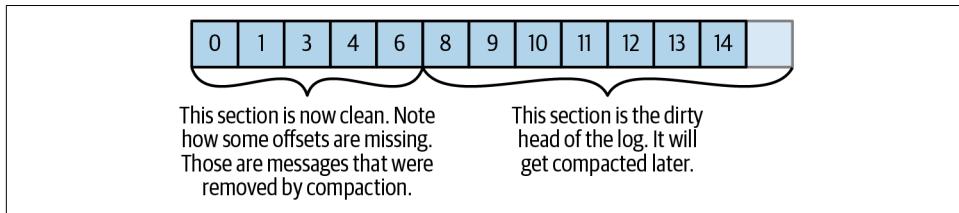


Figure 6-6. Partition with clean and dirty portions

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](#).

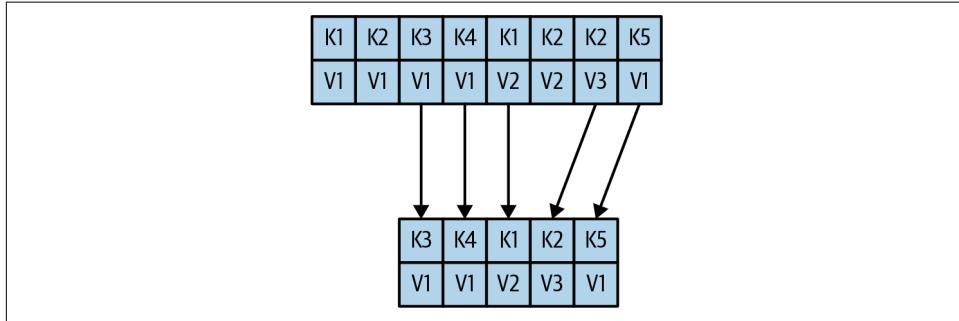


Figure 6-7. Partition segment before and after compaction

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.

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.timeout.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:

- Nonretrievable 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 nonretryable error, a retryable 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 of 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.

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.

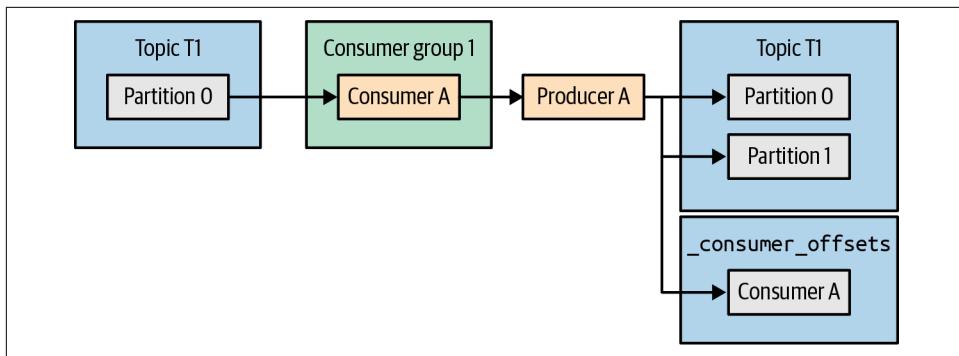


Figure 8-1. Transactional producer with atomic multipartition write

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.

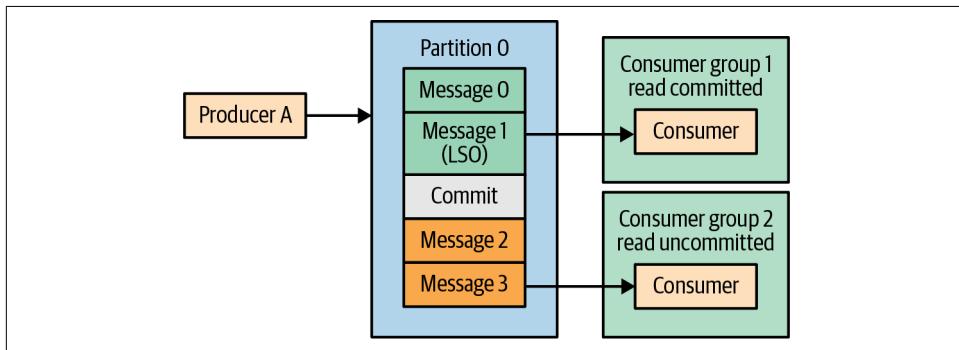


Figure 8-2. Consumers in `read_committed` mode will lag behind consumers with default configuration

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 Mirror-Maker 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(); ⑪
        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.

- ⑧ 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.
- ⑨ 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.
- ⑩ 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.
- ⑪ 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.

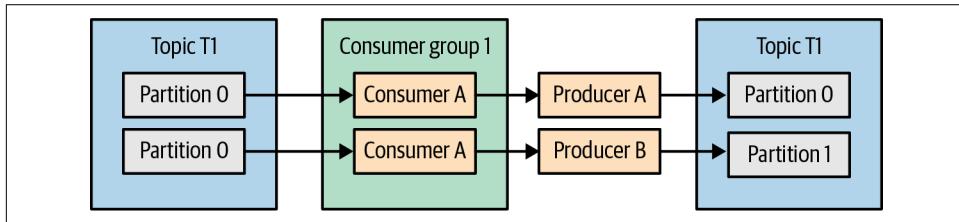


Figure 8-3. Transactional record processor

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.

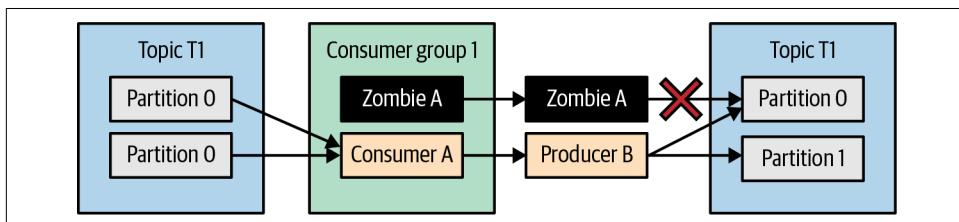


Figure 8-4. Transactional record processor after a rebalance

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.

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 `JscsSchemaConverter`, 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}, "payload": "# Server Basics
#####
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://localhost: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": "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."}}' | 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(JdbcSourceConnector.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" : "gwenwashap",
                "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 archetype](#). 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 Cus-tenborder](#) 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.

Cross-Cluster Data Mirroring

For most of the book we discuss the setup, maintenance, and use of a single Kafka cluster. There are, however, a few scenarios in which an architecture may need more than one cluster.

In some cases, the clusters are completely separated. They belong to different departments or different use cases, and there is no reason to copy data from one cluster to another. Sometimes, different SLAs or workloads make it difficult to tune a single cluster to serve multiple use cases. Other times, there are different security requirements. Those use cases are fairly easy—managing multiple distinct clusters is the same as running a single cluster multiple times.

In other use cases, the different clusters are interdependent, and the administrators need to continuously copy data between the clusters. In most databases, continuously copying data between database servers is called *replication*. Since we've used replication to describe movement of data between Kafka nodes that are part of the same cluster, we'll call copying of data between Kafka clusters *mirroring*. Apache Kafka's built-in cross-cluster replicator is called *MirrorMaker*.

In this chapter, we will discuss cross-cluster mirroring of all or part of the data. We'll start by discussing some of the common use cases for cross-cluster mirroring. Then we'll show a few architectures that are used to implement these use cases and discuss the pros and cons of each architecture pattern. We'll then discuss MirrorMaker itself and how to use it. We'll share operational tips, including deployment and performance tuning. We'll finish by discussing a few alternatives to MirrorMaker.

Use Cases of Cross-Cluster Mirroring

The following is a list of examples of when cross-cluster mirroring would be used:

Regional and central clusters

In some cases, the company has one or more datacenters in different geographical regions, cities, or continents. Each datacenter has its own Kafka cluster. Some applications can work just by communicating with the local cluster, but some applications require data from multiple datacenters (otherwise, you wouldn't be looking at cross-datacenter replication solutions). There are many cases when this is a requirement, but the classic example is a company that modifies prices based on supply and demand. This company can have a datacenter in each city in which it has a presence, collects information about local supply and demand, and adjusts prices accordingly. All this information will then be mirrored to a central cluster where business analysts can run company-wide reports on its revenue.

High availability (HA) and disaster recovery (DR)

The applications run on just one Kafka cluster and don't need data from other locations, but you are concerned about the possibility of the entire cluster becoming unavailable for some reason. For redundancy, you'd like to have a second Kafka cluster with all the data that exists in the first cluster, so in case of emergency you can direct your applications to the second cluster and continue as usual.

Regulatory compliance

Companies operating in different countries may need to use different configurations and policies to conform to legal and regulatory requirements in each country. For instance, some datasets may be stored in separate clusters with strict access control, with subsets of data replicated to other clusters with wider access. To comply with regulatory policies that govern retention period in each region, datasets may be stored in clusters in different regions with different configurations.

Cloud migrations

Many companies these days run their business in both an on-premises datacenter and a cloud provider. Often, applications run on multiple regions of the cloud provider for redundancy, and sometimes multiple cloud providers are used. In these cases, there is often at least one Kafka cluster in each on-premises datacenter and each cloud region. Those Kafka clusters are used by applications in each datacenter and region to transfer data efficiently between the datacenters. For example, if a new application is deployed in the cloud but requires some data that is updated by applications running in the on-premises datacenter and stored in an on-premises database, you can use Kafka Connect to capture database changes to the local Kafka cluster and then mirror these changes to the cloud Kafka

cluster where the new application can use them. This helps control the costs of cross-datacenter traffic as well as improve governance and security of the traffic.

Aggregation of data from edge clusters

Several industries, including retail, telecommunications, transportation, and healthcare, generate data from small devices with limited connectivity. An aggregate cluster with high availability can be used to support analytics and other use cases for data from a large number of edge clusters. This reduces connectivity, availability, and durability requirements on low-footprint edge clusters, for example, in IoT use cases. A highly available aggregate cluster provides business continuity even when edge clusters are offline and simplifies the development of applications that don't have to directly deal with a large number of edge clusters with unstable networks.

Multicuster Architectures

Now that we've seen a few use cases that require multiple Kafka clusters, let's look at some common architectural patterns that we've successfully used when implementing these use cases. Before we go into the architectures, we'll give a brief overview of the realities of cross-datacenter communications. The solutions we'll discuss may seem overly complicated without understanding that they represent trade-offs in the face of specific network conditions.

Some Realities of Cross-Datacenter Communication

The following is a list of some things to consider when it comes to cross-datacenter communication:

High latencies

Latency of communication between two Kafka clusters increases as the distance and the number of network hops between the two clusters increase.

Limited bandwidth

Wide area networks (WANs) typically have far lower available bandwidth than what you'll see inside a single datacenter, and the available bandwidth can vary from minute to minute. In addition, higher latencies make it more challenging to utilize all the available bandwidth.

Higher costs

Regardless of whether you are running Kafka on premise or in the cloud, there are higher costs to communicate between clusters. This is partly because the bandwidth is limited and adding bandwidth can be prohibitively expensive, and also because of the prices vendors charge for transferring data among datacenters, regions, and clouds.

Apache Kafka's brokers and clients were designed, developed, tested, and tuned, all within a single datacenter. We assumed low latency and high bandwidth between brokers and clients. This is apparent in the default timeouts and sizing of various buffers. For this reason, it is not recommended (except in specific cases, which we'll discuss later) to install some Kafka brokers in one datacenter and others in another datacenter.

In most cases, it's best to avoid producing data to a remote datacenter, and when you do, you need to account for higher latency and the potential for more network errors. You can handle the errors by increasing the number of producer retries, and handle the higher latency by increasing the size of the buffers that hold records between attempts to send them.

If we need any kind of replication between clusters, and we ruled out inter-broker communication and producer-broker communication, then we must allow for broker-consumer communication. Indeed, this is the safest form of cross-cluster communication because in the event of network partition that prevents a consumer from reading data, the records remain safe inside the Kafka brokers until communications resume and consumers can read them. There is no risk of accidental data loss due to network partitions. Still, because bandwidth is limited, if there are multiple applications in one datacenter that need to read data from Kafka brokers in another datacenter, we prefer to install a Kafka cluster in each datacenter and mirror the necessary data between them once rather than have multiple applications consume the same data across the WAN.

We'll talk more about tuning Kafka for cross-datacenter communication, but the following principles will guide most of the architectures we'll discuss next:

- No less than one cluster per datacenter.
- Replicate each event exactly once (barring retries due to errors) between each pair of datacenters.
- When possible, consume from a remote datacenter rather than produce to a remote datacenter.

Hub-and-Spoke Architecture

This architecture is intended for the case where there are multiple local Kafka clusters and one central Kafka cluster. See [Figure 10-1](#).

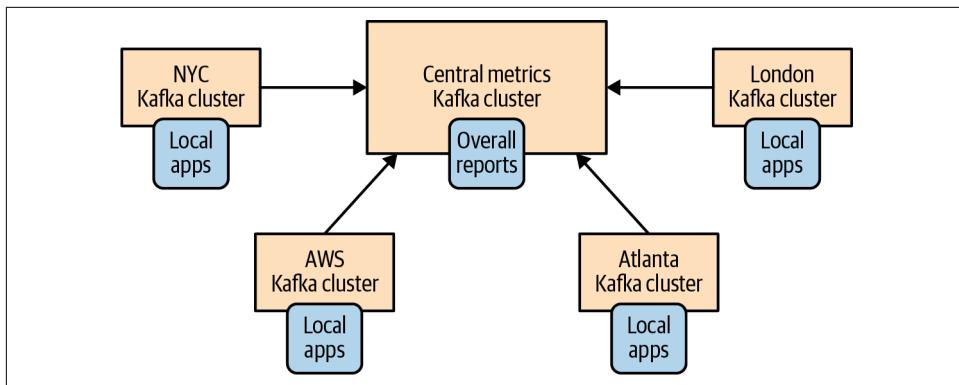


Figure 10-1. The hub-and-spoke architecture

There is also a simpler variation of this architecture with just two clusters: a leader and a follower. See [Figure 10-2](#).

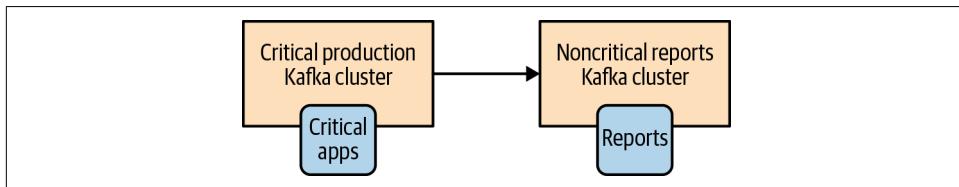


Figure 10-2. A simpler version of the hub-and-spoke architecture

This architecture is used when data is produced in multiple datacenters and some consumers need access to the entire dataset. The architecture also allows for applications in each datacenter to only process data local to that specific datacenter. But it does not give access to the entire dataset from every datacenter.

The main benefit of this architecture is that data is always produced to the local datacenter and events from each datacenter are only mirrored once—to the central datacenter. Applications that process data from a single datacenter can be located at that datacenter. Applications that need to process data from multiple datacenters will be located at the central datacenter where all the events are mirrored. Because replication always goes in one direction and because each consumer always reads from the same cluster, this architecture is simple to deploy, configure, and monitor.

The main drawback of this architecture is the direct result of its benefits and simplicity. Processors in one regional datacenter can't access data in another. To understand better why this is a limitation, let's look at an example of this architecture.

Suppose that we are a large bank and have branches in multiple cities. Let's say that we decide to store user profiles and their account history in a Kafka cluster in each city. We replicate all this information to a central cluster that is used to run the bank's

business analytics. When users connect to the bank website or visit their local branch, they are routed to send events to their local cluster and read events from the same local cluster. However, suppose that a user visits a branch in a different city. Because the user information doesn't exist in the city they are visiting, the branch will be forced to interact with a remote cluster (not recommended) or have no way to access the user's information (really embarrassing). For this reason, use of this pattern is usually limited to only parts of the dataset that can be completely separated between regional datacenters.

When implementing this architecture, for each regional datacenter you need at least one mirroring process on the central datacenter. This process will consume data from each remote regional cluster and produce it to the central cluster. If the same topic exists in multiple datacenters, you can write all the events from this topic to one topic with the same name in the central cluster, or write events from each datacenter to a separate topic.

Active-Active Architecture

This architecture is used when two or more datacenters share some or all of the data, and each datacenter is able to both produce and consume events. See [Figure 10-3](#).

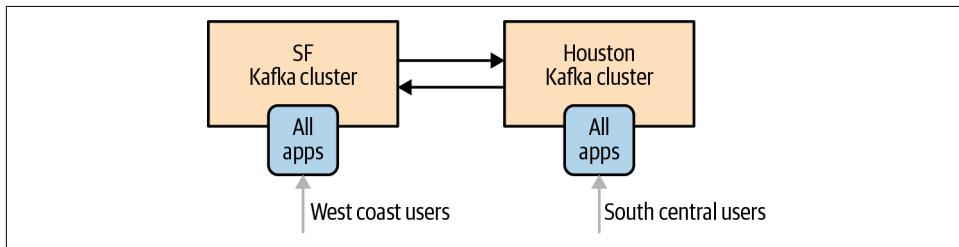


Figure 10-3. The active-active architecture model

The main benefit of this architecture is the ability to serve users from a nearby datacenter, which typically has performance benefits, without sacrificing functionality due to limited availability of data (as we've seen happen in the hub-and-spoke architecture). A secondary benefit is redundancy and resilience. Since every datacenter has all the functionality, if one datacenter is unavailable, you can direct users to a remaining datacenter. This type of failover only requires network redirects of users, typically the easiest and most transparent type of failover.

The main drawback of this architecture is the challenge in avoiding conflicts when data is read and updated asynchronously in multiple locations. This includes technical challenges in mirroring events—for example, how do we make sure the same event isn't mirrored back and forth endlessly? But more importantly, maintaining data consistency between the two datacenters will be difficult. Here are few examples of the difficulties you will encounter:

- If a user sends an event to one datacenter and reads events from another datacenter, it is possible that the event they wrote hasn't arrived at the second datacenter yet. To the user, it will look like they just added a book to their wish list and clicked on the wish list, but the book isn't there. For this reason, when this architecture is used, developers usually find a way to "stick" each user to a specific datacenter and make sure they use the same cluster most of the time (unless they connect from a remote location or the datacenter becomes unavailable).
- An event from one datacenter says the user ordered book A, and an event from more or less the same time at a second datacenter says that the same user ordered book B. After mirroring, both datacenters have both events and thus we can say that each datacenter has two conflicting events. Applications on both datacenters need to know how to deal with this situation. Do we pick one event as the "correct" one? If so, we need consistent rules on how to pick one event so applications on both datacenters will arrive at the same conclusion. Do we decide that both are true and simply send the user two books and have another department deal with returns? Amazon used to resolve conflicts that way, but organizations dealing with stock trades, for example, can't. The specific method for minimizing conflicts and handling them when they occur is specific to each use case. It is important to keep in mind that if you use this architecture, you *will* have conflicts and will need to deal with them.

If you find ways to handle the challenges of asynchronous reads and writes to the same dataset from multiple locations, then this architecture is highly recommended. It is the most scalable, resilient, flexible, and cost-effective option we are aware of. So, it is well worth the effort to figure out solutions for avoiding replication cycles, keeping users mostly in the same datacenter, and handling conflicts when they occur.

Part of the challenge of active-active mirroring, especially with more than two datacenters, is that you will need mirroring tasks for each pair of datacenters and each direction. Many mirroring tools these days can share processes, for example, using the same process for all mirroring to a destination cluster.

In addition, you will want to avoid loops in which the same event is mirrored back and forth endlessly. You can do this by giving each "logical topic" a separate topic for each datacenter and making sure to avoid replicating topics that originated in remote datacenters. For example, logical topic *users* will be topic *SF.users* in one datacenter and *NYC.users* in another datacenter. The mirroring processes will mirror topic *SF.users* from SF to NYC and topic *NYC.users* from NYC to SF. As a result, each event will only be mirrored once, but each datacenter will contain both *SF.users* and *NYC.users*, which means each datacenter will have information for all the users. Consumers will need to consume events from **.users* if they wish to consume all user events. Another way to think of this setup is to see it as a separate namespace for each datacenter that contains all the topics for the specific datacenter. In our example, we'll

have the NYC and the SF namespaces. Some mirroring tools like MirrorMaker prevent replication cycles using a similar naming convention.

Record headers introduced in Apache Kafka in version 0.11.0 enable events to be tagged with their originating datacenter. Header information may also be used to avoid endless mirroring loops and to allow processing events from different datacenters separately. You can also implement this feature by using a structured data format for the record values (Avro is our favorite example) and use this to include tags and headers in the event itself. However, this does require extra effort when mirroring, since none of the existing mirroring tools will support your specific header format.

Active-Standby Architecture

In some cases, the only requirement for multiple clusters is to support some kind of disaster scenario. Perhaps you have two clusters in the same datacenter. You use one cluster for all the applications, but you want a second cluster that contains (almost) all the events in the original cluster that you can use if the original cluster is completely unavailable. Or perhaps you need geographic resiliency. Your entire business is running from a datacenter in California, but you need a second datacenter in Texas that usually doesn't do much and that you can use in case of an earthquake. The Texas datacenter will probably have an inactive ("cold") copy of all the applications that admins can start up in case of emergency and that will use the second cluster ([Figure 10-4](#)). This is often a legal requirement rather than something that the business is actually planning on doing—but you still need to be ready.

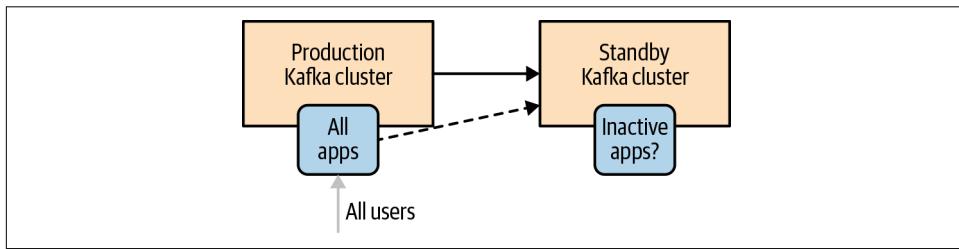


Figure 10-4. The active-standby architecture

The benefits of this setup are simplicity in setup and the fact that it can be used in pretty much any use case. You simply install a second cluster and set up a mirroring process that streams all the events from one cluster to another. No need to worry about access to data, handling conflicts, and other architectural complexities.

The disadvantages are waste of a good cluster and the fact that failover between Kafka clusters is, in fact, much harder than it looks. The bottom line is that it is currently not possible to perform cluster failover in Kafka without either losing data or having duplicate events. Often both. You can minimize them but never fully eliminate them.

It should be obvious that a cluster that does nothing except wait around for a disaster is a waste of resources. Since disasters are (or should be) rare, most of the time we are looking at a cluster of machines that does nothing at all. Some organizations try to fight this issue by having a DR (disaster recovery) cluster that is much smaller than the production cluster. But this is a risky decision because you can't be sure that this minimally sized cluster will hold up during an emergency. Other organizations prefer to make the cluster useful during nondisasters by shifting some read-only workloads to run on the DR cluster, which means they are really running a small version of a hub-and-spoke architecture with a single spoke.

The more serious issue is, how do you failover to a DR cluster in Apache Kafka?

First, it should go without saying that whichever failover method you choose, your SRE team must practice it on a regular basis. A plan that works today may stop working after an upgrade, or perhaps new use cases make the existing tooling obsolete. Once a quarter is usually the bare minimum for failover practices. Strong SRE teams practice far more frequently. Netflix's famous Chaos Monkey, a service that randomly causes disasters, is the extreme—any day may become failover practice day.

Now, let's take a look at what is involved in a failover.

Disaster recovery planning

When planning for disaster recovery, it is important to consider two key metrics. Recovery time objective (RTO) defines the maximum amount of time before all services must resume after a disaster. Recovery point objective (RPO) defines the maximum amount of time for which data may be lost as a result of a disaster. The lower the RTO, the more important it is to avoid manual processes and application restarts, since very low RTO can be achieved only with automated failover. Low RPO requires real-time mirroring with low latencies, and $RPO=0$ requires synchronous replication.

Data loss and inconsistencies in unplanned failover

Because Kafka's various mirroring solutions are all asynchronous (we'll discuss a synchronous solution in the next section), the DR cluster will not have the latest messages from the primary cluster. You should always monitor how far behind the DR cluster is and never let it fall too far behind. But in a busy system you should expect the DR cluster to be a few hundred or even a few thousand messages behind the primary. If your Kafka cluster handles 1 million messages a second and the lag between the primary and the DR cluster is 5 milliseconds, your DR cluster will be 5,000 messages behind the primary in the best-case scenario. So, prepare for unplanned failover to include some data loss. In planned failover, you can stop the primary cluster and wait for the mirroring process to mirror the remaining messages before failing over applications to the DR cluster, thus avoiding this data loss. When unplanned failover occurs and you lose a few thousand messages, note that mirroring solutions currently

don't support transactions, which means that if some events in multiple topics are related to each other (e.g., sales and line items), you can have some events arrive to the DR site in time for the failover and others that don't. Your applications will need to be able to handle a line item without a corresponding sale after you failover to the DR cluster.

Start offset for applications after failover

One of the challenging tasks in failing over to another cluster is making sure applications know where to start consuming data. There are several common approaches. Some are simple but can cause additional data loss or duplicate processing; others are more involved but minimize additional data loss and reprocessing. Let's take a look at a few:

Auto offset reset

Apache Kafka consumers have a configuration for how to behave when they don't have a previously committed offset—they either start reading from the beginning of the partition or from the end of the partition. If you are not somehow mirroring these offsets as part of the DR plan, you need to choose one of these options. Either start reading from the beginning of available data and handle large amounts of duplicates or skip to the end and miss an unknown (and hopefully small) number of events. If your application handles duplicates with no issues, or missing some data is no big deal, this option is by far the easiest. Simply skipping to the end of the topic on failover is a popular failover method due to its simplicity.

Replicate offsets topic

If you are using Kafka consumers from version 0.9.0 and later, the consumers will commit their offsets to a special topic: `__consumer_offsets`. If you mirror this topic to your DR cluster, when consumers start consuming from the DR cluster, they will be able to pick up their old offsets and continue from where they left off. It is simple, but there is a long list of caveats involved.

First, there is no guarantee that offsets in the primary cluster will match those in the secondary cluster. Suppose you only store data in the primary cluster for three days and you start mirroring a topic a week after it was created. In this case, the first offset available in the primary cluster may be offset 57,000,000 (older events were from the first 4 days and were removed already), but the first offset in the DR cluster will be 0. So, a consumer that tries to read offset 57,000,003 (because that's its next event to read) from the DR cluster will fail to do this.

Second, even if you started mirroring immediately when the topic was first created and both the primary and the DR topics start with 0, producer retries can cause offsets to diverge. We discuss an alternative mirroring solution that preserves offsets between primary and DR clusters at the end of this chapter.

Third, even if the offsets were perfectly preserved, because of the lag between primary and DR clusters and because mirroring solutions currently don't support transactions, an offset committed by a Kafka consumer may arrive ahead or behind the record with this offset. A consumer that fails over may find committed offsets without matching records. Or it may find that the latest committed offset in the DR site is older than the latest committed offset in the primary site. See [Figure 10-5](#).

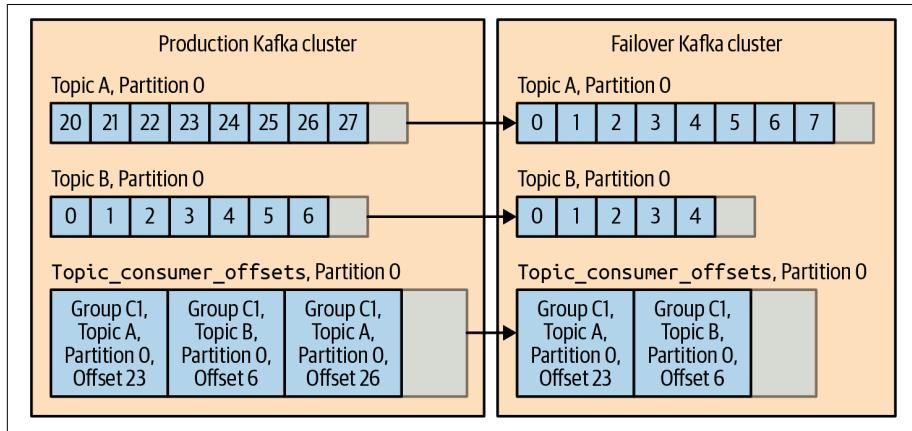


Figure 10-5. A failover causes committed offsets without matching records

In these cases, you need to accept some duplicates if the latest committed offset in the DR site is older than the one committed on the primary or if the offsets in the records in the DR site are ahead of the primary due to retries. You will also need to figure out how to handle cases where the latest committed offset in the DR site doesn't have a matching record—do you start processing from the beginning of the topic or skip to the end?

As you can see, this approach has its limitations. Still, this option lets you failover to another DR with a reduced number of duplicated or missing events compared to other approaches while still being simple to implement.

Time-based failover

From version 0.10.0 onward, each message includes a timestamp indicating the time the message was sent to Kafka. From 0.10.1.0 onward, brokers include an index and an API for looking up offsets by the timestamp. So, if you failover to the DR cluster and you know that your trouble started at 4:05 a.m., you can tell consumers to start processing data from 4:03 a.m. There will be some duplicates from those two minutes, but it is probably better than other alternatives and the behavior is much easier to explain to everyone in the company—"We failed back to 4:03 a.m." sounds better than "We failed back to what may or may not be the

latest committed offsets.” So, this is often a good compromise. The only question is: how do we tell consumers to start processing data from 4:03 a.m.?

One option is to bake it right into your app. Have a user-configurable option to specify the start time for the app. If this is configured, the app can use the new APIs to fetch offset by time, seek to that time, and start consuming from the right point, committing offsets as usual.

This option is great if you wrote all your applications this way in advance. But what if you didn’t? Apache Kafka provides the `kafka-consumer-groups` tool to reset offsets based on a range of options, including timestamp-based reset added in 0.11.0. The consumer group should be stopped while running this type of tool and started immediately after. For example, the following command resets consumer offsets for all topics belonging to a particular group to a specific time:

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-offsets --all-topics --group my-group --to-datetime 2021-03-31T04:03:00.000 --execute
```

This option is recommended in deployments that need to guarantee a level of certainty in their failover.

Offset translation

When discussing mirroring the offsets topic, one of the biggest challenges is the fact that offsets in primary and DR clusters can diverge. In the past, some organizations chose to use an external data store, such as Apache Cassandra, to store mapping of offsets from one cluster to another. Whenever an event is produced to the DR cluster, both offsets are sent to the external data store by the mirroring tool when offsets diverge. These days, mirroring solutions, including Mirror-Maker, use a Kafka topic for storing offset translation metadata. Offsets are stored whenever the difference between the two offsets changes. For example, if offset 495 on the primary mapped to offset 500 on the DR cluster, we’ll record (495,500) in the external store or offset translation topic. If the difference changes later due to duplicates and offset 596 is mapped to 600, then we’ll record the new mapping (596,600). There is no need to store all the offset mappings between 495 and 596; we just assume that the difference remains the same and so offset 550 in the primary cluster will map to 555 in the DR. Then when failover occurs, instead of mapping timestamps (which are always a bit inaccurate) to offsets, we map primary offsets to DR offsets and use those. One of the two techniques listed previously can be used to force consumers to start using the new offsets from the mapping. This still has an issue with offset commits that arrived ahead of the records themselves and offset commits that didn’t get mirrored to the DR on time, but it covers some cases.

After the failover

Let's say that failover was successful. Everything is working just fine on the DR cluster. Now we need to do something with the primary cluster. Perhaps turn it into a DR.

It is tempting to simply modify the mirroring processes to reverse their direction and simply start mirroring from the new primary to the old one. However, this leads to two important questions:

- How do we know where to start mirroring? We need to solve the same problem we have for all our consumers for the mirroring application itself. And remember that all our solutions have cases where they either cause duplicates or miss data—sometimes both.
- In addition, for reasons we discussed previously, it is likely that your original primary will have events that the DR cluster does not. If you just start mirroring new data back, the extra history will remain and the two clusters will be inconsistent.

For this reason, for scenarios where consistency and ordering guarantees are critical, the simplest solution is to first scrape the original cluster—delete all the data and committed offsets—and then start mirroring from the new primary back to what is now the new DR cluster. This gives you a clean slate that is identical to the new primary.

A few words on cluster discovery

One of the important points to consider when planning a standby cluster is that in the event of failover, your applications will need to know how to start communicating with the failover cluster. If you hardcoded the hostnames of your primary cluster brokers in the producer and consumer properties, this will be challenging. Most organizations keep it simple and create a DNS name that usually points to the primary brokers. In case of an emergency, the DNS name can be pointed to the standby cluster. The discovery service (DNS or other) doesn't need to include all the brokers—Kafka clients only need to access a single broker successfully in order to get metadata about the cluster and discover the other brokers. So, including just three brokers is usually fine. Regardless of the discovery method, most failover scenarios do require bouncing consumer applications after failover so they can find the new offsets from which they need to start consuming. For automated failover without application restart to achieve very low RTO, failover logic should be built into client applications.

Stretch Clusters

Active-standby architectures are used to protect the business against the failure of a Kafka cluster by moving applications to communicate with another cluster in case of cluster failure. Stretch clusters are intended to protect the Kafka cluster from failure during a datacenter outage. This is achieved by installing a single Kafka cluster across multiple datacenters.

Stretch clusters are fundamentally different from other multidatacenter scenarios. To start with, they are not multicluster—it is just one cluster. As a result, we don't need a mirroring process to keep two clusters in sync. Kafka's normal replication mechanism is used, as usual, to keep all brokers in the cluster in sync. This setup can include synchronous replication. Producers normally receive an acknowledgment from a Kafka broker after the message was successfully written to Kafka. In the stretch cluster case, we can configure things so the acknowledgment will be sent after the message is written successfully to Kafka brokers in two datacenters. This involves using rack definitions to make sure each partition has replicas in multiple datacenters, and the use of `min.insync.replicas` and `acks=all` to ensure that every write is acknowledged from at least two datacenters. From 2.4.0 onward, brokers can also be configured to enable consumers to fetch from the closest replica using rack definitions. Brokers match their rack with that of the consumer to find the local replica that is most up-to-date, falling back to the leader if a suitable local replica is not available. Consumers fetching from followers in their local datacenter achieve higher throughput, lower latency, and lower cost by reducing cross-datacenter traffic.

The advantages of this architecture are in the synchronous replication—some types of business simply require that their DR site is always 100% synchronized with the primary site. This is often a legal requirement and is applied to any data store across the company—Kafka included. The other advantage is that both datacenters and all brokers in the cluster are used. There is no waste like we saw in active-standby architectures.

This architecture is limited in the type of disasters it protects against. It only protects from datacenter failures, not any kind of application or Kafka failures. The operational complexity is also limited. This architecture demands physical infrastructure that not all companies can provide.

This architecture is feasible if you can install Kafka (and ZooKeeper) in at least three datacenters with high bandwidth and low latency between them. This can be done if your company owns three buildings on the same street, or—more commonly—by using three availability zones inside one region of your cloud provider.

The reason three datacenters are important is because ZooKeeper requires an uneven number of nodes in a cluster and will remain available if a majority of the nodes are available. With two datacenters and an uneven number of nodes, one datacenter will always contain a majority, which means that if this datacenter is unavailable, ZooKeeper is unavailable, and Kafka is unavailable. With three datacenters, you can easily allocate nodes so no single datacenter has a majority. So, if one datacenter is unavailable, a majority of nodes exist in the other two datacenters, and the ZooKeeper cluster will remain available. Therefore, so will the Kafka cluster.



2.5 DC Architecture

A popular model for stretch clusters is a 2.5 DC (datacenter) architecture with both Kafka and ZooKeeper running in two datacenters, and a third “0.5” datacenter with one ZooKeeper node to provide quorum if a datacenter fails.

It is possible to run ZooKeeper and Kafka in two datacenters using a ZooKeeper group configuration that allows for manual failover between two datacenters. However, this setup is uncommon.

Apache Kafka’s MirrorMaker

Apache Kafka contains a tool called MirrorMaker for mirroring data between two datacenters. Early versions of MirrorMaker used a collection of consumers that were members of a consumer group to read data from a set of source topics and a shared Kafka producer in each MirrorMaker process to send those events to the destination cluster. While this was sufficient to mirror data across clusters in some scenarios, it had several issues, particularly latency spikes as configuration changes and addition of new topics resulted in stop-the-world rebalances. MirrorMaker 2.0 is the next-generation multicluster mirroring solution for Apache Kafka that is based on the Kafka Connect framework, overcoming many of the shortcomings of its predecessor. Complex topologies can be easily configured to support a wide range of use cases like disaster recovery, backup, migration, and data aggregation.



More about MirrorMaker

MirrorMaker sounds very simple, but because we were trying to be very efficient and get very close to exactly-once delivery, it turned out to be tricky to implement correctly. MirrorMaker has been rewritten multiple times. The description here and the details in the following sections apply to MirrorMaker 2.0, which was introduced in 2.4.0.

MirrorMaker uses a source connector to consume data from another Kafka cluster rather than from a database. Use of the Kafka Connect framework minimizes administration overhead for busy enterprise IT departments. If you recall the Kafka Connect architecture from [Chapter 9](#), you remember that each connector divides the work among a configurable number of tasks. In MirrorMaker, each task is a consumer and a producer pair. The Connect framework assigns those tasks to different Connect worker nodes as needed—so you may have multiple tasks on one server or have the tasks spread out to multiple servers. This replaces the manual work of figuring out how many MirrorMaker streams should run per instance and how many instances per machine. Connect also has a REST API to centrally manage the configuration for the connectors and tasks. If we assume that most Kafka deployments include Kafka Connect for other reasons (sending database change events into Kafka is a very popular use case), then by running MirrorMaker inside Connect, we can cut down on the number of clusters we need to manage.

MirrorMaker allocates partitions to tasks evenly without using Kafka's consumer group-management protocol to avoid latency spikes due to rebalances when new topics or partitions are added. Events from each partition in the source cluster are mirrored to the same partition in the target cluster, preserving semantic partitioning and maintaining ordering of events for each partition. If new partitions are added to source topics, they are automatically created in the target topic. In addition to data replication, MirrorMaker also supports migration of consumer offsets, topic configuration, and topic ACLs, making it a complete mirroring solution for multicluster deployments. A replication flow defines the configuration of a directional flow from a source cluster to a target cluster. Multiple replication flows can be defined for MirrorMaker to define complex topologies, including the architectural patterns we discussed earlier like hub-and-spoke, active-standby, and active-active architectures.

[Figure 10-6](#) shows the use of MirrorMaker in an active-standby architecture.

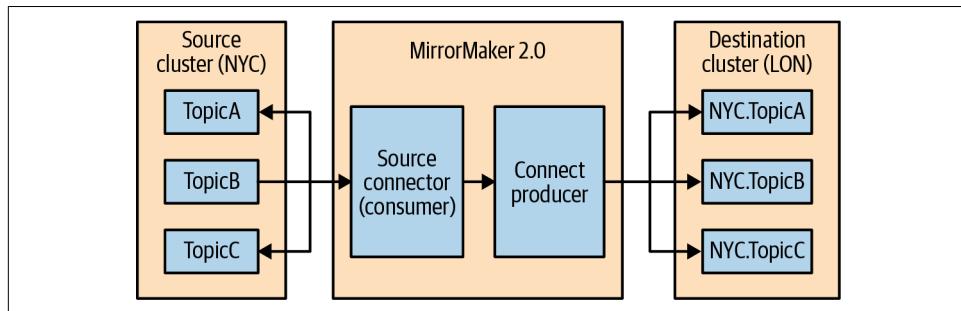


Figure 10-6. The MirrorMaker process in Kafka

Configuring MirrorMaker

MirrorMaker is highly configurable. In addition to the cluster settings to define the topology, Kafka Connect, and connector settings, every configuration property of the underlying producer, consumers, and admin client used by MirrorMaker can be customized. We will show a few examples here and highlight some of the important configuration options, but exhaustive documentation of MirrorMaker is outside our scope.

With that in mind, let's take a look at a MirrorMaker example. The following command starts MirrorMaker with the configuration options specified in the properties file:

```
bin/connect-mirror-maker.sh etc/kafka/connect-mirror-maker.properties
```

Let's look at some of MirrorMaker's configuration options:

Replication flow

The following example shows the configuration options for setting up an active-standby replication flow between two datacenters in New York and London:

```
clusters = NYC, LON  
NYC.bootstrap.servers = kafka.nyc.example.com:9092  
LON.bootstrap.servers = kafka.lon.example.com:9092  
NYC->LON.enabled = true  
NYC->LON.topics = .*
```

- ➊ Define aliases for the clusters used in replication flows.
- ➋ Configure bootstrap for each cluster, using the cluster alias as the prefix.
- ➌ Enable replication flow between a pair of clusters using the prefix `source ->target`. All configuration options for this flow use the same prefix.
- ➍ Configure the topics to be mirrored for this replication flow.

Mirror topics

As shown in the example, for each replication flow, a regular expression may be specified for the topic names that will be mirrored. In this example, we chose to replicate every topic, but it is often good practice to use something like `prod.*` and avoid replicating test topics. A separate topic exclusion list containing topic names or patterns like `test.*` may also be specified to exclude topics that don't require mirroring. Target topic names are automatically prefixed with the source cluster alias by default. For example, in active-active architecture, MirrorMaker replicating topics from an NYC datacenter to a LON datacenter will mirror the topic `orders` from NYC to the topic `NYC.orders` in LON. This default naming strategy prevents replication cycles resulting in events being endlessly mirrored

between the two clusters in active-active mode if topics are mirrored from NYC to LON as well as LON to NYC. The distinction between local and remote topics also supports aggregation use cases since consumers may choose subscription patterns to consume data produced from just the local region or subscribe to topics from all regions to get the complete dataset.

MirrorMaker periodically checks for new topics in the source cluster and starts mirroring these topics automatically if they match the configured patterns. If more partitions are added to the source topic, the same number of partitions is automatically added to the target topic, ensuring that events in the source topic appear in the same partitions in the same order in the target topic.

Consumer offset migration

MirrorMaker contains a utility class `RemoteClusterUtils` to enable consumers to seek to the last checkpointed offset in a DR cluster with offset translation when failing over from a primary cluster. Support for periodic migration of consumer offsets was added in 2.7.0 to automatically commit translated offsets to the target `_consumer_offsets` topic so that consumers switching to a DR cluster can restart from where they left off in the primary cluster with no data loss and minimal duplicate processing. Consumer groups for which offsets are migrated can be customized, and for added protection, MirrorMaker does not overwrite offsets if consumers on the target cluster are actively using the target consumer group, thus avoiding any accidental conflicts.

Topic configuration and ACL migration

In addition to mirroring data records, MirrorMaker may be configured to mirror topic configuration and access control lists (ACLs) of the topics to retain the same behavior for the mirrored topic. The default configuration enables this migration with reasonable periodic refresh intervals that may be sufficient in most cases. Most of the topic configuration settings from the source are applied to the target topic, but a few like `min.insync.replicas` are not applied by default. The list of excluded configs can be customized.

Only literal topic ACLs that match topics being mirrored are migrated, so if you are using prefixed or wildcard ACLs or alternative authorization mechanisms, you will need to configure those on the target cluster explicitly. ACLs for `Topic:Write` are not migrated to ensure that only MirrorMaker is allowed to write to the target topic. Appropriate access must be explicitly granted at the time of failover to ensure that applications work with the secondary cluster.

Connector tasks

The configuration option `tasks.max` limits the maximum number of tasks that the connector associated with MirrorMaker may use. The default is 1, but a

minimum of 2 is recommended. When replicating a lot of topic partitions, higher values should be used if possible to increase parallelism.

Configuration prefixes

MirrorMaker supports customization of configuration options for all its components, including connectors, producers, consumers, and admin clients. Kafka Connect and connector configs can be specified without any prefix. But since MirrorMaker configuration can include configuration for multiple clusters, prefixes can be used to specify cluster-specific configs or configs for a particular replication flow. As we saw in the example earlier, clusters are identified using aliases that are used as a configuration prefix for options related to that cluster. Prefixes can be used to build a hierarchical configuration, with the more specific prefixed configuration having higher precedence than the less specific or nonprefixed configuration. MirrorMaker uses the following prefixes:

- {cluster}.{connector_config}
- {cluster}.admin.{admin_config}
- {source_cluster}.consumer.{consumer_config}
- {target_cluster}.producer.{producer_config}
- {source_cluster}->{target_cluster}.{replication_flow_config}

Multicloud Replication Topology

We have seen an example configuration for a simple active-standby replication flow for MirrorMaker. Now let's look at extending the configuration to support other common architectural patterns.

Active-active topology between New York and London can be configured by enabling replication flow in both directions. In this case, even though all topics from NYC are mirrored to LON and vice versa, MirrorMaker ensures that the same event isn't constantly mirrored back and forth between the pair of clusters since remote topics use the cluster alias as the prefix. It is good practice to use the same configuration file that contains the full replication topology for different MirrorMaker processes since it avoids conflicts when configs are shared using the internal configs topic in the target datacenter. MirrorMaker processes can be started in the target datacenter using the shared configuration file by specifying the target cluster when starting the Mirror-Maker process using the `--clusters` option:

```
clusters = NYC, LON
NYC.bootstrap.servers = kafka.nyc.example.com:9092
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true
NYC->LON.topics = .*
LON->NYC.enabled = true
LON->NYC.topics = .*
```

①
②
③
④

- ① Enable replication from New York to London.
- ② Specify topics that are replicated from New York to London.
- ③ Enable replication from London to New York.
- ④ Specify topics that are replicated from London to New York.

More replication flows with additional source or target clusters can also be added to the topology. For example, we can extend the configuration to support the fan out from NYC to SF and LON by adding a new replication flow for SF:

```
clusters = NYC, LON, SF
SF.bootstrap.servers = kafka.sf.example.com:9092
NYC->SF.enabled = true
NYC->SF.topics = .*
```

Securing MirrorMaker

For production clusters, it is important to ensure that all cross-datacenter traffic is secure. Options for securing Kafka clusters are described in [Chapter 11](#). MirrorMaker must be configured to use a secure broker listener in both source and target clusters, and client-side security options for each cluster must be configured for MirrorMaker to enable it to establish authenticated connections. SSL should be used to encrypt all cross-datacenter traffic. For example, the following configuration may be used to configure credentials for MirrorMaker:

```
NYC.security.protocol=SASL_SSL ❶
NYC.sasl.mechanism=PLAIN
NYC.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required username="MirrorMaker" password="MirrorMaker-password"; ❷
```

- ❶ Security protocol should match that of the broker listener corresponding to the bootstrap servers specified for the cluster. SSL or SASL_SSL is recommended.
- ❷ Credentials for MirrorMaker are specified here using JAAS configuration since SASL is used. For SSL, keystores should be specified if mutual client authentication is enabled.

The principal associated with MirrorMaker must also be granted appropriate permissions on the source and target clusters if authorization is enabled on the clusters. ACLs must be granted for the MirrorMaker process for:

- **Topic:Read** on the source cluster to consume from source topics; **Topic:Create** and **Topic:Write** on the target cluster to create and produce to target topics.

- `Topic:DescribeConfigs` on the source cluster to obtain source topic configuration; `Topic:AlterConfigs` on the target cluster to update target topic configuration.
- `Topic:Alter` on the target cluster to add partitions if new source partitions are detected.
- `Group:Describe` on the source cluster to obtain source consumer group metadata, including offsets; `Group:Read` on the target cluster to commit offsets for those groups in the target cluster.
- `Cluster:Describe` on the source cluster to obtain source topic ACLs; `Cluster:Alter` on the target cluster to update the target topic ACLs.
- `Topic:Create` and `Topic:Write` permissions for internal MirrorMaker topics in the source and target clusters.

Deploying MirrorMaker in Production

In the previous example, we started MirrorMaker in dedicated mode on the command line. You can start any number of these processes to form a dedicated MirrorMaker cluster that is scalable and fault-tolerant. The processes mirroring to the same cluster will find each other and balance load between them automatically. Usually when running MirrorMaker in a production environment, you will want to run MirrorMaker as a service, running in the background with `nohup` and redirecting its console output to a log file. The tool also has `-daemon` as a command-line option that should do that for you. Most companies that use MirrorMaker have their own startup scripts that also include the configuration parameters they use. Production deployment systems like Ansible, Puppet, Chef, and Salt are often used to automate deployment and manage the many configuration options. MirrorMaker may also be run inside a Docker container. MirrorMaker is completely stateless and doesn't require any disk storage (all the data and state are stored in Kafka itself).

Since MirrorMaker is based on Kafka Connect, all deployment modes of Connect can be used with MirrorMaker. Standalone mode may be used for development and testing where MirrorMaker runs as a standalone Connect worker on a single machine. MirrorMaker may also be run as a connector in an existing distributed Connect cluster by explicitly configuring the connectors. For production use, we recommend running MirrorMaker in distributed mode either as a dedicated MirrorMaker cluster or in a shared distributed Connect cluster.

If at all possible, run MirrorMaker at the target datacenter. So, if you are sending data from NYC to SF, MirrorMaker should run in SF and consume data across the US from NYC. The reason for this is that long-distance networks can be a bit less reliable than those inside a datacenter. If there is a network partition and you lose connectivity between the datacenters, having a consumer that is unable to connect to a cluster

is much safer than a producer that can't connect. If the consumer can't connect, it simply won't be able to read events, but the events will still be stored in the source Kafka cluster and can remain there for a long time. There is no risk of losing events. On the other hand, if the events were already consumed and MirrorMaker can't produce them due to network partition, there is always a risk that these events will accidentally get lost by MirrorMaker. So, remote consuming is safer than remote producing.

When do you have to consume locally and produce remotely? The answer is when you need to encrypt the data while it is transferred between the datacenters but you don't need to encrypt the data inside the datacenter. Consumers take a significant performance hit when connecting to Kafka with SSL encryption—much more so than producers. This is because use of SSL requires copying data for encryption, which means consumers no longer enjoy the performance benefits of the usual zero-copy optimization. And this performance hit also affects the Kafka brokers themselves. If your cross datacenter traffic requires encryption, but local traffic does not, then you may be better off placing MirrorMaker at the source datacenter, having it consume unencrypted data locally, and then producing it to the remote datacenter through an SSL encrypted connection. This way, the producer connects to Kafka with SSL but not the consumer, which doesn't impact performance as much. If you use this consume locally and produce remotely approach, make sure MirrorMaker's Connect producer is configured to never lose events by configuring it with `acks=all` and a sufficient number of retries. Also, configure MirrorMaker to fail fast using `errors.tolerance=none` when it fails to send events, which is typically safer to do than to continue and risk data loss. Note that newer versions of Java have significantly increased SSL performance, so producing locally and consuming remotely may be a viable option even with encryption.

Another case where we may need to produce remotely and consume locally is a hybrid scenario when mirroring from an on-premises cluster to a cloud cluster. Secure on-premises clusters are likely to be behind a firewall that doesn't allow incoming connections from the cloud. Running MirrorMaker on premise allows all connections to be from on premises to the cloud.

When deploying MirrorMaker in production, it is important to remember to monitor it as follows:

Kafka Connect monitoring

Kafka Connect provides a wide range of metrics to monitor different aspects like connector metrics to monitor connector status, source connector metrics to monitor throughout, and worker metrics to monitor rebalance delays. Connect also provides a REST API to view and manage connectors.

MirrorMaker metrics monitoring

In addition to metrics from Connect, MirrorMaker adds metrics to monitor mirroring throughput and replication latency. The replication latency metric `replication-latency-ms` shows the time interval between the record timestamp and the time at which the record was successfully produced to the target cluster. This is useful to detect if the target is not keeping up with the source in a timely manner. Increased latency during peak hours may be OK if there is sufficient capacity to catch up later, but sustained increase in latency may indicate insufficient capacity. Other metrics like `record-age-ms`, which shows the age of records at the time of replication, `byte-rate`, which shows replication throughout, and `checkpoint-latency-ms`, which shows offset migration latency, can also be very useful. MirrorMaker also emits periodic heartbeats by default, which can be used to monitor its health.

Lag monitoring

You will definitely want to know if the target cluster is falling behind the source. The lag is the difference in offsets between the latest message in the source Kafka cluster and the latest message in the target cluster. See [Figure 10-7](#).

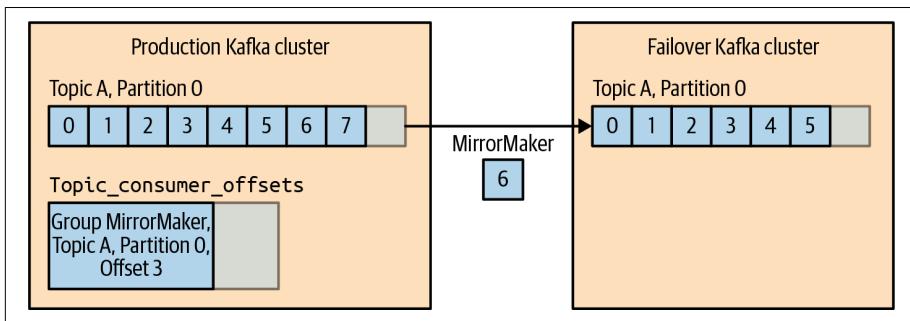


Figure 10-7. Monitoring the lag difference in offsets

In [Figure 10-7](#), the last offset in the source cluster is 7, and the last offset in the target is 5—meaning there is a lag of 2 messages.

There are two ways to track this lag, and neither is perfect:

- Check the latest offset committed by MirrorMaker to the source Kafka cluster. You can use the `kafka-consumer-groups` tool to check for each partition MirrorMaker is reading—the offset of the last event in the partition, the last offset MirrorMaker committed, and the lag between them. This indicator is not 100% accurate because MirrorMaker doesn't commit offsets all the time. It commits offsets every minute by default, so you will see the lag grow for a minute and then suddenly drop. In the diagram, the real lag is 2, but the `kafka-consumer-groups` tool will report a lag of 5 because MirrorMaker

hasn't committed offsets for more recent messages yet. LinkedIn's Burrow monitors the same information but has a more sophisticated method to determine whether the lag represents a real problem, so you won't get false alerts.

- Check the latest offset read by MirrorMaker (even if it isn't committed). The consumers embedded in MirrorMaker publish key metrics in JMX. One of them is the consumer maximum lag (over all the partitions it is consuming). This lag is also not 100% accurate because it is updated based on what the consumer read but doesn't take into account whether the producer managed to send those messages to the destination Kafka cluster and whether they were acknowledged successfully. In this example, the MirrorMaker consumer will report a lag of 1 message rather than 2, because it already read message 6—even though the message wasn't produced to the destination yet.

Note that if MirrorMaker skips or drops messages, neither method will detect an issue because they just track the latest offset. [Confluent Control Center](#) is a commercial tool that monitors message counts and checksums and closes this monitoring gap.

Producer and consumer metrics monitoring

The Kafka Connect framework used by MirrorMaker contains a producer and a consumer. Both have many available metrics, and we recommend collecting and tracking them. The [Kafka documentation](#) lists all the available metrics. Here are a few metrics that are useful in tuning MirrorMaker performance:

Consumer

`fetch-size-avg`, `fetch-size-max`, `fetch-rate`, `fetch-throttle-time-avg`, and `fetch-throttle-time-max`

Producer

`batch-size-avg`, `batch-size-max`, `requests-in-flight`, and `record-retry-rate`

Both

`io-ratio` and `io-wait-ratio`

Canary

If you monitor everything else, a canary isn't strictly necessary, but we like to add it in for multiple layers of monitoring. It provides a process that, every minute, sends an event to a special topic in the source cluster and tries to read the event from the destination cluster. It also alerts you if the event takes more than an acceptable amount of time to arrive. This can mean that MirrorMaker is lagging or that it isn't available at all.

Tuning MirrorMaker

MirrorMaker is horizontally scalable. Sizing of the MirrorMaker cluster depends on the throughput you need and the lag you can tolerate. If you can't tolerate any lag, you have to size MirrorMaker with enough capacity to keep up with your top throughput. If you can tolerate some lag, you can size MirrorMaker to be 75–80% utilized 95–99% of the time. Then, expect some lag to develop when you are at peak throughput. Because MirrorMaker has spare capacity most of the time, it will catch up once the peak is over.

Then you want to measure the throughput you get from MirrorMaker with a different number of connector tasks—configured with the `tasks.max` parameter. This depends a lot on your hardware, datacenter, or cloud provider, so you will want to run your own tests. Kafka ships with the `kafka-performance-producer` tool. Use it to generate load on a source cluster and then connect MirrorMaker and start mirroring this load. Test MirrorMaker with 1, 2, 4, 8, 16, 24, and 32 tasks. Watch where performance tapers off and set `tasks.max` just below this point. If you are consuming or producing compressed events (recommended, since bandwidth is the main bottleneck for cross-datacenter mirroring), MirrorMaker will have to decompress and recompress the events. This uses a lot of CPU, so keep an eye on CPU utilization as you increase the number of tasks. Using this process, you will find the maximum throughput you can get with a single MirrorMaker worker. If it is not enough, you will want to experiment with additional workers. If you are running MirrorMaker on an existing Connect cluster with other connectors, make sure you also take the load from those connectors into account when sizing the cluster.

In addition, you may want to separate sensitive topics—those that absolutely require low latency and where the mirror must be as close to the source as possible—to a separate MirrorMaker cluster. This will prevent a bloated topic or an out-of-control producer from slowing down your most sensitive data pipeline.

This is pretty much all the tuning you can do to MirrorMaker itself. However, you can still increase the throughput of each task and each MirrorMaker worker.

If you are running MirrorMaker across datacenters, tuning the TCP stack can help to increase the effective bandwidth. In Chapters 3 and 4, we saw that TCP buffer sizes can be configured for producers and consumers using `send.buffer.bytes` and `receive.buffer.bytes`. Similarly, broker-side buffer sizes can be configured using `socket.send.buffer.bytes` and `socket.receive.buffer.bytes` on brokers. These configuration options should be combined with optimization of the network configuration in Linux, as follows:

- Increase the TCP buffer size (`net.core.rmem_default`, `net.core.rmem_max`, `net.core.wmem_default`, `net.core.wmem_max`, and `net.core.optmem_max`)
- Enable automatic window scaling (`sysctl -w net.ipv4.tcp_window_scaling=1` or add `net.ipv4.tcp_window_scaling=1` to `/etc/sysctl.conf`)
- Reduce the TCP slow start time (set `/proc/sys/net/ipv4/tcp_slow_start_after_idle` to 0)

Note that tuning the Linux network is a large and complex topic. To understand more about these parameters and others, we recommend reading a network tuning guide such as *Performance Tuning for Linux Servers* by Sandra K. Johnson et al. (IBM Press).

In addition, you may want to tune the underlying producers and consumers of MirrorMaker. First, you will want decide whether the producer or the consumer is the bottleneck—is the producer waiting for the consumer to bring more data or the other way around? One way to decide is to look at the producer and consumer metrics you are monitoring. If one process is idle while the other is fully utilized, you know which one needs tuning. Another method is to do several thread dumps (using `jstack`) and see if the MirrorMaker threads are spending most of the time in poll or in send—more time spent polling usually means the consumer is the bottleneck, while more time spent sending shift points to the producer.

If you need to tune the producer, the following configuration settings can be useful:

`linger.ms` and `batch.size`

If your monitoring shows that the producer consistently sends partially empty batches (i.e., `batch-size-avg` and `batch-size-max` metrics are lower than configured `batch.size`), you can increase throughput by introducing a bit of latency. Increase `linger.ms` and the producer will wait a few milliseconds for the batches to fill up before sending them. If you are sending full batches and have memory to spare, you can increase `batch.size` and send larger batches.

`max.in.flight.requests.per.connection`

Limiting the number of in-flight requests to 1 is currently the only way for MirrorMaker to guarantee that message ordering is preserved if some messages require multiple retries before they are successfully acknowledged. But this means every request that was sent by the producer has to be acknowledged by the target cluster before the next message is sent. This can limit throughput, especially if there is significant latency before the brokers acknowledge the messages. If message order is not critical for your use case, using the default value of 5 for `max.in.flight.requests.per.connection` can significantly increase your throughput.

The following consumer configurations can increase throughput for the consumer:

`fetch.max.bytes`

If the metrics you are collecting show that `fetch-size-avg` and `fetch-size-max` are close to the `fetch.max.bytes` configuration, the consumer is reading as much data from the broker as it is allowed. If you have available memory, try increasing `fetch.max.bytes` to allow the consumer to read more data in each request.

`fetch.min.bytes` and `fetch.max.wait.ms`

If you see in the consumer metrics that `fetch-rate` is high, the consumer is sending too many requests to the brokers and not receiving enough data in each request. Try increasing both `fetch.min.bytes` and `fetch.max.wait.ms` so the consumer will receive more data in each request and the broker will wait until enough data is available before responding to the consumer request.

Other Cross-Cluster Mirroring Solutions

We looked in depth at MirrorMaker because this mirroring software arrives as part of Apache Kafka. However, MirrorMaker also has some limitations when used in practice. It is worthwhile to look at some of the alternatives to MirrorMaker and the ways they address MirrorMaker limitations and complexities. We describe a couple of open source solutions from Uber and LinkedIn and commercial solutions from Confluent.

Uber uReplicator

Uber ran legacy MirrorMaker at very large scale, and as the number of topics and partitions grew and the cluster throughput increased, it started running into several problems. As we saw earlier, the legacy MirrorMaker used consumers that were members of a single consumer group to consume from source topics. Adding MirrorMaker threads, adding MirrorMaker instances, bouncing MirrorMaker instances, or even adding new topics that match the regular expression used in the inclusion filter all caused consumers to rebalance. As we saw in [Chapter 4](#), rebalancing stops all the consumers until new partitions can be assigned to each consumer. With a very large number of topics and partitions, this can take a while. This is especially true when using old consumers like Uber did. In some cases, this caused 5–10 minutes of inactivity, causing mirroring to fall behind and accumulate a large backlog of events to mirror, which can take a long time to recover from. This caused very high latency for consumers reading events from the destination cluster. To avoid rebalances when someone added a topic matching the topic inclusion filter, Uber decided to maintain a list of exact topic names to mirror instead of using a regular expression filter. But this was hard to maintain as all MirrorMaker instances had to be reconfigured and bounced to add a new topic. If not done correctly, this could result in endless rebalances as the consumers won't be able to agree on the topics they subscribe to.

Given these issues, Uber decided to write its own MirrorMaker clone, called uReplicator. Uber decided to use Apache Helix as a central (but highly available) controller to manage the topic list and the partitions assigned to each uReplicator instance. Administrators use a REST API to add new topics to the list in Helix, and uReplicator is responsible for assigning partitions to the different consumers. To achieve this, Uber replaced the Kafka consumers used in MirrorMaker with a Kafka consumer Uber engineers wrote called Helix consumer. This consumer takes its partition assignment from the Apache Helix controller rather than as a result of an agreement between the consumers (see [Chapter 4](#) for details on how this is done in Kafka). As a result, the Helix consumer can avoid rebalances and instead listen to changes in the assigned partitions that arrive from Helix.

Uber engineering wrote a [blog post](#) describing the architecture in more detail and showing the improvements they experienced. uReplicator's dependency on Apache Helix introduces a new component to learn and manage, adding complexity to any deployment. As we saw earlier, MirrorMaker 2.0 solves many of these scalability and fault-tolerance issues of legacy MirrorMaker without any external dependencies.

LinkedIn Brooklin

Like Uber, LinkedIn was also using legacy MirrorMaker for transferring data between Kafka clusters. As the scale of the data grew, it also ran into similar scalability issues and operational challenges. So LinkedIn built a mirroring solution on top of its data streaming system called Brooklin. Brooklin is a distributed service that can stream data between different heterogeneous data source and target systems, including Kafka. As a generic data ingestion framework that can be used to build data pipelines, Brooklin supports multiple use cases:

- Data bridge to feed data into stream processing systems from different data sources
- Stream change data capture (CDC) events from different data stores
- Cross-cluster mirroring solution for Kafka

Brooklin is a scalable distributed system designed for high reliability and has been tested with Kafka at scale. It is used to mirror trillions of messages a day and has been optimized for stability, performance, and operability. Brooklin comes with a REST API for management operations. It is a shared service that can process a large number of data pipelines, enabling the same service to mirror data across multiple Kafka clusters.

Confluent Cross-Datacenter Mirroring Solutions

At the same time that Uber developed its uReplicator, Confluent independently developed Confluent Replicator. Despite the similarities in names, the projects have almost nothing in common—they are different solutions to two different sets of MirrorMaker problems. Like MirrorMaker 2.0, which came later, Confluent's Replicator is based on the Kafka Connect framework and was developed to address issues its enterprise customers encountered when using legacy MirrorMaker to manage their multicloud deployments.

For customers who use stretch clusters for their operational simplicity and low RTO and RPO, Confluent added Multi-Region Cluster (MRC) as a built-in feature of Confluent Server, which is a commercial component of the Confluent Platform. MRC extends Kafka's support for stretch clusters using asynchronous replicas to limit impact on latency and throughput. Like stretch clusters, this is suitable for replication between availability zones or regions with latencies less than 50 ms and benefits from transparent client failover. For distant clusters with less reliable networks, a new built-in feature called Cluster Linking was added to Confluent Server more recently. Cluster Linking extends Kafka's offset-preserving intra-cluster replication protocol to mirror data between clusters.

Let's look at the features supported by each of these solutions:

Confluent Replicator

Confluent Replicator is a mirroring tool similar to MirrorMaker that relies on the Kafka Connect framework for cluster management and can run on existing Connect clusters. Both support data replication for different topologies as well as migration of consumer offsets and topic configuration. There are some differences in features between the two. For example, MirrorMaker supports ACL migration and offset translation for any client, but Replicator doesn't migrate ACLs and supports offset translation (using timestamp interceptor) only for Java clients. Replicator doesn't have the concept of local and remote topics like MirrorMaker, but it supports aggregate topics. Like MirrorMaker, Replicator also avoids replication cycles but does so using provenance headers. Replicator provides a range of metrics, like replication lag, and can be monitored using its REST API or Control Center UI. It also supports schema migration between clusters and can perform schema translation.

Multi-Region Clusters (MRC)

We saw earlier that stretch clusters provide simple transparent failover and fallback for clients without the need for offset translation or client restarts. But stretch clusters require datacenters to be close to each other and provide a stable low-latency network to enable synchronous replication between datacenters. MRC is also suitable only for datacenters within a 50 ms latency, but it uses a

combination of synchronous and asynchronous replication to limit impact on producer performance and provide higher network tolerance.

As we saw earlier, Apache Kafka supports fetching from followers to enable clients to fetch from their closest brokers based on rack ID, thereby reducing cross-datacenter traffic. Confluent Server also adds the concept of *observers*, which are asynchronous replicas that do not join the ISR and hence have no impact on producers using `acks=all` but are able to deliver records to consumers. Operators can configure synchronous replication within a region and asynchronous replication between regions to benefit from both low latency and high durability at the same time. Replica placement constraints in Confluent Server allow you to specify a minimum number of replicas per region using rack IDs to ensure that replicas are spread across regions to guarantee durability. Confluent Platform 6.1 also adds automatic observer promotion with configurable criteria, enabling fast failover without data loss automatically. When `min.insync.replicas` falls below a configured minimum number of synchronous replicas, observers that have caught up are automatically promoted to allow them to join ISRs, bringing the number of ISRs back up to the required minimum. The promoted observers use synchronous replication and may impact throughput, but the cluster remains operational throughout without data loss even if a region fails. When the failed region recovers, observers are automatically demoted, getting the cluster back to normal performance levels.

Cluster Linking

Cluster Linking, introduced as a preview feature in Confluent Platform 6.0, builds inter-cluster replication directly into the Confluent Server. By using the same protocol as inter-broker replication within a cluster, Cluster Linking performs offset-preserving replication across clusters, enabling seamless migration of clients without any need for offset translation. Topic configuration, partitions, consumer offsets, and ACLs are all kept synchronized between the two clusters to enable failover with low RTO if a disaster occurs. A cluster link defines the configuration of a directional flow from a source cluster to a destination cluster. Leader brokers of mirror partitions in the destination cluster fetch partition data from the corresponding source leaders, while followers in the destination replicate from their local leader using the standard replication mechanism in Kafka. Mirror topics are marked as read-only in the destination to prevent any local produce to these topics, ensuring that mirror topics are logically identical to their source topic.

Cluster Linking provides operational simplicity without the need for separate clusters like Connect clusters and is more performant than external tools since it avoids decompression and recompression during mirroring. Unlike MRC, there is no option for synchronous replication, and client failover is a manual process that requires client restart. But Cluster Linking may be used with distant

datacenters with unreliable high-latency networks and reduces cross-datacenter traffic by replicating only once between datacenters. It is suitable for cluster migration and topic sharing use cases.

Summary

We started the chapter by describing the reasons you may need to manage more than a single Kafka cluster and then proceeded to describe several common multicluster architectures, ranging from the simple to the very complex. We went into the details of implementing failover architecture for Kafka and compared the different options currently available. Then we proceeded to discuss the available tools. Starting with Apache Kafka's MirrorMaker, we went into many details of using it in production. We finished by reviewing alternative options that solve some of the issues you might encounter with MirrorMaker.

Whichever architecture and tools you end up using, remember that multicluster configuration and mirroring pipelines should be monitored and tested just like everything else you take into production. Because multicluster management in Kafka can be easier than it is with relational databases, some organizations treat it as an after-thought and neglect to apply proper design, planning, testing, deployment automation, monitoring, and maintenance. By taking multicluster management seriously, preferably as part of a holistic disaster or geodiversity plan for the entire organization that involves multiple applications and data stores, you will greatly increase the chances of successfully managing multiple Kafka clusters.

Securing Kafka

Kafka is used for a variety of use cases ranging from website activity tracking and metrics pipelines to patient record management and online payments. Each use case has different requirements in terms of security, performance, reliability, and availability. While it is always preferable to use the strongest and latest security features available, trade-offs are often necessary since increased security impacts performance, cost, and user experience. Kafka supports several standard security technologies with a range of configuration options to tailor security to each use case.

Like performance and reliability, security is an aspect of the system that must be addressed for the system as a whole, rather than component by component. The security of a system is only as strong as the weakest link, and security processes and policies must be enforced across the system, including the underlying platform. The customizable security features in Kafka enable integration with existing security infrastructure to build a consistent security model that applies to the entire system.

In this chapter, we will discuss the security features in Kafka and see how they address different aspects of security and contribute toward the overall security of the Kafka installation. Throughout the chapter, we will share best practices, potential threats, and techniques to mitigate these threats. We will also review additional measures that can be adopted to secure ZooKeeper and the rest of the platform.

Locking Down Kafka

Kafka uses a range of security procedures to establish and maintain confidentiality, integrity, and availability of data:

- Authentication establishes your identity and determines *who* you are.
- Authorization determines *what* you are allowed to do.

- Encryption protects your data from eavesdropping and tampering.
- Auditing tracks what you have done or have attempted to do.
- Quotas control how much resources you can utilize.

To understand how to lock down a Kafka deployment, let's first look at how data flows through a Kafka cluster. [Figure 11-1](#) shows the main steps in an example data flow. In this chapter, we will use this example flow to examine the different ways in which Kafka can be configured to protect data at every step to guarantee security of the entire deployment.

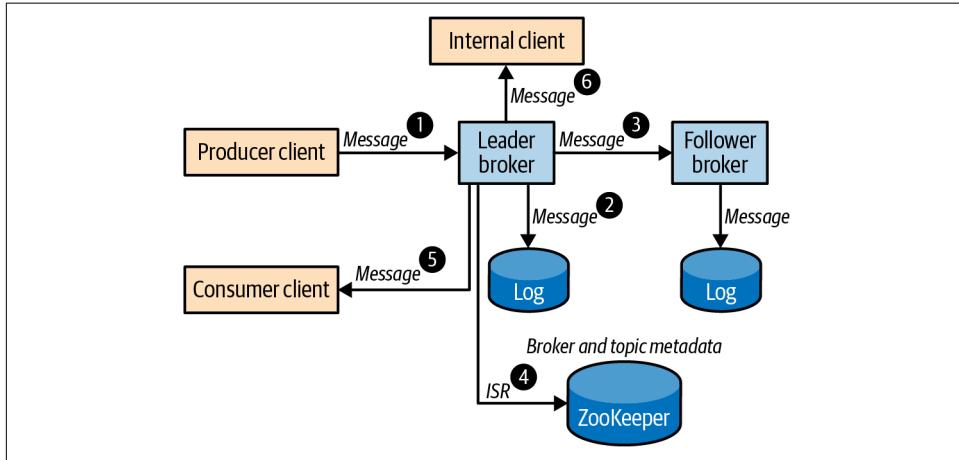


Figure 11-1. Data flow in a Kafka cluster

1. Alice produces a customer order record to a partition of the topic named `customerOrders`. The record is sent to the leader of the partition.
2. The leader broker writes the record to its local log file.
3. A follower broker fetches the message from the leader and writes to its local replica log file.
4. The leader broker updates the partition state in ZooKeeper to update in-sync replicas, if required.
5. Bob consumes customer order records from the topic `customerOrders`. Bob receives the record produced by Alice.
6. An internal application processes all messages arriving in `customerOrders` to produce real-time metrics on popular products.

A secure deployment must guarantee:

Client authenticity

When Alice establishes a client connection to the broker, the broker should authenticate the client to ensure that the message is really coming from Alice.

Server authenticity

Before sending a message to the leader broker, Alice's client should verify that the connection is to the real broker.

Data privacy

All connections where the message flows, as well as all disks where messages are stored, should be encrypted or physically secured to prevent eavesdroppers from reading the data and to ensure that data cannot be stolen.

Data integrity

Message digests should be included for data transmitted over insecure networks to detect tampering.

Access control

Before writing the message to the log, the leader broker should verify that Alice is authorized to write to `customerOrders`. Before returning messages to Bob's consumer, the broker should verify that Bob is authorized to read from the topic. If Bob's consumer uses group management, the broker should also verify that Bob has access to the consumer group.

Auditability

An audit trail that shows all operations that were performed by brokers, Alice, Bob, and other clients should be logged.

Availability

Brokers should apply quotas and limits to avoid some users hogging all the available bandwidth or overwhelming the broker with denial-of-service attacks. ZooKeeper should be locked down to ensure availability of the Kafka cluster since broker availability is dependent on ZooKeeper availability and the integrity of metadata stored in ZooKeeper.

In the following sections, we explore the Kafka security features that can be used to provide these guarantees. We first introduce the Kafka connection model and the security protocols associated with connections from clients to Kafka brokers. We then look at each security protocol in detail and examine the authentication capabilities of each protocol to ascertain client authenticity and server authenticity. We review options for encryption at different stages, including built-in encryption of data in transit in some security protocols to address data privacy and data integrity. Then, we explore customizable authorization in Kafka to manage access control and the main logs that contribute to auditability. Finally, we review security for the rest of the

system, including ZooKeeper and the platform, which is necessary to maintain availability. For details on quotas that contribute to service availability through fair allocation of resources among users, refer to [Chapter 3](#).

Security Protocols

Kafka brokers are configured with listeners on one or more endpoints and accept client connections on these listeners. Each listener can be configured with its own security settings. Security requirements on a private internal listener that is physically protected and only accessible to authorized personnel may be different from the security requirements of an external listener accessible over the public internet. The choice of security protocol determines the level of authentication and encryption of data in transit.

Kafka supports four security protocols using two standard technologies, TLS and SASL. Transport Layer Security (TLS), commonly referred to by the name of its predecessor, Secure Sockets Layer (SSL), supports encryption as well as client and server authentication. Simple Authentication and Security Layer (SASL) is a framework for providing authentication using different mechanisms in connection-oriented protocols. Each Kafka security protocol combines a transport layer (PLAINTEXT or SSL) with an optional authentication layer (SSL or SASL):

PLAINTEXT

PLAINTEXT transport layer with no authentication. Is suitable only for use within private networks for processing data that is not sensitive since no authentication or encryption is used.

SSL

SSL transport layer with optional SSL client authentication. Is suitable for use in insecure networks since client and server authentication as well as encryption are supported.

SASL_PLAINTEXT

PLAINTEXT transport layer with SASL client authentication. Some SASL mechanisms also support server authentication. Does not support encryption and hence is suitable only for use within private networks.

SASL_SSL

SSL transport layer with SASL authentication. Is suitable for use in insecure networks since client and server authentication as well as encryption are supported.



TLS/SSL

TLS is one of the most widely used cryptographic protocols on the public internet. Application protocols like HTTP, SMTP, and FTP rely on TLS to provide privacy and integrity of data in transit. TLS relies on a Public Key Infrastructure (PKI) to create, manage, and distribute digital certificates that can be used for asymmetric encryption, avoiding the need for distributing shared secrets between servers and clients. Session keys generated during the TLS handshake enable symmetric encryption with higher performance for subsequent data transfer.

The listener used for inter-broker communication can be selected by configuring `inter.broker.listener.name` or `security.inter.broker.protocol`. Both server-side and client-side configuration options must be provided in the broker configuration for the security protocol used for inter-broker communication. This is because brokers need to establish client connections for that listener. The following example configures SSL for the inter-broker and internal listeners, and SASL_SSL for the external listener:

```
listeners=EXTERNAL://:9092,INTERNAL://10.0.0.2:9093,BROKER://10.0.0.2:9094  
advertised.listeners=EXTERNAL://broker1.example.com:9092,INTERNAL://  
broker1.local:9093,BROKER://broker1.local:9094  
listener.security.protocol.map=EXTERNAL:SASL_SSL,INTERNAL:SSL,BROKER:SSL  
inter.broker.listener.name=BROKER
```

Clients are configured with a security protocol and bootstrap servers that determine the broker listener. Metadata returned to clients contains only the endpoints corresponding to the same listener as the bootstrap servers:

```
security.protocol=SASL_SSL  
bootstrap.servers=broker1.example.com:9092,broker2.example.com:9092
```

In the next section on authentication, we review the protocol-specific configuration options for brokers and clients for each security protocol.

Authentication

Authentication is the process of establishing the identity of the client and server to verify client authenticity and server authenticity. When Alice's client connects to the leader broker to produce a customer order record, server authentication enables the client to establish that the server that the client is talking to is the actual broker. Client authentication verifies Alice's identity by validating Alice's credentials, like a password or digital certificate, to determine that the connection is from Alice and not an impersonator. Once authenticated, Alice's identity is associated with the connection throughout the lifetime of the connection. Kafka uses an instance of `KafkaPrincipal` to represent client identity and uses this principal to grant access to resources and

allocate quotas for connections with that client identity. The `KafkaPrincipal` for each connection is established during authentication based on the authentication protocol. For example, the principal `User: Alice` may be used for Alice based on the username provided for password-based authentication. `KafkaPrincipal` may be customized by configuring `principal.builder.class` for brokers.



Anonymous Connections

The principal `User: ANONYMOUS` is used for unauthenticated connections. This includes clients on `PLAINTEXT` listeners as well as unauthenticated clients on `SSL` listeners.

SSL

When Kafka is configured with `SSL` or `SASL_SSL` as the security protocol for a listener, `TLS` is used as the secure transport layer for connections on that listener. When a connection is established over `TLS`, the `TLS` handshake process performs authentication, negotiates cryptographic parameters, and generates shared keys for encryption. The server's digital certificate is verified by the client to establish the identity of the server. If client authentication using `SSL` is enabled, the server also verifies the client's digital certificate to establish the identity of the client. All traffic over `SSL` is encrypted, making it suitable for use in insecure networks.



SSL Performance

`SSL` channels are encrypted and hence introduce a noticeable overhead in terms of CPU usage. Zero-copy transfer is currently not supported for `SSL`. Depending on the traffic pattern, the overhead may be up to 20–30%.

Configuring TLS

When `TLS` is enabled for a broker listener using `SSL` or `SASL_SSL`, brokers should be configured with a key store containing the broker's private key and certificate, and clients should be configured with a trust store containing the broker certificate or the certificate of the certificate authority (CA) that signed the broker certificate. Broker certificates should contain the broker hostname as a Subject Alternative Name (SAN) extension or as the Common Name (CN) to enable clients to verify the server hostname. Wildcard certificates can be used to simplify administration by using the same key store for all brokers in a domain.



Server Hostname verification

By default, Kafka clients verify that the hostname of the server stored in the server certificate matches the host that the client is connecting to. The connection hostname may be a bootstrap server that the client is configured with or an advertised listener hostname that was returned by a broker in a metadata response. Hostname verification is a critical part of server authentication that protects against man-in-the-middle attacks and hence should not be disabled in production systems.

Brokers can be configured to authenticate clients connecting over listeners using SSL as the security protocol by setting the broker configuration option `ssl.client.auth=required`. Clients should be configured with a key store, and brokers should be configured with a trust store containing client certificates or the certificate of the CAs that signed the client certificates. If SSL is used for inter-broker communication, broker trust stores should include the CA of the broker certificates as well as the CA of the client certificates. By default, the distinguished name (DN) of the client certificate is used as the `KafkaPrincipal` for authorization and quotas. The configuration option `ssl.principal.mapping.rules` can be used to provide a list of rules to customize the principal. Listeners using `SASL_SSL` disable TLS client authentication and rely on SASL authentication and the `KafkaPrincipal` established by SASL.



SSL Client Authentication

SSL client authentication may be made optional by setting `ssl.client.auth=requested`. Clients that are not configured with key stores will complete the TLS handshake in this case, but will be assigned the principal `User:ANONYMOUS`.

The following examples show how to create key stores and trust stores for server and client authentication using a self-signed CA.

Generate self-signed CA key-pair for brokers:

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -keystore server.ca.p12 \
  -storetype PKCS12 -storepass server-ca-password -keypass server-ca-password \
  -alias ca -dname "CN=BrokerCA" -ext bc=ca:true -validity 365 ❶
$ keytool -export -file server.ca.crt -keystore server.ca.p12 \
  -storetype PKCS12 -storepass server-ca-password -alias ca -rfc ❷
```

- ❶ Create a key-pair for the CA and store it in a PKCS12 file `server.ca.p12`. We use this for signing certificates.
- ❷ Export the CA's public certificate to `server.ca.crt`. This will be included in trust stores and certificate chains.

Create key stores for brokers with a certificate signed by the self-signed CA. If using wildcard hostnames, the same key store can be used for all brokers. Otherwise, create a key store for each broker with its fully qualified domain name (FQDN):

```
$ keytool -genkey -keyalg RSA -keysize 2048 -keystore server.ks.p12 \
  -storepass server-ks-password -keypass server-ks-password -alias server \
  -storetype PKCS12 -dname "CN=Kafka,0=Confluent,C=GB" -validity 365 ① \
$ keytool -certreq -file server.csr -keystore server.ks.p12 -storetype PKCS12 \
  -storepass server-ks-password -keypass server-ks-password -alias server ② \
$ keytool -gencert -infile server.csr -outfile server.crt \
  -keystore server.ca.p12 -storetype PKCS12 -storepass server-ca-password \
  -alias ca -ext SAN=DNS:broker1.example.com -validity 365 ③ \
$ cat server.crt server.ca.crt > serverchain.crt \
$ keytool -importcert -file serverchain.crt -keystore server.ks.p12 \
  -storepass server-ks-password -keypass server-ks-password -alias server \
  -storetype PKCS12 -noprompt ④ \
```

- ① Generate a private key for a broker and store it in the PKCS12 file server.ks.p12.
- ② Generate a certificate signing request.
- ③ Use the CA key store to sign the broker's certificate. The signed certificate is stored in server.crt.
- ④ Import the broker's certificate chain into the broker's key store.

If TLS is used for inter-broker communication, create a trust store for brokers with the broker's CA certificate to enable brokers to authenticate one another:

```
$ keytool -import -file server.ca.crt -keystore server.ts.p12 \
  -storetype PKCS12 -storepass server-ts-password -alias server -noprompt
```

Generate a trust store for clients with the broker's CA certificate:

```
$ keytool -import -file server.ca.crt -keystore client.ts.p12 \
  -storetype PKCS12 -storepass client-ts-password -alias ca -noprompt
```

If TLS client authentication is enabled, clients must be configured with a key store. The following script generates a self-signed CA for clients and creates a key store for clients with a certificate signed by the client CA. The client CA is added to the broker trust store so that brokers can verify client authenticity:

```
# Generate self-signed CA key-pair for clients
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore client.ca.p12 \
  -storetype PKCS12 -storepass client-ca-password -keypass client-ca-password \
  -alias ca -dname CN=ClientCA -ext bc=ca:true -validity 365 ① \
keytool -export -file client.ca.crt -keystore client.ca.p12 -storetype PKCS12 \
  -storepass client-ca-password -alias ca -rfc \
# Create key store for clients
keytool -genkey -keyalg RSA -keysize 2048 -keystore client.ks.p12 \
```

```

-storepass client-ks-password -keypass client-ks-password -alias client \
-storetype PKCS12 -dname "CN=Metrics App,O=Confluent,C=GB" -validity 365 ②
keytool -certreq -file client.csr -keystore client.ks.p12 -storetype PKCS12 \
-storepass client-ks-password -keypass client-ks-password -alias client
keytool -gencert -infile client.csr -outfile client.crt \
-keystore client.ca.p12 -storetype PKCS12 -storepass client-ca-password \
-alias ca -validity 365
cat client.crt client.ca.crt > clientchain.crt
keytool -importcert -file clientchain.crt -keystore client.ks.p12 \
-storepass client-ks-password -keypass client-ks-password -alias client \
-storetype PKCS12 -noprompt ③

# Add client CA certificate to broker's trust store
keytool -import -file client.ca.crt -keystore server.ts.p12 -alias client \
-storetype PKCS12 -storepass server-ts-password -noprompt ④

```

- ① We create a new CA for clients in this example.
- ② Clients authenticating with this certificate use User:CN=Metrics App, O=Confluent,C=GB as the principal, by default.
- ③ We add the client certificate chain to the client key store.
- ④ The broker's trust store should contain the CAs of all clients.

Once we have the key and trust stores, we can configure TLS for brokers. Brokers require a trust store only if TLS is used for inter-broker communication or if client authentication is enabled:

```

ssl.keystore.location=/path/to/server.ks.p12
ssl.keystore.password=server-ks-password
ssl.key.password=server-ks-password
ssl.keystore.type=PKCS12
ssl.truststore.location=/path/to/server.ts.p12
ssl.truststore.password=server-ts-password
ssl.truststore.type=PKCS12
ssl.client.auth=required

```

Clients are configured with the generated trust store. The key store should be configured for clients if client authentication is required.

```

ssl.truststore.location=/path/to/client.ts.p12
ssl.truststore.password=client-ts-password
ssl.truststore.type=PKCS12
ssl.keystore.location=/path/to/client.ks.p12
ssl.keystore.password=client-ks-password
ssl.key.password=client-ks-password
ssl.keystore.type=PKCS12

```



Trust Stores

Trust store configuration can be omitted in brokers as well as clients when using certificates signed by well-known trusted authorities. The default trust stores in the Java installation will be sufficient to establish trust in this case. Installation steps are described in [Chapter 2](#).

Key stores and trust stores must be updated periodically before certificates expire to avoid TLS handshake failures. Broker SSL stores can be dynamically updated by modifying the same file or setting the configuration option to a new versioned file. In both cases, the Admin API or the Kafka configs tool can be used to trigger the update. The following example updates the key store for the external listener of a broker with broker id 0 using the configs tool:

```
$ bin/kafka-configs.sh --bootstrap-server localhost:9092
  --command-config admin.props
  --entity-type brokers --entity-name 0 --alter --add-config \
    'listener.name.external.ssl.keystore.location=/path/to/server.ks.p12'
```

Security considerations

TLS is widely used to provide transport layer security for several protocols, including HTTPS. As with any security protocol, it is important to understand the potential threats and mitigation strategies when adopting a protocol for mission-critical applications. Kafka enables only the newer protocols TLSv1.2 and TLSv1.3 by default, since older protocols like TLSv1.1 have known vulnerabilities. Due to issues with insecure renegotiation, Kafka does not support renegotiation for TLS connections. Hostname verification is enabled by default to prevent man-in-the-middle attacks. Security can be tightened further by restricting cipher suites. Strong ciphers with at least a 256-bit encryption key size protect against cryptographic attacks and ensure data integrity when transporting data over an insecure network. Some organizations require TLS protocol and ciphers to be restricted to comply with security standards like FIPS 140-2.

Since key stores containing private keys are stored on the filesystem by default, it is vital to limit access to key store files using filesystem permissions. Standard Java TLS features can be used to enable certificate revocation if a private key is compromised. Short-lived keys can be used to reduce exposure in this case.

TLS handshakes are expensive and utilize a significant amount of time on network threads in brokers. Listeners using TLS on insecure networks should be protected against denial-of-service attacks using connection quotas and limits to protect availability of brokers. The broker configuration option `connection.failed.authentication.delay.ms` can be used to delay failed response on authentication failures to reduce the rate at which authentication failures are retried by clients.

SASL

Kafka protocol supports authentication using SASL and has built-in support for several commonly used SASL mechanisms. SASL can be combined with TLS as the transport layer to provide a secure channel with authentication and encryption. SASL authentication is performed through a sequence of server challenges and client responses where the SASL mechanism defines the sequence and wire format of challenges and responses. Kafka brokers support the following SASL mechanisms out of the box with customizable callbacks to integrate with existing security infrastructure:

GSSAPI

Kerberos authentication is supported using SASL/GSSAPI and can be used to integrate with Kerberos servers like Active Directory or OpenLDAP.

PLAIN

Username/password authentication that is typically used with a custom server-side callback to verify passwords from an external password store.

SCRAM-SHA-256 and SCRAM-SHA-512

Username/password authentication available out of the box with Kafka without the need for additional password stores.

OAUTHBearer

Authentication using OAuth bearer tokens that is typically used with custom callbacks to acquire and validate tokens granted by standard OAuth servers.

One or more SASL mechanisms may be enabled on each SASL-enabled listener in the broker by configuring `sasl.enabled.mechanisms` for that listener. Clients may choose any of the enabled mechanisms by configuring `sasl.mechanism`.

Kafka uses the Java Authentication and Authorization Service (JAAS) for configuring SASL. The configuration option `sasl.jaas.config` contains a single JAAS configuration entry that specifies a login module and its options. Brokers use the `listener` and `mechanism` prefixes when configuring `sasl.jaas.config`. For example, `listener.name.external.gssapi.sasl.jaas.config` configures the JAAS configuration entry for SASL/GSSAPI on the listener named EXTERNAL. The login process on brokers and clients uses the JAAS configuration to determine the public and private credentials used for authentication.



JAAS Configuration File

JAAS configuration may also be specified in configuration files using the Java system property `java.security.auth.login.config`. However, the Kafka option `sasl.jaas.config` is recommended since it supports password protection and separate configuration for each SASL mechanism when multiple mechanisms are enabled on a listener.

SASL mechanisms supported by Kafka can be customized to integrate with third-party authentication servers using callback handlers. A login callback handler may be provided for brokers or clients to customize the login process, for example, to acquire credentials to be used for authentication. A server callback handler may be provided to perform authentication of client credentials, for example, to verify passwords using an external password server. A client callback handler may be provided to inject client credentials instead of including them in the JAAS configuration.

In the following subsections, we explore the SASL mechanisms supported by Kafka in more detail.

SASL/GSSAPI

Kerberos is a widely used network authentication protocol that uses strong cryptography to support secure mutual authentication over an insecure network. Generic Security Service Application Program Interface (GSS-API) is a framework for providing security services to applications using different authentication mechanisms. [RFC-4752](#) introduces the SASL mechanism GSSAPI for authentication using GSS-API's Kerberos V5 mechanism. The availability of open source as well as enterprise-grade commercial implementations of Kerberos servers has made Kerberos a popular choice for authentication across many sectors with strict security requirements. Kafka supports Kerberos authentication using SASL/GSSAPI.

Configuring SASL/GSSAPI. Kafka uses GSSAPI security providers included in the Java runtime environment to support secure authentication using Kerberos. JAAS configuration for GSSAPI includes the path of a keytab file that contains the mapping of principals to their long-term keys in encrypted form. To configure GSSAPI for brokers, create a keytab for each broker with a principal that includes the broker's hostname. Broker hostnames are verified by clients to ensure server authenticity and prevent man-in-the-middle attacks. Kerberos requires a secure DNS service for host name lookup during authentication. In deployments where forward and reverse lookup do not match, the Kerberos configuration file `krb5.conf` on clients can be configured to set `rdns=false` to disable reverse lookup. JAAS configuration for each broker should include the Kerberos V5 login module from the Java runtime, the pathname of the keytab file, and the full broker principal:

```
sasl.enabled.mechanisms=GSSAPI
listener.name.external.gssapi.sasl.jaas.config=\ ①
    com.sun.security.auth.module.Krb5LoginModule required \
        useKeyTab=true storeKey=true      \
        keyTab="/path/to/broker1.keytab" \ ②
        principal="kafka/broker1.example.com@EXAMPLE.COM"; ③
```

- ① We use `sasl.jaas.config` prefixed with the listener prefix, which contains the listener name and SASL mechanism in lowercase.
- ② Keytab files must be readable by the broker process.
- ③ Service principal for brokers should include the broker hostname.

If SASL/GSSAPI is used for inter-broker communication, inter-broker SASL mechanism and the Kerberos service name should also be configured for brokers:

```
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka
```

Clients should be configured with their own keytab and principal in the JAAS configuration and `sasl.kerberos.service.name` to indicate the name of the service they are connecting to:

```
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka ①
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true storeKey=true      \
    keyTab="/path/to/alice.keytab" \ ②
    principal="Alice@EXAMPLE.COM";
```

- ① The service name for the Kafka service should be specified for clients.
- ② Clients may use principals without hostname.

The short name of the principal is used as the client identity by default. For example, `User:Alice` is the client principal and `User:kafka` is the broker principal in the example. The broker configuration `sasl.kerberos.principal.to.local.rules` can be used to apply a list of rules to transform the fully qualified principal to a custom principal.

Security considerations. Use of SASL_SSL is recommended in production deployments using Kerberos to protect the authentication flow as well as data traffic on the connection after authentication. If TLS is not used to provide a secure transport layer, eavesdroppers on the network may gain enough information to mount a dictionary attack or brute-force attack to steal client credentials. It is safer to use randomly generated keys for brokers instead of keys generated from passwords that are easier to crack. Weak encryption algorithms like DES-MD5 should be avoided in favor of

stronger algorithms. Access to keytab files must be restricted using filesystem permissions since any user in possession of the file may impersonate the user.

SASL/GSSAPI requires a secure DNS service for server authentication. Because denial-of-service attacks against the KDC or DNS service can result in authentication failures in clients, it is necessary to monitor the availability of these services. Kerberos also relies on loosely synchronized clocks with configurable variability to detect replay attacks. It is important to ensure that clock synchronization is secure.

SASL/PLAIN

[RFC-4616](#) defines a simple username/password authentication mechanism that can be used with TLS to provide secure authentication. During authentication, the client sends a username and password to the server, and the server verifies the password using its password store. Kafka has built-in SASL/PLAIN support that can be integrated with a secure external password database using a custom callback handler.

Configuring SASL/PLAIN. The default implementation of SASL/PLAIN uses the broker's JAAS configuration as the password store. All client usernames and passwords are included as login options, and the broker verifies that the password provided by a client during authentication matches one of these entries. A broker username and password are required only if SASL/PLAIN is used for inter-broker communication:

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.external.plain.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required \
        username="kafka" password="kafka-password" \ ①
        user_kafka="kafka-password" \
        user_Alice="Alice-password"; ②
```

- ① The username and password used for inter-broker connections initiated by the broker.
- ② When Alice's client connects to the broker, the password provided by Alice is validated against this password in the broker's config.

Clients must be configured with username and password for authentication:

```
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required username="Alice" password="Alice-password";
```

The built-in implementation that stores all passwords in every broker's JAAS configuration is insecure and not very flexible since all brokers will need to be restarted to add or remove a user. When using SASL/PLAIN in production, a custom server callback handler can be used to integrate brokers with a secure third-party password server. Custom callback handlers can also be used to support password rotation. On

the server side, a server callback handler should support both old and new passwords for an overlapping period until all clients switch to the new password. The following example shows a callback handler that verifies encrypted passwords from files generated using the Apache tool `htpasswd`:

```
public class PasswordVerifier extends PlainServerCallbackHandler {

    private final List<String> passwdFiles = new ArrayList<>(); ①

    @Override
    public void configure(Map<String, ?> configs, String mechanism,
        List<AppConfigurationEntry> jaasEntries) {
        Map<String, ?> loginOptions = jaasEntries.get(0).getOptions();
        String files = (String) loginOptions.get("password.files"); ②
        Collections.addAll(passwdFiles, files.split(","));
    }

    @Override
    protected boolean authenticate(String user, char[] password) {
        return passwdFiles.stream() ③
            .anyMatch(file -> authenticate(file, user, password));
    }

    private boolean authenticate(String file, String user, char[] password) {
        try {
            String cmd = String.format("htpasswd -vb %s %s %s",
                file, user, new String(password)); ④
            return Runtime.getRuntime().exec(cmd).waitFor() == 0;
        } catch (Exception e) {
            return false;
        }
    }
}
```

- ① We use multiple password files so that we can support password rotation.
- ② We pass pathnames of password files as a JAAS option in the broker configuration. Custom broker configuration options may also be used.
- ③ We check if the password matches in any of the files, allowing both old and new passwords to be used for a period of time.
- ④ We use `htpasswd` for simplicity. A secure database can be used for production deployments.

Brokers are configured with the password validation callback handler and its options:

```
listener.name.external.plain.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required \
    password.files="/path/to/htpasswd.props,/path/to/oldhtpasswd.props";
```

```
listener.name.external.plain.sasl.server.callback.handler.class=\
com.example.PasswordVerifier
```

On the client side, a client callback handler that implements `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` can be used to load passwords dynamically at runtime when a connection is established instead of loading statically from the JAAS configuration during startup. Passwords may be loaded from encrypted files or using an external secure server to improve security. The following example loads passwords dynamically from a file using configuration classes in Kafka:

```
@Override
public void handle(Callback[] callbacks) throws IOException {
    Properties props = Utils.loadProps(passwdFile); ①
    PasswordConfig config = new PasswordConfig(props);
    String user = config.getString("username");
    String password = config.getPassword("password").value(); ②
    for (Callback callback: callbacks) {
        if (callback instanceof NameCallback)
            ((NameCallback) callback).setName(user);
        else if (callback instanceof PasswordCallback) {
            ((PasswordCallback) callback).setPassword(password.toCharArray());
        }
    }
}

private static class PasswordConfig extends AbstractConfig {
    static ConfigDef CONFIG = new ConfigDef()
        .define("username", STRING, HIGH, "User name")
        .define("password", PASSWORD, HIGH, "User password"); ③
    PasswordConfig(Properties props) {
        super(CONFIG, props, false);
    }
}
```

- ① We load the config file within the callback to ensure we use the latest password to support password rotation.
- ② The underlying configuration library returns the actual password value even if the password is externalized.
- ③ We define password configs with the `PASSWORD` type to ensure that passwords are not included in log entries.

Clients as well as brokers that use SASL/PLAIN for inter-broker communication can be configured with the client-side callback:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule \
    required file="/path/to/credentials.props";
sasl.client.callback.handler.class=com.example.PasswordProvider
```

Security considerations. Since SASL/PLAIN transmits clear-text passwords over the wire, the PLAIN mechanism should be enabled only with encryption using SASL_SSL to provide a secure transport layer. Passwords stored in clear text in the JAAS configuration of brokers and clients are not secure, so consider encrypting or externalizing these passwords in a secure password store. Instead of using the built-in password store that stores all client passwords in the broker JAAS configuration, use a secure external password server that stores passwords securely and enforces strong password policies.



Clear-Text Passwords

Avoid clear-text passwords in configuration files even if the files can be protected using filesystem permissions. Consider externalizing or encrypting passwords to ensure that passwords are not inadvertently exposed. Kafka's password protection feature is described later in this chapter.

SASL/SCRAM

RFC-5802 introduces a secure username/password authentication mechanism that addresses the security concerns with password authentication mechanisms like SASL/PLAIN, which send passwords over the wire. The Salted Challenge Response Authentication Mechanism (SCRAM) avoids transmitting clear-text passwords and stores passwords in a format that makes it impractical to impersonate clients. Salting combines passwords with some random data before applying a one-way cryptographic hash function to store passwords securely. Kafka has a built-in SCRAM provider that can be used in deployments with secure ZooKeeper without the need for additional password servers. The SCRAM mechanisms SCRAM-SHA-256 and SCRAM-SHA-512 are supported by the Kafka provider.

Configuring SASL/SCRAM. An initial set of users can be created after starting ZooKeeper prior to starting brokers. Brokers load SCRAM user metadata into an in-memory cache during startup, ensuring that all users, including the broker user for inter-broker communication, can authenticate successfully. Users can be added or deleted at any time. Brokers keep the cache up-to-date using notifications based on a ZooKeeper watcher. In this example, we create a user with the principal `User:Alice` and password `Alice-password` for SASL mechanism SCRAM-SHA-512:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config \
  'SCRAM-SHA-512=[iterations=8192,password=Alice-password]' \
  --entity-type users --entity-name Alice
```

One or more SCRAM mechanisms can be enabled on a listener by configuring the mechanisms on the broker. A username and password are required for brokers only if the listener is used for inter-broker communication:

```
sasl.enabled.mechanisms=SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
listener.name.external.scram-sha-512.sasl.jaas.config=\
    org.apache.kafka.common.security.scram.ScramLoginModule required \
        username="kafka" password="kafka-password"; ①
```

- ① Username and password for inter-broker connections initiated by the broker.

Clients must be configured to use one of the SASL mechanisms enabled in the broker, and the client JAAS configuration must include a username and password:

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
    required username="Alice" password="Alice-password";
```

You can add new SCRAM users using `--add-config` and delete users using the `--delete-config` option of the `configs` tool. When an existing user is deleted, new connections cannot be established for that user, but existing connections of the user will continue to work. A reauthentication interval can be configured for the broker to limit the amount of time existing connections may continue to operate after a user is deleted. The following example deletes the SCRAM-SHA-512 config for Alice to remove Alice's credentials for that mechanism:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config \
    'SCRAM-SHA-512' --entity-type users --entity-name Alice
```

Security considerations. SCRAM applies a one-way cryptographic hash function on the password combined with a random salt to avoid the actual password being transmitted over the wire or stored in a database. However, any password-based system is only as secure as the passwords. Strong password policies must be enforced to protect the system from brute-force or dictionary attacks. Kafka provides safeguards by supporting only the strong hashing algorithms SHA-256 and SHA-512 and avoiding weaker algorithms like SHA-1. This is combined with a high default iteration count of 4,096 and unique random salts for every stored key to limit the impact if ZooKeeper security is compromised.

You should take additional precautions to protect the keys transmitted during handshake and the keys stored in ZooKeeper to protect against brute-force attacks. SCRAM must be used with SASL_SSL as the security protocol to avoid eavesdroppers from gaining access to hashed keys during authentication. ZooKeeper must also be SSL-enabled, and ZooKeeper data must be protected using disk encryption to ensure that stored keys cannot be retrieved even if the store is compromised. In deployments without a secure ZooKeeper, SCRAM callbacks can be used to integrate with a secure external credential store.

SASL/OAUTHBEARER

OAuth is an authorization framework that enables applications to obtain limited access to HTTP services. [RFC-7628](#) defines the OAUTHBEARER SASL mechanism that enables credentials obtained using OAuth 2.0 to access protected resources in non-HTTP protocols. OAUTHBEARER avoids security vulnerabilities in mechanisms that use long-term passwords by using OAuth 2.0 bearer tokens with a shorter lifetime and limited resource access. Kafka supports SASL/OAUTHBEARER for client authentication, enabling integration with third-party OAuth servers. The built-in implementation of OAUTHBEARER uses unsecured JSON Web Tokens (JWTs) and is not suitable for production use. Custom callbacks can be added to integrate with standard OAuth servers to provide secure authentication using the OAUTHBEARER mechanism in production deployments.

Configuring SASL/OAUTHBEARER. The built-in implementation of SASL/OAUTHBEARER in Kafka does not validate tokens and hence only requires the login module to be specified in the JAAS configuration. If the listener is used for inter-broker communication, details of the token used for client connections initiated by brokers must also be provided. The option `unsecuredLoginStringClaim_sub` is the subject claim that determines the `KafkaPrincipal` for the connection by default:

```
sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
listener.name.external.oauthbearer.sasl.jaas.config=\
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
        required unsecuredLoginStringClaim_sub="kafka"; ①
```

- ① Subject claim for the token used for inter-broker connections.

Clients must be configured with the subject claim option `unsecuredLoginStringClaim_sub`. Other claims and token lifetime may also be configured:

```
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=\
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule \
        required unsecuredLoginStringClaim_sub="Alice"; ①
```

- ① `User:Alice` is the default `KafkaPrincipal` for connections using this configuration.

To integrate Kafka with third-party OAuth servers for using bearer tokens in production, Kafka clients must be configured with `sasl.login.callback.handler.class` to acquire tokens from the OAuth server using the long-term password or a refresh token. If OAUTHBEARER is used for inter-broker communication, brokers must also be configured with a login callback handler to acquire tokens for client connections created by the broker for inter-broker communication:

```

@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    OAuthBearerToken token = null;
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerTokenCallback) {
            token = acquireToken(); ①
            ((OAuthBearerTokenCallback) callback).token(token);
        } else if (callback instanceof SaslExtensionsCallback) { ②
            ((SaslExtensionsCallback) callback).extensions(processExtensions(token));
        } else
            throw new UnsupportedCallbackException(callback);
    }
}

```

- ① Clients must acquire a token from the OAuth server and set a valid token on the callback.
- ② The client may also include optional extensions.

Brokers must also be configured with a server callback handler using `listener.name.<listener-name>.oauthbearer.sasl.server.callback.handler`. class for validating tokens provided by the client:

```

@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerValidatorCallback) {
            OAuthBearerValidatorCallback cb = (OAuthBearerValidatorCallback) callback;
            try {
                cb.token(validatedToken(cb.tokenValue())); ①
            } catch (OAuthBearerIllegalTokenException e) {
                OAuthBearerValidationResult r = e.reason();
                cb.error(errorStatus(r), r.failureScope(), r.failureOpenIdConfig());
            }
        } else if (callback instanceof OAuthBearerExtensionsValidatorCallback) {
            OAuthBearerExtensionsValidatorCallback ecb =
                (OAuthBearerExtensionsValidatorCallback) callback;
            ecb.inputExtensions().map().forEach((k, v) ->
                ecb.valid(validateExtension(k, v))); ②
        } else {
            throw new UnsupportedCallbackException(callback);
        }
    }
}

```

- ① `OAuthBearerValidatorCallback` contains the token from the client. Brokers validate this token.
- ② Brokers validate any optional extensions from the client.

Security considerations. Since SASL/OAUTHBEARER clients send OAuth 2.0 bearer tokens over the network and these tokens may be used to impersonate clients, TLS must be enabled to encrypt authentication traffic. Short-lived tokens can be used to limit exposure if tokens are compromised. Reauthentication may be enabled for brokers to prevent connections outliving the tokens used for authentication. A reauthentication interval configured on brokers, combined with token revocation support, limit the amount of time an existing connection may continue to use a token after revocation.

Delegation tokens

Delegation tokens are shared secrets between Kafka brokers and clients that provide a lightweight configuration mechanism without the requirement to distribute SSL key stores or Kerberos keytabs to client applications. Delegation tokens can be used to reduce the load on authentication servers, like the Kerberos Key Distribution Center (KDC). Frameworks like Kafka Connect can use delegation tokens to simplify security configuration for workers. A client that has authenticated with Kafka brokers can create delegation tokens for the same user principal and distribute these tokens to workers, which can then authenticate directly with Kafka brokers. Each delegation token consists of a token identifier and a hash-based message authentication code (HMAC) used as a shared secret. Client authentication with delegation tokens is performed using SASL/SCRAM with the token identifier as username and HMAC as the password.

Delegation tokens can be created or renewed using the Kafka Admin API or the `delegation-tokens` command. To create delegation tokens for the principal `User:Alice`, the client must be authenticated using Alice's credentials for any authentication protocol other than delegation tokens. Clients authenticated using delegation tokens cannot create other delegation tokens:

```
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \
--command-config admin.props --create --max-life-time-period -1 \
--renewer-principal User:Bob ①
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 \
--command-config admin.props --renew --renew-time-period -1 --hmac c2VjcmV0 ②
```

- ① If Alice runs this command, the generated token can be used to impersonate Alice. The owner of this token is `User:Alice`. We also configure `User:Bob` as a token renewer.
- ② The renewal command can be run by the token owner (Alice) or the token renewer (Bob).

Configuring delegation tokens. To create and validate delegation tokens, all brokers must be configured with the same master key using the configuration option `delegation.token.master.key`. This key can only be rotated by restarting all brokers. All existing tokens should be deleted before updating the master key since they can no longer be used, and new tokens should be created after the key is updated on all brokers.

At least one of the SASL/SCRAM mechanisms must be enabled on brokers to support authentication using delegation tokens. Clients should be configured to use SCRAM with a token identifier as username and token HMAC as the password. The Kafka Principal for the connections using this configuration will be the original principal associated with the token, e.g., `User:Alice`:

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule \
    required tokenauth="true" username="MTIz" password="c2VjcmV0"; ①
```

- ① SCRAM configuration with `tokenauth` is used to configure delegation tokens.

Security considerations. Like the built-in SCRAM implementation, delegation tokens are suitable for production use only in deployments where ZooKeeper is secure. All the security considerations described under SCRAM also apply to delegation tokens.

The master key used by brokers for generating tokens must be protected using encryption or by externalizing the key in a secure password store. Short-lived delegation tokens can be used to limit exposure if a token is compromised. Reauthentication can be enabled in brokers to prevent connections operating with expired tokens and to limit the amount of time existing connections may continue to operate after token deletion.

Reauthentication

As we saw earlier, Kafka brokers perform client authentication when a connection is established by the client. Client credentials are verified by the brokers, and the connection authenticates successfully if the credentials are valid at that time. Some security mechanisms like Kerberos and OAuth use credentials with a limited lifetime. Kafka uses a background login thread to acquire new credentials before the old ones expire, but the new credentials are used only to authenticate new connections by default. Existing connections that were authenticated with old credentials continue to process requests until disconnection occurs due to a request timeout, an idle timeout, or network errors. Long-lived connections may continue to process requests long after the credentials used to authenticate the connections expire. Kafka brokers support reauthentication for connections authenticated using SASL using the configuration option `connections.max.reauth.ms`. When this option is set to a positive

integer, Kafka brokers determine the session lifetime for SASL connections and inform clients of this lifetime during the SASL handshake. Session lifetime is the lower of the remaining lifetime of the credential or `connections.max.reauth.ms`. Any connection that doesn't reauthenticate within this interval is terminated by the broker. Clients perform reauthentication using the latest credentials acquired by the background login thread or injected using custom callbacks. Reauthentication can be used to tighten security in several scenarios:

- For SASL mechanisms like GSSAPI and OAUTHBEARER that use credentials with a limited lifetime, reauthentication guarantees that all active connections are associated with valid credentials. Short-lived credentials limit exposure in case credentials that are compromised.
- Password-based SASL mechanisms like PLAIN and SCRAM can support password rotation by adding periodic login. Reauthentication limits the amount of time requests are processed on connections authenticated with the old password. Custom server callback that allows both old and new passwords for a period of time can be used to avoid outages until all clients migrate to the new password.
- `connections.max.reauth.ms` forces reauthentication in all SASL mechanisms, including those with nonexpiring credentials. This limits the amount of time a credential may be associated with an active connection after it has been revoked.
- Connections from clients without SASL reauthentication support are terminated on session expiry, forcing the clients to reconnect and authenticate again, thus providing the same security guarantees for expired or revoked credentials.



Compromised Users

If a user is compromised, action must be taken to remove the user from the system as soon as possible. All new connections will fail to authenticate with Kafka brokers once the user is removed from the authentication server. Existing connections will continue to process requests until the next reauthentication timeout. If `connections.max.reauth.ms` is not configured, no timeout is applied and existing connections may continue to use the compromised user's identity for a long time. Kafka does not support SSL renegotiation due to known vulnerabilities during renegotiation in older SSL protocols. Newer protocols like TLSv1.3 do not support renegotiation. So, existing SSL connections may continue to use revoked or expired certificates. Deny ACLs for the user principal can be used to prevent these connections from performing any operation. Since ACL changes are applied with very small latencies across all brokers, this is the quickest way to disable access for compromised users.

Security Updates Without Downtime

Kafka deployments need regular maintenance to rotate secrets, apply security fixes, and update to the latest security protocols. Many of these maintenance tasks are performed using rolling updates where one by one, brokers are shut down and restarted with an updated configuration. Some tasks like updating SSL key stores and trust stores can be performed using dynamic config updates without restarting brokers.

When adding a new security protocol to an existing deployment, a new listener can be added to brokers with the new protocol while retaining the old listener with the old protocol to ensure that client applications can continue to function using the old listener during the update. For example, the following sequence can be used to switch from PLAINTEXT to SASL_SSL in an existing deployment:

1. Add a new listener on a new port to each broker using the Kafka configs tool. Use a single config update command to update `listeners` and `advertised.listeners` to include the old listener as well as the new listener, and provide all the configuration options for the new SASL_SSL listener with the listener prefix.
2. Modify all client applications to use the new SASL_SSL listener.
3. If inter-broker communication is being updated to use the new SASL_SSL listener, perform a rolling update of brokers with the new `inter.broker.listener.name`.
4. Use the configs tool to remove the old listener from `listeners` and `advertised.listeners` and to remove any unused configuration options of the old listener.

SASL mechanisms can be added or removed from existing SASL listeners without downtime using rolling updates on the same listener port. The following sequence switches the mechanism from PLAIN to SCRAM-SHA-256:

1. Add all existing users to the SCRAM store using the Kafka configs tool.
2. Set `sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256`, configure `listener.name.<_listener-name_>.scram-sha-256.sasl.jaas.config` for the listener, and perform a rolling update of brokers.
3. Modify all client applications to use `sasl.mechanism=SCRAM-SHA-256` and update `sasl.jaas.config` to use SCRAM.
4. If the listener is used for inter-broker communication, use a rolling update of brokers to set `sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256`.
5. Perform another rolling update of brokers to remove the PLAIN mechanism. Set `sasl.enabled.mechanisms=SCRAM-SHA-256` and remove `listener.name`.

`<listener-name>.plain.sasl.jaas.config` and any other configuration options for PLAIN.

Encryption

Encryption is used to preserve data privacy and data integrity. As we discussed earlier, Kafka listeners using SSL and SASL_SSL security protocols use TLS as the transport layer, providing secure encrypted channels that protect data transmitted over an insecure network. TLS cipher suites can be restricted to strengthen security and adhere to security requirements like the Federal Information Processing Standard (FIPS).

Additional measures must be taken to protect data at rest to ensure that sensitive data cannot be retrieved even by users with physical access to the disk that stores Kafka logs. To avoid security breaches even if the disk is stolen, physical storage can be encrypted using whole disk encryption or volume encryption.

While encryption of transport layer and data storage may provide adequate protection in many deployments, additional protection may be required to avoid granting automatic data access to platform administrators. Unencrypted data present in broker memory may appear in heap dumps, and administrators with direct access to the disk will be able to access these, as well as Kafka logs containing potentially sensitive data. In deployments with highly sensitive data or Personally Identifiable Information (PII), extra measures are required to preserve data privacy. To comply with regulatory requirements, especially in cloud deployments, it is necessary to guarantee that confidential data cannot be accessed by platform administrators or cloud providers by any means. Custom encryption providers can be plugged into Kafka clients to implement end-to-end encryption that guarantees that the entire data flow is encrypted.

End-to-End Encryption

In [Chapter 3](#) on Kafka producers, we saw that *serializers* are used to convert messages into the byte array stored in Kafka logs, and in [Chapter 4](#) on Kafka consumers, we saw that *deserializers* converted the byte array back to the message. Serializers and deserializers can be integrated with an encryption library to perform encryption of the message during serialization, and decryption during deserialization. Message encryption is typically performed using symmetric encryption algorithms like AES. A shared encryption key stored in a key management system (KMS) enables producers to encrypt the message and consumers to decrypt the message. Brokers do not require access to the encryption key and never see the unencrypted contents of the message, making this approach safe to use in cloud environments. Encryption parameters that are required to decrypt the message may be stored in message headers or in the message payload if older consumers without header support need access

to the message. A digital signature may also be included in message headers to verify message integrity.

Figure 11-2 shows a Kafka data flow with end-to-end encryption.

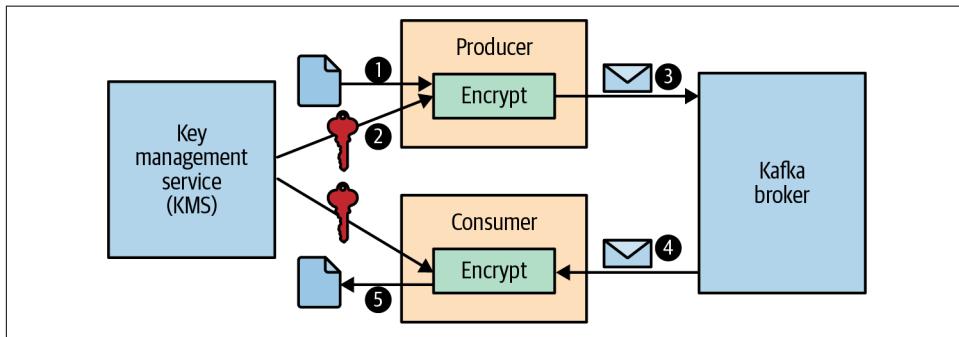


Figure 11-2. End-to-end encryption

1. We send a message using a Kafka producer.
2. The producer uses an encryption key from KMS to encrypt the message.
3. The encrypted message is sent to the broker. The broker stores the encrypted message in the partition logs.
4. The broker sends the encrypted message to consumers.
5. The consumer uses the encryption key from KMS to decrypt the message.

Producers and consumers must be configured with credentials to obtain shared keys from KMS. Periodic key rotation is recommended to harden security, since frequent rotation limits the number of compromised messages in case of a breach and also protects against brute-force attacks. Consumption must be supported with both old and new keys during the retention period of messages encrypted with the old key. Many KMS systems support graceful key rotation out of the box for symmetric encryption without requiring any special handling in Kafka clients. For compacted topics, messages encrypted with old keys may be retained for a long time, and it may be necessary to re-encrypt old messages. To avoid interference with newer messages, producers and consumers must be offline during this process.



Compression of Encrypted Messages

Compressing messages after encryption is unlikely to provide any benefit in terms of space reduction compared to compressing prior to encryption. Serializers may be configured to perform compression before encrypting the message, or applications may be configured to perform compression prior to producing messages. In either case, it is better to disable compression in Kafka since it adds overhead without providing any additional benefit. For messages transmitted over an insecure transport layer, known security exploits of compressed encrypted messages must also be taken into account.

In many environments, especially when TLS is used as the transport layer, message keys do not require encryption since they typically do not contain sensitive data like message payloads. But in some cases, clear-text keys may not comply with regulatory requirements. Since message keys are used for partitioning and compaction, transformation of keys must preserve the required hash equivalence to ensure that a key retains the same hash value even if encryption parameters are altered. One approach would be to store a secure hash of the original key as the message key and store the encrypted message key in the message payload or in a header. Since Kafka serializes message key and value independently, a producer interceptor can be used to perform this transformation.

Authorization

Authorization is the process that determines what operations you are allowed to perform on which resources. Kafka brokers manage access control using a customizable authorizer. We saw earlier that whenever connections are established from a client to a broker, the broker authenticates the client and associates a `KafkaPrincipal` that represents the client identity with the connection. When a request is processed, the broker verifies that the principal associated with the connection is authorized to perform that request. For example, when Alice's producer attempts to write a new customer order record to the topic `customerOrders`, the broker verifies that `User:Alice` is authorized to write to that topic.

Kafka has a built-in authorizer, `AclAuthorizer`, that can be enabled by configuring the authorizer class name as follows:

```
authorizer.class.name=kafka.security.authorizer.AclAuthorizer
```



SimpleAclAuthorizer

`AclAuthorizer` was introduced in Apache Kafka 2.3. Older versions from 0.9.0.0 onward had a built-in authorizer, `kafka.security.auth.SimpleAclAuthorizer`, which has been deprecated but is still supported.

AclAuthorizer

`AclAuthorizer` supports fine-grained access control for Kafka resources using access control lists (ACLs). ACLs are stored in ZooKeeper and cached in memory by every broker to enable high-performance lookup for authorizing requests. ACLs are loaded into the cache when the broker starts up, and the cache is kept up-to-date using notifications based on a ZooKeeper watcher. Every Kafka request is authorized by verifying that the `KafkaPrincipal` associated with the connection has permissions to perform the requested operation on the requested resources.

Each ACL binding consists of:

- Resource type: `Cluster|Topic|Group|TransactionalId|DelegationToken`
- Pattern type: `Literal|Prefixed`
- Resource name: Name of the resource or prefix, or the wildcard `*`
- Operation: `Describe|Create|Delete|Alter|Read|Write|DescribeConfigs|AlterConfigs`
- Permission type: `Allow|Deny`; Deny has higher precedence.
- Principal: Kafka principal represented as `<principalType>:<principalName>`, e.g., `User:Bob` or `Group:Sales`. ACLs may use `User:*` to grant access to all users.
- Host: Source IP address of the client connection or `*` if all hosts are authorized.

For example, an ACL may specify:

```
User:Alice has Allow permission for Write to Prefixed Topic:customer from  
192.168.0.1
```

`AclAuthorizer` authorizes an action if there are no Deny ACLs that match the action and there is at least one Allow ACL that matches the action. `Describe` permission is implicitly granted if `Read`, `Write`, `Alter`, or `Delete` permission is granted. `DescribeConfigs` permission is implicitly granted if `AlterConfigs` permission is granted.



Wildcard ACLs

ACLs with pattern type `Literal` and resource name `*` are used as wildcard ACLs that match all resource names of a resource type.

Brokers must be granted `Cluster:ClusterAction` access in order to authorize controller requests and replica fetch requests. Producers require `Topic:Write` for producing to a topic. For idempotent produce without transactions, producers must also be granted `Cluster:IdempotentWrite`. Transactional producers require `TransactionalId:Write` access to the transaction ID and `Group:Read` for consumer groups to commit offsets. Consumers require `Topic:Read` to consume from a topic and `Group:Read` for the consumer group if using group management or offset management. Administrative operations require appropriate `Create`, `Delete`, `Describe`, `Alter`, `DescribeConfigs`, or `AlterConfigs` access. [Table 11-1](#) shows the Kafka requests to which each ACL is applied.

Table 11-1. Access granted for each Kafka ACL

ACL	Kafka requests	Notes
<code>Cluster:ClusterAction</code>	Inter-broker requests, including controller requests and follower fetch requests for replication	Should only be granted to brokers.
<code>Cluster:Create</code>	<code>CreateTopics</code> and auto-topic creation	Use <code>Topic:Create</code> for fine-grained access control to create specific topics.
<code>Cluster:Alter</code>	<code>CreateAcls</code> , <code>DeleteAcls</code> , <code>AlterReplicaLogDirs</code> , <code>ElectReplicaLeader</code> , <code>AlterPartitionReassignments</code>	
<code>Cluster:AlterConfigs</code>	<code>AlterConfigs</code> and <code>IncrementalAlterConfigs</code> for broker and broker logger, <code>AlterClientQuotas</code>	
<code>Cluster:Describe</code>	<code>DescribeAcls</code> , <code>DescribeLogDirs</code> , <code>ListGroups</code> , <code>ListPartitionReassignments</code> , describing authorized operations for cluster in Metadata request	Use <code>Group:Describe</code> for fine-grained access control for <code>ListGroups</code> .
<code>Cluster:DescribeConfigs</code>	<code>DescribeConfigs</code> for broker and broker logger, <code>DescribeClientQuotas</code>	
<code>Cluster:IdempotentWrite</code>	<code>IdempotentInitProducerId</code> and <code>Produce</code> requests	Only required for nontransactional idempotent producers.
<code>Topic:Create</code>	<code>CreateTopics</code> and auto-topic creation	
<code>Topic:Delete</code>	<code>DeleteTopics</code> , <code>DeleteRecords</code>	
<code>Topic:Alter</code>	<code>CreatePartitions</code>	
<code>Topic:AlterConfigs</code>	<code>AlterConfigs</code> and <code>IncrementalAlterConfigs</code> for topics	
<code>Topic:Describe</code>	Metadata request for topic, <code>OffsetForLeaderEpoch</code> , <code>ListOffset</code> , <code>OffsetFetch</code>	
<code>Topic:DescribeConfigs</code>	<code>DescribeConfigs</code> for topics, for returning configs in <code>CreateTopics</code> response	

ACL	Kafka requests	Notes
Topic:Read	Consumer Fetch, OffsetCommit, TxnOffsetCommit, OffsetDelete	Should be granted to consumers.
Topic:Write	Produce, AddPartitionToTxn	Should be granted to producers.
Group:Read	JoinGroup, SyncGroup, LeaveGroup, Heartbeat, OffsetCommit, AddOffsetsToTxn, TxnOffsetCommit	Required for consumers using consumer group management or Kafka-based offset management. Also required for transactional producers to commit offsets within a transaction.
Group:Describe	FindCoordinator, DescribeGroup, ListGroups, OffsetFetch	
Group:Delete	DeleteGroups, OffsetDelete	
TransactionalId:Write	Produce and InitProducerId with transactions, AddPartitionToTxn, AddOffsetsToTxn, TxnOffsetCommit, EndTxn	Required for transactional producers.
TransactionalId:Describe	FindCoordinator for transaction coordinator	
DelegationToken:Describe	DescribeTokens	

Kafka provides a tool for managing ACLs using the authorizer configured in brokers. ACLs can be created directly in ZooKeeper as well. This is useful to create broker ACLs prior to starting brokers:

```
$ bin/kafka-acls.sh --add --cluster --operation ClusterAction \
    --authorizer-properties zookeeper.connect=localhost:2181 \
    --allow-principal User:kafka
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
    --command-config admin.props --add --topic customerOrders \
    --producer --allow-principal User:Alice
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
    --command-config admin.props --add --resource-pattern-type PREFIXED \
    --topic customer --operation Read --allow-principal User:Bob
```

- ① ACLs for broker user are created directly in ZooKeeper.
- ② By default, the ACLs command grants literal ACLs. `User:Alice` is granted access to write to the topic `customerOrders`.
- ③ The prefixed ACL grants permission for Bob to read all topics starting with `customer`.

`AclAuthorizer` has two configuration options to grant broad access to resources or principals in order to simplify management of ACLs, especially when adding authorization to existing clusters for the first time:

```
super.users=User:Carol;User:Admin  
allow.everyone.if.no.acl.found=true
```

Super users are granted access for all operations on all resources without any restrictions and cannot be denied access using Deny ACLs. If Carol's credentials are compromised, Carol must be removed from `super.users`, and brokers must be restarted to apply the changes. It is safer to grant specific access using ACLs to users in production systems to ensure access can be revoked easily, if required.



Super User Separator

Unlike other list configurations in Kafka that are comma-separated, `super.users` are separated by a semicolon since user principals such as distinguished names from SSL certificates often contain commas.

If `allow.everyone.if.no.acl.found` is enabled, all users are granted access to resources without any ACLs. This option may be useful when enabling authorization for the first time in a cluster or during development, but is not suitable for production use since access may be granted unintentionally to new resources. Access may also be unexpectedly removed when ACLs for a matching prefix or wildcard are added if the condition for `no.acl.found` no longer applies.

Customizing Authorization

Authorization can be customized in Kafka to implement additional restrictions or add new types of access control, like role-based access control.

The following custom authorizer restricts usage of some requests to the internal listener alone. For simplicity, the requests and listener name are hard-coded here, but they can be configured using custom authorizer properties instead for flexibility:

```
public class CustomAuthorizer extends AclAuthorizer {  
    private static final Set<Short> internalOps =  
        Utils.mkSet(CREATE_ACLS.id, DELETE_ACLS.id);  
    private static final String internalListener = "INTERNAL";  
  
    @Override  
    public List<AuthorizationResult> authorize(  
        AuthorizableRequestContext context, List<Action> actions) {  
        if (!context.listenerName().equals(internalListener) && ①  
            internalOps.contains((short) context.requestType()))  
            return Collections.nCopies(actions.size(), DENIED);  
        else
```

```

        return super.authorize(context, actions); ②
    }
}

```

- ① Authorizers are given the request context with metadata that includes listener names, security protocol, request types, etc., enabling custom authorizers to add or remove restrictions based on the context.
- ② We reuse functionality from the built-in Kafka authorizer using the public API.

Kafka authorizer can also be integrated with external systems to support group-based access control or role-based access control. Different principal types can be used to create ACLs for group principals or role principals. For instance, roles and groups from an LDAP server can be used to periodically populate `groups` and `roles` in the Scala class below to support Allow ACLs at different levels:

```

class RbacAuthorizer extends AclAuthorizer {

    @volatile private var groups = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
        .withDefaultValue(Set.empty) ①
    @volatile private var roles = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
        .withDefaultValue(Set.empty) ②

    override def authorize(context: AuthorizableRequestContext,
        actions: util.List[Action]): util.List[AuthorizationResult] = {
        val principals = groups(context.principal) + context.principal
        val allPrincipals = principals.flatMap(roles) ++ principals ③
        val contexts = allPrincipals.map(authorizeContext(context, _))
        actions.asScala.map { action =>
            val authorized = contexts.exists(
                super.authorize(_, List(action).asJava).get(0) == ALLOWED)
            if (authorized) ALLOWED else DENIED ④
        }.asJava
    }

    private def authorizeContext(context: AuthorizableRequestContext,
        contextPrincipal: KafkaPrincipal): AuthorizableRequestContext = {
        new AuthorizableRequestContext { ⑤
            override def principal() = contextPrincipal
            override def clientId() = context.clientId
            override def requestType() = context.requestType
            override def requestVersion() = context.requestVersion
            override def correlationId() = context.correlationId
            override def securityProtocol() = context.securityProtocol
            override def listenerName() = context.listenerName
            override def clientAddress() = context.clientAddress
        }
    }
}

```

- ❶ Groups to which each user belongs, populated from an external source like LDAP.
- ❷ Roles associated with each user, populated from an external source like LDAP.
- ❸ We perform authorization for the user as well as for all the groups and roles of the user.
- ❹ If any of the contexts are authorized, we return ALLOWED. Note that this example doesn't support Deny ACLs for groups or roles.
- ❺ We create an authorization context for each principal with the same metadata as the original context.

ACLs can be assigned for the group `Sales` or the role `Operator` using the standard Kafka ACL tool:

```
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
--command-config admin.props --add --topic customer --producer \
--resource-pattern-type PREFIXED --allow-principal Group:Sales ❶
$ bin/kafka-acls.sh --bootstrap-server localhost:9092 \
--command-config admin.props --add --cluster --operation Alter \
--allow-principal=Role:Operator ❷
```

- ❶ We use the principal `Group:Sales` with the custom principal type `Group` to create an ACL that applies to users belonging to the group `Sales`.
- ❷ We use the principal `Role:Operator` with the custom principal type `Role` to create an ACL that applies to users with the role `Operator`.

Security Considerations

Since `AclAuthorizer` stores ACLs in ZooKeeper, access to ZooKeeper should be restricted. Deployments without a secure ZooKeeper can implement custom authorizers to store ACLs in a secure external database.

In large organizations with a large number of users, managing ACLs for individual resources may become very cumbersome. Reserving different resource prefixes for different departments enables the use of prefixed ACLs that minimize the number of ACLs required. This can be combined with group- or role-based ACLs, as shown in the example earlier, to further simplify access control in large deployments.

Restricting user access using the principle of least privilege can limit exposure if a user is compromised. This means granting access only to the resources necessary for each user principal to perform their operations, and removing ACLs when they are no longer required. ACLs should be removed immediately when a user principal is

no longer in use, for instance, when a person leaves the organization. Long-running applications can be configured with service credentials rather than credentials associated with a specific user to avoid any disruption when employees leave the organization. Since long-lived connections with a user principal may continue to process requests even after the user has been removed from the system, Deny ACLs can be used to ensure that the principal is not unintentionally granted access through ACLs with wildcard principals. Reuse of principals must be avoided if possible to prevent access from being granted to connections using the older version of a principal.

Auditing

Kafka brokers can be configured to generate comprehensive *log4j* logs for auditing and debugging. The logging level as well as the appenders used for logging and their configuration options can be specified in *log4j.properties*. The logger instances `kafka.authorizer.logger` used for authorization logging and `kafka.request.logger` used for request logging can be configured independently to customize the log level and retention for audit logging. Production systems can use frameworks like the Elastic Stack to analyze and visualize these logs.

Authorizers generate INFO-level log entries for every attempted operation for which access was denied, and log entries at the DEBUG level for every operation for which access was granted. For example:

```
DEBUG Principal = User:alice is Allowed Operation = Write from host = 127.0.0.1  
on resource = Topic:LITERAL:customerOrders for request = Produce with resourceR-  
efCount = 1 (kafka.authorizer.logger)  
INFO Principal = User:Mallory is Denied Operation = Describe from host =  
10.0.0.13 on resource = Topic:LITERAL:customerOrders for request = Metadata  
with resourceRefCount = 1 (kafka.authorizer.logger)
```

Request logging generated at the DEBUG level also includes details of the user principal and client host. Full details of the request are included if the request logger is configured to log at the TRACE level. For example:

```
DEBUG Completed request:RequestHeader(apiKey=PRODUCE, apiVersion=8,  
clientId=producer-1, correlationId=6) --  
{acks=-1,timeout=30000,partitionSizes=[customerOrders-0=15514]},response:  
{responses=[{topic=customerOrders,partition_responses=[{partition=0,error_code=0  
,base_offset=13,log_append_time=-1,log_start_offset=0,record_errors=[],error_mes-  
sage=null}]}],throttle_time_ms=0} from connection  
127.0.0.1:9094-127.0.0.1:61040-0;totalTime:2.42,requestQueueTime:0.112,local-  
Time:2.15,remoteTime:0.0,throttleTime:0,responseQueueTime:0.04,sendTime:  
0.118,securityProtocol:SASL_SSL,principal:User:Alice,listener:SASL_SSL,clientInf-  
ormation:ClientInformation(softwareName=apache-kafka-java,  
softwareVersion=2.7.0-SNAPSHOT) (kafka.request.logger)
```

Authorizer and request logs can be analyzed to detect suspicious activities. Metrics that track authentication failures, as well as authorization failure logs, can be

extremely useful for auditing and provide valuable information in the event of an attack or unauthorized access. For end-to-end auditability and traceability of messages, audit metadata can be included in message headers when messages are produced. End-to-end encryption can be used to protect the integrity of this metadata.

Securing ZooKeeper

ZooKeeper stores Kafka metadata that is critical for maintaining the availability of Kafka clusters, and hence it is vital to secure ZooKeeper in addition to securing Kafka. ZooKeeper supports authentication using SASL/GSSAPI for Kerberos authentication and SASL/DIGEST-MD5 for username/password authentication. ZooKeeper also added TLS support in 3.5.0, enabling mutual authentication as well as encryption of data in transit. Note that SASL/DIGEST-MD5 should only be used with TLS encryption and is not suitable for production use due to known security vulnerabilities.

SASL

SASL configuration for ZooKeeper is provided using the Java system property `java.security.auth.login.config`. The property must be set to a JAAS configuration file that contains a login section with the appropriate login module and its options for the ZooKeeper server. Kafka brokers must be configured with the client-side login section for ZooKeeper clients to talk to SASL-enabled ZooKeeper servers. The Server section that follows provides the JAAS configuration for the ZooKeeper server to enable Kerberos authentication:

```
Server {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true storeKey=true  
    keyTab="/path/to/zk.keytab"  
    principal="zookeeper/zk1.example.com@EXAMPLE.COM";  
};
```

To enable SASL authentication on ZooKeeper servers, configure authentication providers in the ZooKeeper configuration file:

```
authProvider.sasl=org.apache.zookeeper.server.auth.SASLAuthenticationProvider  
kerberos.removeHostFromPrincipal=true  
kerberos.removeRealmFromPrincipal=true
```



Broker Principal

By default, ZooKeeper uses the full Kerberos principal, e.g., `kafka/broker1.example.com@EXAMPLE.COM`, as the client identity. When ACLs are enabled for ZooKeeper authorization, ZooKeeper servers should be configured with `kerberos.removeHostFromPrincipal=true` and `kerberos.removeRealmFromPrincipal=true` to ensure that all brokers have the same principal.

Kafka brokers must be configured to authenticate to ZooKeeper using SASL with a JAAS configuration file that provides client credentials for the broker:

```
Client {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true storeKey=true  
    keyTab="/path/to/broker1.keytab"  
    principal="kafka/broker1.example.com@EXAMPLE.COM";  
};
```

SSL

SSL may be enabled on any ZooKeeper endpoint, including those that use SASL authentication. Like Kafka, SSL may be configured to enable client authentication, but unlike Kafka, connections with both SASL and SSL client authentication authenticate using both protocols and associate multiple principals with the connection. ZooKeeper authorizer grants access to a resource if any of the principals associated with the connection have access.

To configure SSL on a ZooKeeper server, a key store with the hostname of the server or a wildcarded host should be configured. If client authentication is enabled, a trust store to validate client certificates is also required:

```
secureClientPort=2181  
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory  
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider  
ssl.keyStore.location=/path/to/zk.ks.p12  
ssl.keyStore.password=zk-ks-password  
ssl.keyStore.type=PKCS12  
ssl.trustStore.location=/path/to/zk.ts.p12  
ssl.trustStore.password=zk-ts-password  
ssl.trustStore.type=PKCS12
```

To configure SSL for Kafka connections to ZooKeeper, brokers should be configured with a trust store to validate ZooKeeper certificates. If client authentication is enabled, a key store is also required:

```
zookeeper.ssl.client.enable=true  
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty  
zookeeper.ssl.keystore.location=/path/to/zkclient.ks.p12  
zookeeper.ssl.keystore.password=zkclient-ks-password
```

```
zookeeper.ssl.keystore.type=PKCS12
zookeeper.ssl.truststore.location=/path/to/zkclient.ts.p12
zookeeper.ssl.truststore.password=zkclient-ts-password
zookeeper.ssl.truststore.type=PKCS12
```

Authorization

Authorization can be enabled for ZooKeeper nodes by setting ACLs for the path. When brokers are configured with `zookeeper.set.acl=true`, the broker sets ACLs for ZooKeeper nodes when creating the node. By default, metadata nodes are readable by everyone but modifiable only by brokers. Additional ACLs may be added if required for internal admin users who may need to update metadata directly in ZooKeeper. Sensitive paths, like nodes containing SCRAM credentials, are not world-readable by default.

Securing the Platform

In the previous sections, we discussed the options for locking down access to Kafka and ZooKeeper in order to safeguard Kafka deployments. Security design for a production system should use a threat model that addresses security threats not just for individual components but also for the system as a whole. Threat models build an abstraction of the system and identify potential threats and the associated risks. Once the threats are evaluated, documented, and prioritized based on risks, mitigation strategies must be implemented for each potential threat to ensure that the whole system is protected. When assessing potential threats, it is important to consider external threats as well as insider threats. For systems that store Personally Identifiable Information (PII) or other sensitive data, additional measures to comply with regulatory policies must also be implemented. An in-depth discussion of standard threat modeling techniques is outside the scope of this chapter.

In addition to protecting data in Kafka and metadata in ZooKeeper using secure authentication, authorization, and encryption, extra steps must be taken to ensure that the platform is secure. Defenses may include network firewall solutions to protect the network and encryption to protect physical storage. Key stores, trust stores, and Kerberos keytab files that contain credentials used for authentication must be protected using filesystem permissions. Access to configuration files containing security-critical information like credentials must be restricted. Since passwords stored in clear-text in configuration files are insecure even if access is restricted, Kafka supports externalizing passwords in a secure store.

Password Protection

Customizable configuration providers can be configured for Kafka brokers and clients to retrieve passwords from a secure third-party password store. Passwords may

also be stored in encrypted form in configuration files with custom configuration providers that perform decryption.

The custom configuration provider that follows uses the tool gpg to decrypt broker or client properties stored in a file:

```
public class GpgProvider implements ConfigProvider {  
  
    @Override  
    public void configure(Map<String, ?> configs) {}  
  
    @Override  
    public ConfigData get(String path) {  
        try {  
            String passphrase = System.getenv("PASSPHRASE"); ①  
            String data = Shell.execCommand(  
                "gpg", "--decrypt", "--passphrase", passphrase, path); ②  
            Properties props = new Properties();  
            props.load(new StringReader(data)); ③  
            Map<String, String> map = new HashMap<>();  
            for (String name : props.stringPropertyNames())  
                map.put(name, props.getProperty(name));  
            return new ConfigData(map);  
        } catch (IOException e) {  
            throw new RuntimeException(e); ④  
        }  
    }  
  
    @Override  
    public ConfigData get(String path, Set<String> keys) { ⑤  
        ConfigData configData = get(path);  
        Map<String, String> data = configData.data().entrySet()  
            .stream().filter(e -> keys.contains(e.getKey()))  
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));  
        return new ConfigData(data, configData.ttl());  
    }  
  
    @Override  
    public void close() {}  
}
```

- ① We provide the passphrase for decoding passwords to the process in the environment variable PASSPHRASE.
- ② We decrypt the configs using gpg. The return value contains the full set of decrypted configs.
- ③ We parse the configs in `data` as Java properties.
- ④ We fail fast with a `RuntimeException` if an error is encountered.

- ⑤ Caller may request a subset of keys from the path; here we get all values and return the requested subset.

You may recall that in the section on SASL/PLAIN, we used standard Kafka configuration classes to load credentials from an external file. We can now encrypt that file using gpg:

```
gpg --symmetric --output credentials.props.gpg \
--passphrase "$PASSPHRASE" credentials.props
```

We now add indirect configs and config provider options to the original properties file so that Kafka clients load their credentials from the encrypted file:

```
username=${gpg:/path/to/credentials.props.gpg:username}
password=${gpg:/path/to/credentials.props.gpg:password}
config.providers=gpg
config.providers.gpg.class=com.example.GpgProvider
```

Sensitive broker configuration options can also be stored encrypted in ZooKeeper using the Kafka configs tool without using custom providers. The following command can be executed before starting brokers to store encrypted SSL key store passwords for brokers in ZooKeeper. The password encoder secret must be configured in each broker's configuration file to decrypt the value:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter \
--entity-type brokers --entity-name 0 --add-config \
'listener.name.external.ssl.keystore.password=server-ks-
password,password.encoder.secret=encoder-secret'
```

Summary

The frequency and scale of data breaches have been increasing over the last decade as cyberattacks have become increasingly sophisticated. In addition to the significant cost of isolating and resolving breaches and the cost of outages until security fixes have been applied, data breaches may also result in regulatory penalties and long-term damage to brand reputation. In this chapter, we explored the vast array of options available to guarantee the confidentiality, integrity, and availability of data stored in Kafka.

Going back to the example data flow at the start of this chapter, we reviewed the options available for different aspects of security throughout the flow:

Client authenticity

When Alice's client establishes connection to a Kafka broker, a listener using SASL or SSL with client authentication can verify that the connection is really from Alice and not an imposter. Reauthentication can be configured to limit exposure in case a user is compromised.

Server authenticity

Alice's client can verify that its connection is to the genuine broker using SSL with hostname validation or using SASL mechanisms with mutual authentication, like Kerberos or SCRAM.

Data privacy

Use of SSL to encrypt data in transit protects data from eavesdroppers. Disk or volume encryption protects data at rest even if the disk is stolen. For highly sensitive data, end-to-end encryption provides fine-grained data access control and ensures that cloud providers and platform administrators with physical access to network and disks cannot access the data.

Data integrity

SSL can be used to detect tampering of data over an insecure network. Digital signatures can be included in messages to verify integrity when using end-to-end encryption.

Access control

Every operation performed by Alice, Bob, and even brokers is authorized using a customizable authorizer. Kafka has a built-in authorizer that enables fine-grained access control using ACLs.

Auditability

Authorizer logs and request logs can be used to track operations and attempted operations for auditing and anomaly detection.

Availability

A combination of quotas and configuration options to manage connections can be used to protect brokers from denial-of-service attacks. ZooKeeper can be secured using SSL, SASL, and ACLs to ensure that the metadata needed to ensure the availability of Kafka brokers is secure.

With the wide choice of options available for security, choosing the appropriate options for each use case can be a daunting task. We reviewed the security concerns to consider for each security mechanism, and the controls and policies that can be adopted to limit the potential attack surface. We also reviewed the additional measures required to lock down ZooKeeper and the rest of the platform. The standard security technologies supported by Kafka and the various extension points to integrate with the existing security infrastructure in your organization enable you to build consistent security solutions to protect the whole platform.

Administering Kafka

Managing a Kafka cluster requires additional tooling to perform administrative changes to topics, configurations, and more. Kafka provides several command-line interface (CLI) utilities that are useful for making administrative changes to your clusters. The tools are implemented in Java classes, and a set of scripts are provided natively to call those classes properly. While these tools provide basic functions, you may find they are lacking for more complex operations or are unwieldy to use at larger scales. This chapter will describe only the basic tools that are available as part of the Apache Kafka open source project. More information about advanced tools that have been developed in the community, outside of the core project, can be found on the [Apache Kafka website](#).



Authorizing Admin Operations

While Apache Kafka implements authentication and authorization to control topic operations, default configurations do not restrict the use of these tools. This means that these CLI tools can be used without any authentication required, which will allow operations such as topic changes to be executed with no security check or audit. Always ensure that access to this tooling on your deployments is restricted to administrators only to prevent unauthorized changes.

Topic Operations

The `kafka-topics.sh` tool provides easy access to most topic operations. It allows you to create, modify, delete, and list information about topics in the cluster. While some topic configurations are possible through this command, they have been deprecated, and it is recommended to use the more robust method of using the

`kafka-config.sh` tool for configuration changes. To use the `kafka-topics.sh` command, you must provide the cluster connection string and port through the `--bootstrap-server` option. In the examples that follow, the cluster connect string is being run locally on one of the hosts in the Kafka cluster, and we will be using local host:`9092`.

Throughout this chapter, all the tools will be located in the directory `/usr/local/kafka/bin/`. The example commands in this section will assume you are in this directory or have added the directory to your \$PATH.



Check the Version

Many of the command-line tools for Kafka have a dependency on the version of Kafka running to operate correctly. This includes some commands that may store data in ZooKeeper rather than connecting to the brokers themselves. For this reason, it is important to make sure the version of the tools that you are using matches the version of the brokers in the cluster. The safest approach is to run the tools on the Kafka brokers themselves, using the deployed version.

Creating a New Topic

When creating a new topic through the `--create` command, there are several required arguments to create a new topic in a cluster. These arguments must be provided when using this command even though some of them may have broker-level defaults configured already. Additional arguments and configuration overrides are possible at this time, as well using the `--config` option, but are covered later in the chapter. Here is a list of the three required arguments:

`--topic`

The name of the topic that you wish to create.

`--replication-factor`

The number of replicas of the topic to maintain within the cluster.

`--partitions`

The number of partitions to create for the topic.



Good Topic Naming Practices

Topic names may contain alphanumeric characters, underscores, dashes, and periods; however, it is not recommended to use periods in topic names. Internal metrics inside of Kafka convert period characters to underscore characters (e.g., “topic.1” becomes “topic_1” in metrics calculations), which can result in conflicts in topic names.

Another recommendation is to avoid using a double underscore to start your topic name. By convention, topics internal to Kafka operations are created with a double underscore naming convention (like the `_consumer_offsets` topic, which tracks consumer group offset storage). As such it is not recommended to have topic names that begin with the double underscore naming convention to prevent confusion.

Creating a new topic is simple. Run `kafka-topics.sh` as follows:

```
# kafka-topics.sh --bootstrap-server <connection-string>:<port> --create --  
topic <string>  
--replication-factor <integer> --partitions <integer>  
#
```

The command will cause the cluster to create a topic with the specified name and number of partitions. For each partition, the cluster will select the specified number of replicas appropriately. This means that if the cluster is set up for rack-aware replica assignment, the replicas for each partition will be in separate racks. If rack-aware assignment is not desired, specify the `--disable-rack-aware` command-line argument.

For example, create a topic named “my-topic” with eight partitions that have two replicas each:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --create  
--topic my-topic --replication-factor 2 --partitions 8  
Created topic "my-topic".  
#
```



Using `if-exists` and `if-not-exists` Arguments Properly

When using `kafka-topics.sh` in automation, you may want to use the `--if-not-exists` argument while creating new topics that will not return an error if the topic already exists.

While an `--if-exists` argument is provided for the `--alter` command, using it is not recommended. Using this argument will cause the command to not return an error if the topic being changed does not exist. This can mask problems where a topic does not exist that should have been created.

Listing All Topics in a Cluster

The `--list` command lists all topics in a cluster. The list is formatted with one topic per line, in no particular order, which is useful for generating a full list of topics.

Here's an example of the `--list` option listing all topics in the cluster:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --list
__consumer_offsets
my-topic
other-topic
```

You'll notice the internal `__consumer_offsets` topic is listed here. Running the command with `--exclude-internal` will remove all topics from the list that begin with the double underscore mentioned earlier, which can be beneficial.

Describing Topic Details

It is also possible to get detailed information on one or more topics in the cluster. The output includes the partition count, topic configuration overrides, and a listing of each partition with its replica assignments. This can be limited to a single topic by providing a `--topic` argument to the command.

For example, describing our recently created "my-topic" in the cluster:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 8      ReplicationFactor: 2      Configs: seg
ment.bytes=1073741824
    Topic: my-topic Partition: 0      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 2      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 3      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 4      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 5      Leader: 0      Replicas: 0,1      Isr: 0,1
    Topic: my-topic Partition: 6      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 7      Leader: 0      Replicas: 0,1      Isr: 0,1
#
#
```

The `--describe` command also has several useful options for filtering the output. These can be helpful for diagnosing cluster issues more easily. For these commands we generally do not specify the `--topic` argument because the intention is to find all topics or partitions in a cluster that match the criteria. These options will not work with the `list` command. Here is a list of useful pairings to use:

--topics-with-overrides

This will describe only the topics that have configurations that differ from the cluster defaults.

--exclude-internal

The previously mentioned command will remove all topics from the list that begin with the double underscore naming convention.

The following commands are used to help find topic partitions that may have problems:

--under-replicated-partitions

This shows all partitions where one or more of the replicas are not in sync with the leader. This isn't necessarily bad, as cluster maintenance, deployments, and rebalances will cause under-replicated partitions (or URPs) but is something to be aware of.

--at-min-isr-partitions

This shows all partitions where the number of replicas, including the leader, exactly match the setting for minimum in-sync replicas (ISRs). These topics are still available for producer or consumer clients, but all redundancy has been lost, and they are in danger of becoming unavailable.

--under-min-isr-partitions

This shows all partitions where the number of ISRs is below the configured minimum for successful produce actions. These partitions are effectively in read-only mode and cannot be produced to.

--unavailable-partitions

This shows all topic partitions without a leader. This is a serious situation and indicates that the partition is offline and unavailable for producer or consumer clients.

Here's an example of finding topics that are at the minimum ISR settings. In this example, the topic is configured for a min-ISR of 1 and has a replication factor (RF) of 2. Host 0 is online, and host 1 has gone down for maintenance:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --at-min-isr-partitions
    Topic: my-topic Partition: 0    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 1    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 2    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 3    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 4    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 5    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 6    Leader: 0        Replicas: 0,1    Isr: 0
    Topic: my-topic Partition: 7    Leader: 0        Replicas: 0,1    Isr: 0
#
#
```

Adding Partitions

It is sometimes necessary to increase the number of partitions for a topic. Partitions are the way topics are scaled and replicated across a cluster. The most common reason to increase the partition count is to horizontally scale a topic across more brokers by decreasing the throughput for a single partition. Topics may also be increased if a consumer needs to expand to run more copies in a single consumer group since a partition can only be consumed by a single member in the group.

Following is an example of increasing the number of partitions for a topic named "my-topic" to 16 using the `--alter` command, followed by a verification that it worked:

```
# kafka-topics.sh --bootstrap-server localhost:9092  
--alter --topic my-topic --partitions 16  
  
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic  
Topic: my-topic PartitionCount: 16      ReplicationFactor: 2      Configs: seg  
ment.bytes=1073741824  
          Topic: my-topic Partition: 0      Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 2      Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 3      Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 4      Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 5      Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 6      Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 7      Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 8      Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 9      Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 10     Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 11     Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 12     Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 13     Leader: 0      Replicas: 0,1      Isr: 0,1  
          Topic: my-topic Partition: 14     Leader: 1      Replicas: 1,0      Isr: 1,0  
          Topic: my-topic Partition: 15     Leader: 0      Replicas: 0,1      Isr: 0,1  
#
```



Adjusting Keyed Topics

Topics that are produced with keyed messages can be very difficult to add partitions to from a consumer's point of view. This is because the mapping of keys to partitions will change when the number of partitions is changed. For this reason, it is advisable to set the number of partitions for a topic that will contain keyed messages once, when the topic is created, and avoid resizing the topic.

Reducing Partitions

It is not possible to reduce the number of partitions for a topic. Deleting a partition from a topic would cause part of the data in that topic to be deleted as well, which would be inconsistent from a client point of view. In addition, trying to redistribute the data to the remaining partitions would be difficult and result in out-of-order messages. Should you need to reduce the number of partitions, it is recommended to delete the topic and re-create it or (if deletion is not possible) create a new version of the existing topic and move all produce traffic to the new topic (e.g., “my-topic-v2”).

Deleting a Topic

Even a topic with no messages uses cluster resources such as disk space, open filehandles, and memory. The controller also has junk metadata that it must retain knowledge of, which can hinder performance at large scale. If a topic is no longer needed, it can be deleted to free up these resources. To perform this action, the brokers in the cluster must be configured with the `delete.topic.enable` option set to `true`. If it's set to `false`, then the request to delete the topic will be ignored and will not succeed.

Topic deletion is an asynchronous operation. This means that running this command will mark a topic for deletion, but the deletion may not happen immediately, depending on the amount of data and cleanup needed. The controller will notify the brokers of the pending deletion as soon as possible (after existing controller tasks complete), and the brokers will then invalidate the metadata for the topic and delete the files from disk. It is highly recommended that operators not delete more than one or two topics at a time, and give those ample time to complete before deleting other topics, due to limitations in the way the controller executes these operations. In the small cluster shown in the examples in this book, topic deletion will happen almost immediately, but in larger clusters it may take longer.



Data Loss Ahead

Deleting a topic will also delete all its messages. This is not a reversible operation. Make sure it is executed carefully.

Here is an example of deleting the topic named “my-topic” using the `--delete` argument. Depending on the version of Kafka, there will be a note letting you know that the argument will not work if another config is not set:

```
# kafka-topics.sh --bootstrap-server localhost:9092  
--delete --topic my-topic
```

Note: This will have no impact if `delete.topic.enable` is not set to `true`.

```
#
```

You will notice there is no visible feedback that the topic deletion was completed successfully or not. Verify that deletion was successful by running the `--list` or `--describe` options to see that the topic is no longer in the cluster.

Consumer Groups

Consumer groups are coordinated groups of Kafka consumers consuming from topics or multiple partitions of a single topic. The `kafka-consumer-groups.sh` tool helps manage and gain insight into the consumer groups that are consuming from topics in the cluster. It can be used to list consumer groups, describe specific groups, delete consumer groups or specific group info, or reset consumer group offset information.



ZooKeeper-Based Consumer Groups

In older versions of Kafka, consumer groups could be managed and maintained in ZooKeeper. This behavior was deprecated in versions 0.11.0.* and later, and old consumer groups are no longer used. Some versions of the provided scripts may still show deprecated `--zookeeper` connection string commands, but it is not recommended to use them unless you have an old environment with some consumer groups that have not upgraded to later versions of Kafka.

List and Describe Groups

To list consumer groups, use the `--bootstrap-server` and `--list` parameters. Ad hoc consumers utilizing the `kafka-consumer-groups.sh` script will show up as `console-consumer-<generated_id>` in the consumer list:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
console-consumer-95554
console-consumer-9581
my-consumer
#
```

For any group listed, you can get more details by changing the `--list` parameter to `--describe` and adding the `--group` parameter. This will list all the topics and partitions that the group is consuming from, as well as additional information such as the offsets for each topic partition. [Table 12-1](#) has a full description of all the fields provided in the output.

For example, get consumer group details for the ad hoc group named “my-consumer”:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--describe --group my-consumer
GROUP          TOPIC        PARTITION  CURRENT-OFFSET  LOG-END-OFFSET
LAG            CONSUMER-ID
HOST           CLIENT-ID
my-consumer    my-topic     0          2              4
2              consumer-1-029af89c-873c-4751-a720-cefd41a669d6  /
127.0.0.1      consumer-1
my-consumer    my-topic     1          2              3
1              consumer-1-029af89c-873c-4751-a720-cefd41a669d6  /
127.0.0.1      consumer-1
my-consumer    my-topic     2          2              3
1              consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2  /
127.0.0.1      consumer-2
#
#
```

Table 12-1. Fields provided for group named “my-consumer”

Field	Description
GROUP	The name of the consumer group.
TOPIC	The name of the topic being consumed.
PARTITION	The ID number of the partition being consumed.
CURRENT-OFFSET	The next offset to be consumed by the consumer group for this topic partition. This is the position of the consumer within the partition.
LOG-END-OFFSET	The current high-water mark offset from the broker for the topic partition. This is the offset of the next message to be produced to this partition.
LAG	The difference between the consumer Current-Offset and the broker Log-End-Offset for this topic partition.
CONSUMER-ID	A generated unique consumer-id based on the provided client-id.
HOST	Address of the host the consumer group is reading from.
CLIENT-ID	String provided by the client identifying the client that is consuming from the group.

Delete Group

Deletion of consumer groups can be performed with the `--delete` argument. This will remove the entire group, including all stored offsets for all topics that the group is consuming. To perform this action, all consumers in the group should be shut down as the consumer group must not have any active members. If you attempt to delete a group that is not empty, an error stating “The group is not empty” will be thrown and nothing will happen. It is also possible to use the same command to delete offsets for a single topic that the group is consuming without deleting the entire group by adding the `--topic` argument and specifying which topic offsets to delete.

Here is an example of deleting the entire consumer group named “my-consumer”:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete --group my-consumer
Deletion of requested consumer groups ('my-consumer') was successful.
#
```

Offset Management

In addition to displaying and deleting the offsets for a consumer group, it is also possible to retrieve the offsets and store new offsets in a batch. This is useful for resetting the offsets for a consumer when there is a problem that requires messages to be reread, or for advancing offsets and skipping past a message that the consumer is having a problem with (e.g., if there is a badly formatted message that the consumer cannot handle).

Export offsets

To export offsets from a consumer group to a CSV file, use the `--reset-offsets` argument with the `--dry-run` option. This will allow us to create an export of the current offsets in a file format that can be reused for importing or rolling back the offsets later. The CSV format export will be in the following configuration:

```
<topic-name>,<partition-number>,<offset>
```

Running the same command without the `--dry-run` option will reset the offsets completely, so be careful.

Here is an example of exporting the offsets for the topic “my-topic” that is being consumed by the consumer group named “my-consumer” to a file named `offsets.csv`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--export --group my-consumer --topic my-topic
--reset-offsets --to-current --dry-run > offsets.csv

# cat offsets.csv
my-topic,0,8905
my-topic,1,8915
my-topic,2,9845
my-topic,3,8072
my-topic,4,8008
my-topic,5,8319
my-topic,6,8102
my-topic,7,12739
#
```

Import offsets

The import offset tool is the opposite of exporting. It takes the file produced by exporting offsets in the previous section and uses it to set the current offsets for the

consumer group. A common practice is to export the current offsets for the consumer group, make a copy of the file (so that you preserve a backup), and edit the copy to replace the offsets with the desired values.



Stop Consumers First

Before performing this step, it is important that all consumers in the group are stopped. They will not read the new offsets if they are written while the consumer group is active. The consumers will just overwrite the imported offsets.

In the following example, we import the offsets for the consumer group named “my-consumer” from the file we created in the last example named *offsets.csv*:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--reset-offsets --group my-consumer
--from-file offsets.csv --execute
    TOPIC          PARTITION  NEW-OFFSET
    my-topic        0          8905
    my-topic        1          8915
    my-topic        2          9845
    my-topic        3          8072
    my-topic        4          8008
    my-topic        5          8319
    my-topic        6          8102
    my-topic        7          12739
#
#
```

Dynamic Configuration Changes

There is a plethora of configurations for topics, clients, brokers, and more that can be updated dynamically during runtime without having to shut down or redeploy a cluster. The `kafka-configs.sh` is the main tool for modifying these configs. Currently there are four main categories, or *entity-types*, of dynamic config changes that can be made: *topics*, *brokers*, *users*, and *clients*. For each entity-type there are specific configurations that can be overridden. New dynamic configs are being added constantly with each release of Kafka, so it is good to ensure you have the same version of this tool that matches the version of Kafka you are running. For ease of setting up these configs consistently via automation, the `--add-config-file` argument can be used with a preformatted file of all the configs you want to manage and update.

Overriding Topic Configuration Defaults

There are many configurations that are set by default for topics that are defined in the static broker configuration files (e.g., retention time policy). With dynamic configurations, we can override the cluster-level defaults for individual topics to accommodate

different use cases within a single cluster. **Table 12-2** shows the valid configuration keys for topics that can be altered dynamically.

The format of the command to change a topic configuration is:

```
kafka-configs.sh --bootstrap-server localhost:9092  
--alter --entity-type topics --entity-name <topic-name>  
--add-config <key>=<value>[,<key>=<value>...]
```

Here is an example of setting the retention for the topic named “my-topic” to 1 hour (3,600,000 ms):

```
# kafka-configs.sh --bootstrap-server localhost:9092  
--alter --entity-type topics --entity-name my-topic  
--add-config retention.ms=3600000  
Updated config for topic: "my-topic".  
#
```

Table 12-2. Valid keys for topics

Configuration key	Description
cleanup.policy	If set to compact, the messages in this topic will be discarded and only the most recent message with a given key is retained (log compacted).
compression.type	The compression type used by the broker when writing message batches for this topic to disk.
delete.retention.ms	How long, in milliseconds, deleted tombstones will be retained for this topic. Only valid for log compacted topics.
file.delete.delay.ms	How long, in milliseconds, to wait before deleting log segments and indices for this topic from disk.
flush.messages	How many messages are received before forcing a flush of this topic’s messages to disk.
flush.ms	How long, in milliseconds, before forcing a flush of this topic’s messages to disk.
follower.replication.throttled.replicas	A list of replicas for which log replication should be throttled by the follower.
index.interval.bytes	How many bytes of messages can be produced between entries in the log segment’s index.
leader.replication.throttled.replica	A list of replicas for which log replication should be throttled by the leader.
max.compaction.lag.ms	Maximum time limit a message won’t be eligible for compaction in the log.
max.message.bytes	The maximum size of a single message for this topic, in bytes.
message.downconversion.enable	Allows the message format version to be down-converted to the previous version if enabled with some overhead.
message.format.version	The message format version that the broker will use when writing messages to disk. Must be a valid API version number.
message.timestamp.difference.max.ms	The maximum allowed difference, in milliseconds, between the message timestamp and the broker timestamp when the message is received. This is only valid if the message.timestamp.type is set to CreateTime.

Configuration key	Description
<code>message.timestamp.type</code>	Which timestamp to use when writing messages to disk. Current values are <code>CreateTime</code> for the timestamp specified by the client and <code>LogAppendTime</code> for the time when the message is written to the partition by the broker.
<code>min.cleanable.dirty.ratio</code>	How frequently the log compactor will attempt to compact partitions for this topic, expressed as a ratio of the number of uncompacted log segments to the total number of log segments. Only valid for log compacted topics.
<code>min.compaction.lag.ms</code>	Minimum time a message will remain uncompacted in the log.
<code>min.insync.replicas</code>	The minimum number of replicas that must be in sync for a partition of the topic to be considered available.
<code>preallocate</code>	If set to <code>true</code> , log segments for this topic should be preallocated when a new segment is rolled.
<code>retention.bytes</code>	The amount of messages, in bytes, to retain for this topic.
<code>retention.ms</code>	How long messages should be retained for this topic, in milliseconds.
<code>segment.bytes</code>	The amount of messages, in bytes, that should be written to a single log segment in a partition.
<code>segment.index.bytes</code>	The maximum size, in bytes, of a single log segment index.
<code>segment.jitter.ms</code>	A maximum number of milliseconds that is randomized and added to <code>segment.ms</code> when rolling log segments.
<code>segment.ms</code>	How frequently, in milliseconds, the log segment for each partition should be rotated.
<code>unclean.leader.election.enable</code>	If set to <code>false</code> , unclean leader elections will not be permitted for this topic.

Overriding Client and User Configuration Defaults

For Kafka clients and users, there are only a few configurations that can be overridden, which are all essentially types of quotas. Two of the more common configurations to change are the bytes/sec rates allowed for producers and consumers with a specified client ID on a per-broker basis. The full list of shared configurations that can be modified for both *users* and *clients* is shown in [Table 12-3](#).



Uneven Throttling Behavior in Poorly Balanced Clusters

Because throttling occurs on a per-broker basis, even balance of leadership of partitions across a cluster becomes particularly important to enforce this properly. If you have 5 brokers in a cluster and you specify a producer quota of 10 MBps for a client, that client will be allowed to produce 10 MBps *on each* broker at the same time for a total of 50 MBps, assuming a balanced leadership across all 5 hosts. However, if leadership for every partition is all on broker 1, the same producer will only be able to produce a max of 10 MBps.

Table 12-3. The configurations (keys) for clients

Configuration key	Description
consumer_bytes_rate	The amount of messages, in bytes, that a single client ID is allowed to consume from a single broker in one second.
producer_bytes_rate	The amount of messages, in bytes, that a single client ID is allowed to produce to a single broker in one second.
controller_mutations_rate	The rate at which mutations are accepted for the create topics request, the create partitions request, and the delete topics request. The rate is accumulated by the number of partitions created or deleted.
request_percentage	The percentage per quota window (out of a total of $(\text{num.io.threads} + \text{num.network.threads}) \times 100\%$) for requests from the user or client.



Client ID Versus Consumer Group

The client ID is not necessarily the same as the consumer group name. Consumers can set their own client ID, and you may have many consumers that are in different groups that specify the same client ID. It is considered a best practice to set the client ID for each consumer group to something unique that identifies that group. This allows a single consumer group to share a quota, and it makes it easier to identify in logs what group is responsible for requests.

Compatible user and client config changes can be specified together for compatible configs that apply to both. Here is an example of the command to change the controller mutation rate for both a user and client in one configuration step:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --add-config "controller_mutations_rate=10"
--entity-type clients --entity-name <client ID>
--entity-type users --entity-name <user ID>
#
```

Overriding Broker Configuration Defaults

Broker- and cluster-level configs will primarily be configured statically in the cluster configuration files, but there is a plethora of configs that can be overridden during runtime without needing to redeploy Kafka. More than 80 overrides can be altered with `kafka-configs.sh` for brokers. As such, we will not list them all in this book, but they can be referenced by the `--help` command or found in the [open source documentation](#). A few important configs worth pointing out specifically are:

`min.insync.replicas`

Adjusts the minimum number of replicas that need to acknowledge a write for a produce request to be successful when producers have set `acks` to `all` (or `-1`).

`unclean.leader.election.enable`

Allows replicas to be elected as leader even if it results in data loss. This is useful when it is permissible to have some lossy data, or to turn on for short times to unstick a Kafka cluster if unrecoverable data loss cannot be avoided.

`max.connections`

The maximum number of connections allowed to a broker at any time. We can also use `max.connections.per.ip` and `max.connections.per.ip.overrides` for more fine-tuned throttling.

Describing Configuration Overrides

All configuration overrides can be listed using the `kafka-configs.sh` tool. This will allow you to examine the specific configuration for a topic, broker, or client. Similar to other tools, this is done using the `--describe` command.

In the following example, we can get all the configuration overrides for the topic named “my-topic,” which we observe is only the retention time:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--describe --entity-type topics --entity-name my-topic
Configs for topics:my-topic are
retention.ms=3600000
#
```



Topic Overrides Only

The configuration description will only show overrides—it does not include the cluster default configurations. There is not a way to dynamically discover the configuration of the brokers themselves. This means that when using this tool to discover topic or client settings in automation, the user must have separate knowledge of the cluster default configuration.

Removing Configuration Overrides

Dynamic configurations can be removed entirely, which will cause the entity to revert back to the cluster defaults. To delete a configuration override, use the `--alter` command along with the `--delete-config` parameter.

For example, delete a configuration override for `retention.ms` for a topic named “my-topic”:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

Producing and Consuming

While working with Kafka, you will often find it is necessary to manually produce or consume some sample messages in order to validate what's going on with your applications. Two utilities are provided to help with this, `kafka-console-consumer.sh` and `kafka-console-producer.sh`, which were touched upon briefly in [Chapter 2](#) to verify our installation. These tools are wrappers around the main Java client libraries that allow you to interact with Kafka topics without having to write an entire application to do it.



Piping Output to Another Application

While it is possible to write applications that wrap around the console consumer or producer (e.g., to consume messages and pipe them to another application for processing), this type of application is quite fragile and should be avoided. It is difficult to interact with the console consumer in a way that does not lose messages. Likewise, the console producer does not allow for using all features, and properly sending bytes is tricky. It is best to use either the Java client libraries directly or a third-party client library for other languages that use the Kafka protocol directly.

Console Producer

The `kafka-console-producer.sh` tool can be used to write messages into a Kafka topic in your cluster. By default, messages are read one per line, with a tab character separating the key and the value (if no tab character is present, the key is null). As with the console consumer, the producer reads in and produces raw bytes using the default serializer (which is `DefaultEncoder`).

The console producer requires that a minimum of two arguments are provided to know what Kafka cluster to connect to and which topic to produce to within that cluster. The first is the customary `--bootstrap-server` connection string we are used to using. When you are done producing, send an end-of-file (EOF) character to close the client. In most common terminals, this is done with Control-D.

Here we can see an example of producing four messages to a topic named "my-topic":

```
# kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic
>Message 1
>Test Message 2
>Test Message 3
>Message 4
>^D
#
```

Using producer configuration options

It is possible to pass normal producer configuration options to the console producer as well. This can be done in two ways, depending on how many options you need to pass and how you prefer to do it. The first is to provide a producer configuration file by specifying `--producer.config <config-file>`, where `<config-file>` is the full path to a file that contains the configuration options. The other way is to specify the options on the command line with one or more arguments of the form `--producer-property <key>=<value>`, where `<key>` is the configuration option name and `<value>` is the value to set it to. This can be useful for producer options like message-batching configurations (such as `linger.ms` or `batch.size`).



Confusing Command-Line Options

The `--property` command-line option is available for both the console producer and the console consumer, but this should not be confused with the `--producer-property` or `--consumer-property` options, respectively. The `--property` option is only used for passing configurations to the message formatter, and not the client itself.

The console producer has many command-line arguments available to use with the `--producer-property` option for adjusting its behavior. Some of the more useful options are:

`--batch-size`

Specifies the number of messages sent in a single batch if they are not being sent synchronously.

`--timeout`

If a producer is running in asynchronous mode, this provides the max amount of time waiting for the batch size before producing to avoid long waits on low-producing topics.

`--compression-codec <string>`

Specify the type of compression to be used when producing messages. Valid types can be one of the following: `none`, `gzip`, `snappy`, `zstd`, or `lz4`. The default value is `gzip`.

`--sync`

Produce messages synchronously, waiting for each message to be acknowledged before sending the next one.

Line-reader options

The `kafka.tools.ConsoleProducer$LineMessageReader` class, which is responsible for reading standard input and creating producer records, also has several useful options that can be passed to the console producer using the `--property` command-line option:

`ignore.error`

Set to `false` to throw an exception when `parse.key` is set to `true` and a key separator is not present. Defaults to `true`.

`parse.key`

Set to `false` to always set the key to null. Defaults to `true`.

`key.separator`

Specify the delimiter character to use between the message key and message value when reading. Defaults to a tab character.



Changing Line-Reading Behavior

You can provide your own class to Kafka for customized methods of reading lines. The class that you create must extend `kafka.common.MessageReader` and will be responsible for creating the `ProducerRecord`. Specify your class on the command line with the `--line-reader` option, and make sure the JAR containing your class is in the classpath. The default is `kafka.tools.ConsoleProducer$LineMessageReader`.

When producing messages, the `LineMessageReader` will split the input on the first instance of the `key.separator`. If there are no characters remaining after that, the value of the message will be empty. If no key separator character is present on the line, or if `parse.key` is false, the key will be null.

Console Consumer

The `kafka-console-consumer.sh` tool provides a means to consume messages out of one or more topics in your Kafka cluster. The messages are printed in standard output, delimited by a new line. By default, it outputs the raw bytes in the message, without the key, with no formatting (using the `DefaultFormatter`). Similar to the producer, there are a few basic options needed to get started: a connection string to the cluster, which topic you want to consume from, and the timeframe you want to consume.



Checking Tool Versions

It is very important to use a consumer that is the same version as your Kafka cluster. Older console consumers can potentially damage the cluster by interacting with the cluster or ZooKeeper in incorrect ways.

As in other commands, the connection string to the cluster will be the `--bootstrap-server` option; however, you can choose from two options for selecting the topics to consume:

`--topic`

Specifies a single topic to consume from.

`--whitelist`

A regular expression matching all topics to consume from (remember to properly escape the regex so that it is not processed improperly by the shell).

Only one of the previous options should be selected and used. Once the console consumer has started, the tool will continue to try and consume until the shell escape command is given (in this case, Ctrl-C). Here is an example of consuming all topics from our cluster that match the prefix `my` (of which there is only one in this example, “`my-topic`”):

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
Message 1
Test Message 2
Test Message 3
Message 4
^C
#
```

Using consumer configuration options

In addition to these basic command-line options, it is possible to pass normal consumer configuration options to the console consumer as well. Similar to the `kafka-console-producer.sh` tool, this can be done in two ways, depending on how many options you need to pass and how you prefer to do it. The first is to provide a consumer configuration file by specifying `--consumer.config <config-file>`, where `<config-file>` is the full path to a file that contains the configuration options. The other way is to specify the options on the command line with one or more arguments of the form `--consumer-property <key>=<value>`, where `<key>` is the configuration option name and `<value>` is the value to set it to.

There are a few other commonly used options for the console consumer that are helpful to know and be familiar with:

```
--formatter <classname>
    Specifies a message formatter class to be used to decode the messages. This
    defaults to kafka.tools.DefaultMessageFormatter.

--from-beginning
    Consume messages in the topic(s) specified from the oldest offset. Otherwise,
    consumption starts from the latest offset.

--max-messages <int>
    The maximum number of messages to consume before exiting.

--partition <int>
    Consume only from the partition with the ID given.

--offset
    The offset ID to consume from, if provided (<int>). Other valid options are
    earliest, which will consume from the beginning, and latest, which will start
    consuming from the most recent offset.

--skip-message-on-error
    Skip a message if there is an error when processing instead of halting. Useful for
    debugging.
```

Message formatter options

There are three message formatters available to use besides the default:

kafka.tools.LoggingMessageFormatter

Outputs messages using the logger, rather than standard out. Messages are
printed at the INFO level and include the timestamp, key, and value.

kafka.tools.ChecksumMessageFormatter

Prints only message checksums.

kafka.tools.NoOpMessageFormatter

Consumes messages but does not output them at all.

The following is an example of consuming the same messages from before but with
the kafka.tools.ChecksumMessageFormatter being used rather than the default:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
--formatter kafka.tools.ChecksumMessageFormatter
checksum:0
checksum:0
checksum:0
checksum:0
#
```

The `kafka.tools.DefaultMessageFormatter` also has several useful options that can be passed using the `--property` command-line option, shown in [Table 12-4](#).

Table 12-4. Message formatter properties

Property	Description
<code>print.timestamp</code>	Set to <code>true</code> to display the timestamp of each message (if available).
<code>print.key</code>	Set to <code>true</code> to display the message key in addition to the value.
<code>print.offset</code>	Set to <code>true</code> to display the message offset in addition to the value.
<code>print.partition</code>	Set to <code>true</code> to display the topic partition a message is consumed from.
<code>key.separator</code>	Specify the delimiter character to use between the message key and message value when printing.
<code>line.separator</code>	Specify the delimiter character to use between messages.
<code>key.deserializer</code>	Provide a class name that is used to deserialize the message key before printing.
<code>value.deserializer</code>	Provide a class name that is used to deserialize the message value before printing.

The deserializer classes must implement `org.apache.kafka.common.serialization.Deserializer`, and the console consumer will call the `toString` method on them to get the output to display. Typically, you would implement these deserializers as a Java class that you would insert into the classpath for the console consumer by setting the `CLASSPATH` environment variable before executing `kafka-console-consumer.sh`.

Consuming the offsets topics

It is sometimes useful to see what offsets are being committed for the cluster's consumer groups. You may want to see if a particular group is committing offsets at all, or how often offsets are being committed. This can be done by using the console consumer to consume the special internal topic called `__consumer_offsets`. All consumer offsets are written as messages to this topic. In order to decode the messages in this topic, you must use the formatter class `kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter`.

Putting all we have learned together, the following is an example of consuming the earliest message from the `__consumer_offsets` topic:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic __consumer_offsets --from-beginning --max-messages 1
--formatter "kafka.coordinator.group.GroupMetadataManager\$\$OffsetsMessageFormatter"
--consumer-property exclude.internal.topics=false
[my-group-name,my-topic,0]::[OffsetMetadata[1,NO_METADATA]
CommitTime 1623034799990 ExpirationTime 1623639599990]
Processed a total of 1 messages
#
```

Partition Management

A default Kafka installation also contains a few scripts for working with the management of partitions. One of these tools allows for the reelection of leader replicas; another is a low-level utility for assigning partitions to brokers. Together these tools can assist in situations where a more manual hands-on approach to balance message traffic within a cluster of Kafka brokers is needed.

Preferred Replica Election

As described in [Chapter 7](#), partitions can have multiple replicas for reliability. It is important to understand that only one of these replicas can be the leader for the partition at any given point in time, and all produce and consume operations happen on that broker. Maintaining a balance of which partition's replicas have leadership on which broker is necessary to ensure the load is spread out through a full Kafka cluster.

Leadership is defined within Kafka as the first in-sync replica in the replica list. However, when a broker is stopped or loses connectivity to the rest of the cluster, leadership is transferred to another in-sync replica, and the original does not resume leadership of any partitions automatically. This can cause wildly inefficient balance after a deployment across a full cluster if automatic leader balancing is not enabled. As such it is recommended to ensure that this setting is enabled or to use other open source tooling such as Cruise Control to ensure that a good balance is maintained at all times.

If you find that your Kafka cluster has a poor balance, a lightweight, generally non-impacting procedure can be performed called *preferred leader election*. This tells the cluster controller to select the ideal leader for partitions. Clients can track leadership changes automatically, so they will be able to move to the new broker in the cluster in which leadership is transferred. This operation can be manually triggered using the `kafka-leader-election.sh` utility. An older version of this tool called `kafka-preferred-replica-election.sh` is also available but has been deprecated in favor of the new tool, which allows for more customization, such as specifying whether we want a “preferred” or “unclean” election type.

As an example, starting a preferred leader election for all topics in a cluster can be executed with the following command:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --all-topic-partitions
#
```

It is also possible to start elections on specific partitions or topics. This can be done by passing in a topic name with the `--topic` option and a partition with the

--partition option directly. It is also possible to pass in a list of several partitions to be elected. This is done by configuring a JSON file that we will call *partitions.json*:

```
{  
  "partitions": [  
    {  
      "partition": 1,  
      "topic": "my-topic"  
    },  
    {  
      "partition": 2,  
      "topic": "foo"  
    }  
  ]  
}
```

In this example, we will start a preferred replica election with a specified list of partitions in a file named *partitions.json*:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092  
--election-type PREFERRED --path-to-json-file partitions.json  
#
```

Changing a Partition's Replicas

Occasionally it may be necessary to change the replica assignments manually for a partition. Some examples of when this might be needed are:

- There is an uneven load on brokers that the automatic leader distribution is not correctly handling.
- If a broker is taken offline and the partition is under replicated.
- If a new broker is added and we want to more quickly balance new partitions on it.
- You want to adjust the replication factor of a topic.

The `kafka-reassign-partitions.sh` can be used to perform this operation. This is a multistep process to generate a move set and then execute on the provided move set proposal. First, we want to use a broker list and a topic list to generate a proposal for the set of moves. This will require the generation of a JSON file with a list of topics to be supplied. The next step executes the moves that were generated by the previous proposal. Finally, the tool can be used with the generated list to track and verify the progress or completion of the partition reassessments.

Let's generate a hypothetical scenario in which you have a four-broker Kafka cluster. You've recently added two new brokers, bringing the total up to six, and you want to move two of your topics onto brokers 5 and 6.

To generate a set of partition moves, you must first create a file that contains a JSON object listing the topics. The JSON object is formatted as follows (the version number is currently always 1):

```
{  
    "topics": [  
        {  
            "topic": "foo1"  
        },  
        {  
            "topic": "foo2"  
        }  
    ],  
    "version": 1  
}
```

Once we've defined our JSON file, we can use it to generate a set of partition moves to move the topics listed in the file *topics.json* to the brokers with IDs 5 and 6:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092  
--topics-to-move-json-file topics.json  
--broker-list 5,6 --generate  
{"version":1,  
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},  
 {"topic":"foo1","partition":0,"replicas":[3,4]},  
 {"topic":"foo2","partition":2,"replicas":[1,2]},  
 {"topic":"foo2","partition":0,"replicas":[3,4]},  
 {"topic":"foo1","partition":1,"replicas":[2,3]},  
 {"topic":"foo2","partition":1,"replicas":[2,3]}]  
}  
  
Proposed partition reassignment configuration  
  
{"version":1,  
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},  
 {"topic":"foo1","partition":0,"replicas":[5,6]},  
 {"topic":"foo2","partition":2,"replicas":[5,6]},  
 {"topic":"foo2","partition":0,"replicas":[5,6]},  
 {"topic":"foo1","partition":1,"replicas":[5,6]},  
 {"topic":"foo2","partition":1,"replicas":[5,6]}]  
}  
#
```

The output proposed here is formatted correctly, to which we can save two new JSON files that we will call *revert-reassignment.json* and *expand-cluster-reassignment.json*. The first file can be used to move partitions back to where they were originally if you need to roll back for some reason. The second file can be used for the next step, as this is just a proposal and hasn't executed anything yet. You'll notice in the output that there isn't a good balance of leadership, as the proposal will result in all leadership moving to broker 5. We will ignore this for now and presume the cluster automatic leadership balancing is enabled, which will help distribute it later. It should be

noted that the first step can be skipped if you know exactly where you want to move your partitions to and you manually craft the JSON to move partitions.

To execute the proposed partition reassignment from the file *expand-cluster-reassignment.json*, run the following command:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--execute
  Current partition replica assignment

  {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
    {"topic":"foo1","partition":0,"replicas":[3,4]},
    {"topic":"foo2","partition":2,"replicas":[1,2]},
    {"topic":"foo2","partition":0,"replicas":[3,4]},
    {"topic":"foo1","partition":1,"replicas":[2,3]},
    {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
  {"topic":"foo1","partition":0,"replicas":[5,6]},
  {"topic":"foo2","partition":2,"replicas":[5,6]},
  {"topic":"foo2","partition":0,"replicas":[5,6]},
  {"topic":"foo1","partition":1,"replicas":[5,6]},
  {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
#
```

This will start the reassignment of the specified partition replicas to the new brokers. The output is the same as the generated proposal verification. The cluster controller performs this reassignment action by adding the new replicas to the replica list for each partition, which will temporarily increase the replication factor of these topics. The new replicas will then copy all existing messages for each partition from the current leader. Depending on the size of the partitions on disk, this can take a significant amount of time as the data is copied across the network to the new replicas. Once replication is complete, the controller removes the old replicas from the replica list by reducing the replication factor to the original size with the old replicas removed.

Here are a few other useful features of the command you could take advantage of:

--additional

This option will allow you to add to the existing reassessments so they can continue to be performed without interruption and without the need to wait until the original movements have completed in order to start a new batch.

--disable-rack-aware

There may be times when, due to rack awareness settings, the end-state of a proposal may not be possible. This can be overridden with this command if necessary.

--throttle

This value is in units of bytes/sec. Partition reassessments have a big impact on the performance of your cluster, as they will cause changes in the consistency of the memory page cache and use network and disk I/O. Throttling the movement of partitions can be useful to prevent this issue. This can be combined with the --additional tag to throttle an already-started reassignment process that may be causing issues.



Improving Network Utilization When Reassigning Replicas

When removing many partitions from a single broker, such as if that broker is being removed from the cluster, it may be useful to remove all leadership from the broker first. This can be done by manually moving leaderships off the broker; however, using the preceding tooling to do this is arduous. Other open source tools such as Cruise Control include features like broker “demotion,” which safely moves leadership off a broker and is probably the simplest way to do this.

However, if you do not have access to such tools, a simple restart of a broker will suffice. As a broker is preparing to shut down, all leadership for the partitions on that particular broker will move to other brokers in the clusters. This can significantly increase the performance of reassessments and reduce the impact on the cluster, as the replication traffic will be distributed to many brokers. However, if automatic leader reassignment is enabled after the broker is bounced, leadership may return to this broker, so it may be beneficial to temporarily disable this feature.

To check on the progress of the partition moves, the tool can be used to verify the status of the reassignment. This will show which reassessments are currently in progress, which reassessments have completed, and (if there was an error) which reassessments have failed. To do this, you must have the file with the JSON object that was used in the execute step.

Here is an example of potential results using the --verify option when running the preceding partition reassignment from the file *expand-cluster-reassignment.json*:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--verify
Status of partition reassignment:
```

```

Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo1,1] is in progress
Reassignment of partition [foo1,2] is in progress
Reassignment of partition [foo2,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
Reassignment of partition [foo2,2] completed successfully
#

```

Changing the replication factor

The `kafka-reassign-partitions.sh` tool can also be used to increase or decrease the replication factor (RF) for a partition. This may be necessary in situations where a partition was created with the wrong RF, you want increased redundancy as you expand your cluster, or you want to decrease redundancy for cost savings. One clear example is that if a cluster RF default setting is adjusted, existing topics will not automatically be increased. The tool can be used to increase RF on the existing partitions.

As an example, if we wanted to increase topic “foo1” from the previous example from an RF = 2 to RF = 3, then we could craft a JSON similar to the execution proposal we used before, except we’d add in an additional broker ID to the replica set. For example, we could construct a JSON called *increase-foo1-RF.json* in which we add broker 4 to the existing set of 5,6 that we already have:

```
{
  {"version":1,
  "partitions":[{"topic":"foo1","partition":1,"replicas":[5,6,4]},
    {"topic":"foo1","partition":2,"replicas":[5,6,4]},
    {"topic":"foo1","partition":3,"replicas":[5,6,4]},
  ]
}
```

We’d then use the commands shown earlier to execute on this proposal. When it completes, we can verify the RF has been increased by either using the `--verify` flag or using the `kafka-topics.sh` script to describe the topic:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --topic foo1 --describe
Topic:foo1 PartitionCount:3      ReplicationFactor:3      Configs:
  Topic: foo1 Partition: 0    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
  Topic: foo1 Partition: 1    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
  Topic: foo1 Partition: 2    Leader: 5      Replicas: 5,6,4 Isr: 5,6,4
#

```

Cancelling replica reassessments

Cancelling a replica reassignment in the past was a dangerous process that required unsafe manual manipulation of ZooKeeper nodes (or znodes) by deleting the `/admin/reassign_partitions` znode. Fortunately, this is no longer the case. The `kafka-reassign-partitions.sh` script (as well as the AdminClient it is a wrapper

for) now supports the `--cancel` option, which will cancel the active reassessments that are ongoing in a cluster. When stopping an in-progress partition move, the `--cancel` command is designed to restore the replica set to the one it was prior to reassignment being initiated. As such, if replicas are being removed from a dead broker or an overloaded broker, it may leave the cluster in an undesirable state. There is also no guarantee that the reverted replica set will be in the same order as it was previously.

Dumping Log Segments

On occasion you may have the need to read the specific content of a message, perhaps because you ended up with a “poison pill” message in your topic that is corrupted and your consumer cannot handle it. The `kafka-dump-log.sh` tool is provided to decode the log segments for a partition. This will allow you to view individual messages without needing to consume and decode them. The tool takes a comma-separated list of log segment files as an argument and can print out either message summary information or detailed message data.

In this example, we will dump the logs from a sample topic, “my-topic,” which is a new topic with only four messages in it. First, we will simply decode the log segment file named `000000000000000000000000.log` and retrieve basic metadata info about each message without actually printing the message contents. In our example Kafka installation, the Kafka data directory is set up in `/tmp/kafka-logs`. As such, our directory for finding the log segments will be `/tmp/kafka-logs/<topic-name>-<partition>`, in this case, `/tmp/kafka-logs/my-topic-0/`:

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 0
CreateTime: 1623034799990 size: 77 magic: 2
compresscodec: NONE crc: 1773642166 isvalid: true
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 77
CreateTime: 1623034803631 size: 82 magic: 2
compresscodec: NONE crc: 1638234280 isvalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 159
CreateTime: 1623034808233 size: 82 magic: 2
compresscodec: NONE crc: 4143814684 isvalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 241
CreateTime: 1623034811837 size: 77 magic: 2
```

```
compresscodec: NONE crc: 3096928182 isvalid: true
#
```

In the next example, we add the `--print-data-log` option, which will provide us the actual payload information and more:

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
--print-data-log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 0
CreateTime: 162303479990 size: 77 magic: 2
compresscodec: NONE crc: 1773642166 isvalid: true
| offset: 0 CreateTime: 162303479990 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 1
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 77
  CreateTime: 1623034803631 size: 82 magic: 2
  compresscodec: NONE crc: 1638234280 isvalid: true
| offset: 1 CreateTime: 1623034803631 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 2
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 159
  CreateTime: 1623034808233 size: 82 magic: 2
  compresscodec: NONE crc: 4143814684 isvalid: true
| offset: 2 CreateTime: 1623034808233 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 3
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 241
  CreateTime: 1623034811837 size: 77 magic: 2
  compresscodec: NONE crc: 3096928182 isvalid: true
| offset: 3 CreateTime: 1623034811837 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 4
#
#
```

The tool also contains a few other useful options, such as validating the index file that goes along with a log segment. The index is used for finding messages within a log segment, and if corrupted, will cause errors in consumption. Validation is performed whenever a broker starts up in an unclean state (i.e., it was not stopped normally), but it can be performed manually as well. There are two options for checking indices, depending on how much checking you want to do. The option `--index-sanity-check` will just check that the index is in a usable state, while `--verify-index-only` will check for mismatches in the index without printing out all the index entries. Another useful option, `--value-decoder-class`, allows serialized messages to be deserialized by passing in a decoder.

Replica Verification

Partition replication works similar to a regular Kafka consumer client: the follower broker starts replicating at the oldest offset and checkpoints the current offset to disk periodically. When replication stops and restarts, it picks up from the last checkpoint. It is possible for previously replicated log segments to get deleted from a broker, and the follower will not fill in the gaps in this case.

To validate that the replicas for a topic's partitions are the same across the cluster, you can use the `kafka-replica-verification.sh` tool for verification. This tool will fetch messages from all the replicas for a given set of topic partitions, check that all messages exist on all replicas, and print out the max lag for given partitions. This process will operate continuously in a loop until canceled. To do this, you must provide an explicit comma-separated list of brokers to connect to. By default, all topics are validated; however, you may also provide the tool a regular expression that matches the topics you wish to validate.



Caution: Cluster Impact Ahead

The replica verification tool will have an impact on your cluster similar to reassigning partitions, as it must read all messages from the oldest offset in order to verify the replica. In addition, it reads from all replicas for a partition in parallel, so it should be used with caution.

For example, verify the replicas for the topics starting with `my` on kafka brokers 1 and 2, which contain partition 0 of “`my-topic`”:

```
# kafka-replica-verification.sh --broker-list kafka.host1.domain.com:  
9092,kafka.host2.domain.com:9092  
--topic-white-list 'my.*'  
  
2021-06-07 03:28:21,829: verification process is started.  
2021-06-07 03:28:51,949: max lag is 0 for partition my-topic-0 at offset 4  
among 1 partitions  
2021-06-07 03:29:22,039: max lag is 0 for partition my-topic-0 at offset 4  
among 1 partitions  
...  
#
```

Other Tools

Several more tools are included in the Kafka distribution that are not covered in depth in this book that can be useful in administering your Kafka cluster for specific use cases. Further information about them can be found on the [Apache Kafka website](#):

Client ACLs

A command-line tool, `kafka-acls.sh`, is provided for interacting with access controls for Kafka clients. This includes full features for authorizer properties, set up for deny or allow principles, cluster- or topic-level restrictions, ZooKeeper TLS file configuration, and much more.

Lightweight MirrorMaker

A lightweight `kafka-mirror-maker.sh` script is available for mirroring data. A more in-depth look at replication can be found in [Chapter 10](#).

Testing tools

There are several other scripts used for testing Kafka or helping to perform upgrades of features. `kafka-broker-api-versions.sh` helps to easily identify different versions of usable API elements when upgrading from one Kafka version to another and check for compatibility issues. There are producer and consumer performance tests scripts. There are several scripts to help administer ZooKeeper as well. There is also `trogdor.sh`, which is a test framework designed to run benchmarks and other workloads to attempt to stress test the system.

Unsafe Operations

There are some administrative tasks that are technically possible to do but should not be attempted except in the most extreme situations. Often this is when you are diagnosing a problem and have run out of options, or you have found a specific bug that you need to work around temporarily. These tasks are usually undocumented, unsupported, and pose some amount of risk to your application.

Several of the more common of these tasks are documented here so that in an emergency situation, there is a potential option for recovery. Their use is not recommended under normal cluster operations and should be considered carefully before being executed.



Danger: Here Be Dragons

The operations in this section often involve working with the cluster metadata stored in ZooKeeper directly. This can be a very dangerous operation, so you must be very careful to not modify the information in ZooKeeper directly, except as noted.

Moving the Cluster Controller

Every Kafka cluster has a single broker that is designated as a controller. The controller has a special thread that is responsible for overseeing cluster operations in addition to normal broker work. Normally, controller election is done automatically

through ephemeral ZooKeeper znode monitoring. When a controller turns off or becomes unavailable, other brokers nominate themselves as soon as possible, since once the controller shuts down, the znode is removed.

On occasion, when troubleshooting a misbehaving cluster or broker, it may be useful to forcibly move the controller to a different broker without shutting down the host. One such example is when the controller has suffered an exception or other problem that has left it running but not functional. Moving the controller in these situations does not normally have a high risk, but as it is not a normal task, it should not be performed regularly.

To forcibly move a controller, deleting the ZooKeeper znode at `/admin/controller` manually will cause the current controller to resign, and the cluster will randomly select a new controller. There is currently no way to specify a specific broker to be controller in Apache Kafka.

Removing Topics to Be Deleted

When attempting to delete a topic in Kafka, a ZooKeeper node requests that the deletion is created. Once every replica completes deletion of the topic and acknowledges deletion is complete, the znode will be removed. Under normal circumstances, this is executed by the cluster very quickly. However, sometimes things can go wrong with this process. Here are some scenarios in which a deletion request may become stuck:

1. A requester has no way of knowing whether topic deletion is enabled in the cluster and can request deletion of a topic from a cluster in which deletion is disabled.
2. A very large topic is requested to be deleted, but before the request is handled, one or more of the replica sets goes offline due to hardware failures, and the deletion cannot complete as the controller cannot ack that the deletion was completed successfully.

To “unstick” topic deletion, first delete the `/admin/delete_topic/<topic>` znode. Deleting the topic ZooKeeper nodes (but not the parent `/admin/delete_topic` node) will remove the pending requests. If the deletion is re-queued by cached requests in the controller, it may be necessary to also forcibly move the controller as shown earlier immediately after removing the topic znode to ensure that no cached requests are pending in the controller.

Deleting Topics Manually

If you are running a cluster with delete topics disabled, or if you find yourself needing to delete some topics outside of the normal flow of operations, it is possible to manually delete them from the cluster. This requires a full shutdown of all brokers in the

cluster, however, and cannot be done while any of the brokers in the cluster are running.



Shut Down Brokers First

Modifying the cluster metadata in ZooKeeper when the cluster is online is a very dangerous operation and can put the cluster into an unstable state. Never attempt to delete or modify topic metadata in ZooKeeper while the cluster is online.

To delete a topic from the cluster:

1. Shut down all brokers in the cluster.
2. Remove the ZooKeeper path `/brokers/topics/<topic>` from the Kafka cluster path. Note that this node has child nodes that must be deleted first.
3. Remove the partition directories from the log directories on each broker. These will be named `<topic>-<int>`, where `<int>` is the partition ID.
4. Restart all brokers.

Summary

Running a Kafka cluster can be a daunting endeavor, with numerous configurations and maintenance tasks to keep the systems running at peak performance. In this chapter, we discussed many of the routine tasks, such as managing topic and client configurations, that you will need to handle frequently. We also covered some of the more esoteric tasks that you'll need for debugging problems, like examining log segments. Finally, we covered a few of the operations that, while not safe or routine, can be used to get you out of a sticky situation. All together, these tools will help you to manage your Kafka cluster. As you begin to scale your Kafka clusters larger, even the use of these tools may become arduous and difficult to manage. It is highly recommended to engage with the open source Kafka community and take advantage of the many other open source projects in the ecosystem to help automate many of the tasks outlined in this chapter.

Now that we are confident in the tools needed to administer and manage our cluster, it is still impossible without proper monitoring in place. [Chapter 13](#) will discuss ways to monitor broker and cluster health and operations so you can be sure Kafka is working well (and know when it isn't). We will also offer best practices for monitoring your clients, including both producers and consumers.

Monitoring Kafka

The Apache Kafka applications have numerous measurements for their operation—so many, in fact, that it can easily become confusing as to what is important to watch and what can be set aside. These range from simple metrics about the overall rate of traffic, to detailed timing metrics for every request type, to per-topic and per-partition metrics. They provide a detailed view into every operation in the broker, but they can also make you the bane of whoever is responsible for managing your monitoring system.

This chapter will detail the most critical metrics to monitor all the time and how to respond to them. We'll also describe some of the more important metrics to have on hand when debugging problems. This is not an exhaustive list of the metrics that are available, however, because the list changes frequently, and many will only be informative to a hard-core Kafka developer.

Metric Basics

Before getting into the specific metrics provided by the Kafka broker and clients, let's discuss the basics of how to monitor Java applications and some best practices around monitoring and alerting. This will provide a basis for understanding how to monitor the applications and why the specific metrics described later in this chapter have been chosen as the most important.

Where Are the Metrics?

All of the metrics exposed by Kafka can be accessed via the Java Management Extensions (JMX) interface. The easiest way to use them in an external monitoring system is to use a collection agent provided by your monitoring system and attach it to the Kafka process. This may be a separate process that runs on the system and connects

to the JMX interface, such as with the Nagios XI `check_jmx` plug-in or `jmxtrans`. You can also utilize a JMX agent that runs directly in the Kafka process to access metrics via an HTTP connection, such as Jolokia or MX4J.

An in-depth discussion of how to set up monitoring agents is outside the scope of this chapter, and there are far too many choices to do justice to all of them. If your organization does not currently have experience with monitoring Java applications, it may be worthwhile to instead consider monitoring as a service. There are many companies that offer monitoring agents, metrics collection points, storage, graphing, and alerting in a services package. They can assist you further with setting up the monitoring agents required.



Finding the JMX Port

To aid with configuring applications that connect to JMX on the Kafka broker directly, such as monitoring systems, the broker sets the configured JMX port in the broker information that is stored in ZooKeeper. The `/brokers/ids/<ID>` znode contains JSON-formatted data for the broker, including `hostname` and `jmx_port` keys. However, it should be noted that remote JMX is disabled by default in Kafka for security reasons. If you are going to enable it, you must properly configure security for the port. This is because JMX not only allows a view into the state of the application, it also allows code execution. It is highly recommended that you use a JMX metrics agent that is loaded into the application.

Nonapplication metrics

Not all metrics will come from Kafka itself. There are five general groupings of where you can get your metrics from. **Table 13-1** describes the categories when we are monitoring the Kafka brokers.

Table 13-1. Metric sources

Category	Description
Application metrics	These are the metrics you get from Kafka itself, from the JMX interface.
Logs	Another type of monitoring data that comes from Kafka itself. Because it is some form of text or structured data, and not just a number, it requires a little more processing.
Infrastructure metrics	These metrics come from systems that you have in front of Kafka but are still within the request path and under your control. An example is a load balancer.
Synthetic clients	This is data from tools that are external to your Kafka deployment, just like a client, but are under your direct control and are typically not performing the same work as your clients. An external monitor like Kafka Monitor falls in this category.
Client metrics	These are metrics that are exposed by the Kafka clients that connect to your cluster.

Logs generated by Kafka are discussed later in this chapter, as are client metrics. We will also touch very briefly on synthetic metrics. Infrastructure metrics, however, are dependent on your specific environment and are outside the scope of the discussion here. The further along in your Kafka journey you are, the more important these metric sources will be to fully understanding how your applications are running, as the lower in the list, the more objective a view of Kafka they provide. For example, relying on metrics from your brokers will suffice at the start, but later on you will want a more objective view of how they are performing. A familiar example for the value of objective measurements is monitoring the health of a website. The web server is running properly, and all of the metrics it is reporting say that it is working. However, there is a problem with the network between your web server and your external users, which means that none of your users can reach the web server. A synthetic client that is running outside your network and checks the accessibility of the website would detect this and alert you to the situation.

What Metrics Do I Need?

The specific metrics that are important to you is a question that is nearly as loaded as what the best editor to use is. It will depend significantly on what you intend to do with them, what tools you have available for collecting data, how far along in using Kafka you are, and how much time you have available to spend on building infrastructure around Kafka. A broker internals developer will have far different needs than a site reliability engineer who is running a Kafka deployment.

Alerting or debugging?

The first question you should ask yourself is whether or not your primary goal is to alert you when there is a problem with Kafka, or to debug problems that happen. The answer will usually involve a little of both, but knowing whether a metric is for one or the other will allow you to treat it differently once it is collected.

A metric that is destined for alerting is useful for a very short period of time—typically, not much longer than the amount of time it takes to respond to a problem. You can measure this on the order of hours, or maybe days. These metrics will be consumed by automation that responds to known problems for you, as well as the human operators in cases where automation does not exist yet. It is usually important for these metrics to be more objective, as a problem that does not impact clients is far less critical than one that does.

Data that is primarily for debugging has a longer time horizon because you are frequently diagnosing problems that have existed for some time, or taking a deeper look at a more complex problem. This data will need to remain available for days or weeks past when it is collected. It is also usually going to be more subjective measurements, or data from the Kafka application itself. Keep in mind that it is not always necessary

to collect this data into a monitoring system. If the metrics are used for debugging problems in place, it is sufficient that the metrics are available when needed. You do not need to overwhelm the monitoring system by collecting tens of thousands of values on an ongoing basis.



Historical Metrics

There is a third type of data that you will need eventually, and that is historical data on your application. The most common use for historical data is for capacity management purposes, and so it includes information about resources used, including compute resources, storage, and network. These metrics will need to be stored for a very long period of time, measured in years. You also may need to collect additional metadata to put the metrics into context, such as when brokers were added to or removed from the cluster.

Automation or humans?

Another question to consider is who the consumer of the metrics will be. If the metrics are consumed by automation, they should be very specific. It's OK to have a large number of metrics, each describing small details, because this is why computers exist: to process a lot of data. The more specific the data is, the easier it is to create automation that acts on it, because the data does not leave as much room for interpretation as to its meaning. On the other hand, if the metrics will be consumed by humans, presenting a large number of metrics will be overwhelming. This becomes even more important when defining alerts based on those measurements. It is far too easy to succumb to "alert fatigue," where there are so many alerts going off that it is difficult to know how severe the problem is. It is also hard to properly define thresholds for every metric and keep them up-to-date. When the alerts are overwhelming or often incorrect, we begin to not trust that the alerts are correctly describing the state of our applications.

Think about the operations of a car. To properly adjust the ratio of air to fuel while the car is running, the computer needs a number of measurements of air density, fuel, exhaust, and other minutiae about the operation of the engine. These measurements would be overwhelming to the human operator of the vehicle, however. Instead, we have a "Check Engine" light. A single indicator tells you that there is a problem, and there is a way to find out more detailed information to tell you exactly what the problem is. Throughout this chapter, we will identify the metrics that will provide the highest amount of coverage to keep your alerting simple.

Application Health Checks

No matter how you collect metrics from Kafka, you should make sure that you have a way to also monitor the overall health of the application process via a simple health check. This can be done in two ways:

- An external process that reports whether the broker is up or down (health check)
- Alerting on the lack of metrics being reported by the Kafka broker (sometimes called *stale metrics*)

Though the second method works, it can make it difficult to differentiate between a failure of the Kafka broker and a failure of the monitoring system itself.

For the Kafka broker, this can simply be connecting to the external port (the same port that clients use to connect to the broker) to check that it responds. For client applications, it can be more complex, ranging from a simple check of whether the process is running, to an internal method that determines application health.

Service-Level Objectives

One area of monitoring that is especially critical for infrastructure services, such as Kafka, is that of service-level objectives, or SLOs. This is how we communicate to our clients what level of service they can expect from the infrastructure service. The clients want to be able to treat services like Kafka as an opaque system: they do not want or need to understand the internals of how it works—only the interface that they are using and knowing it will do what they need it to do.

Service-Level Definitions

Before discussing SLOs in Kafka, there must be agreement on the terminology that is used. Frequently, you will hear engineers, managers, executives, and everyone else use terms in the “service-level” space incorrectly, which leads to confusion about what is actually being talked about.

A *service-level indicator* (SLI) is a metric that describes one aspect of a service’s reliability. It should be closely aligned with your client’s experience, so it is usually true that the more objective these measurements are, the better they are. In a request processing system, such as Kafka, it is usually best to express these measurements as a ratio between the number of good events and the total number of events—for example, the proportion of requests to a web server that return a 2xx, 3xx, or 4xx response.

A *service-level objective* (SLO), which can also be called a *service-level threshold* (SLT), combines an SLI with a target value. A common way to express the target is by the number of nines (99.9% is “three nines”), though it is by no means required. The SLO should also include a time frame that it is measured over, frequently on the scale of

days. For example, 99% of requests to the web server must return a 2xx, 3xx, or 4xx response over 7 days.

A *service-level agreement* (SLA) is a contract between a service provider and a client. It usually includes several SLOs, as well as details about how they are measured and reported, how the client seeks support from the service provider, and penalties that the service provider will be subject to if they are not performing within the SLA. For example, an SLA for the preceding SLO might state that if the service provider is not operating within the SLO, they will refund all fees paid by the client for the time period that the service was not within the SLO.



Operational-Level Agreement

The term *operational-level agreement* (OLA) is less frequently used. It describes agreements between multiple internal services or support providers in the overall delivery of an SLA. The goal is to assure that the multiple activities that are necessary to fulfill the SLA are properly described and accounted for in the day-to-day operations.

It is very common to hear people talk about SLAs when they really mean SLOs. While those who are providing a service to paying clients may have SLAs with those clients, it is rare that the engineers running the applications are responsible for anything more than the performance of that service within the SLOs. In addition, those who only have internal clients (i.e., are running Kafka as internal data infrastructure for a much larger service) generally do not have SLAs with those internal customers. This should not prevent you from setting and communicating SLOs, however, as doing that will lead to fewer assumptions by customers as to how they think Kafka should be performing.

What Metrics Make Good SLIs?

In general, the metrics for your SLIs should be gathered using something external to the Kafka brokers. The reason for this is that SLOs should describe whether or not the typical user of your service is happy, and you can't measure that subjectively. Your clients do not care if you think your service is running correctly; it is their experience (in aggregate) that matters. This means that infrastructure metrics are OK, synthetic clients are good, and client-side metrics are probably the best for most of your SLIs.

While by no means an exhaustive list, the most common SLIs that are used in request/response and data storage systems are in [Table 13-2](#).



Customers Always Want More

There are some SLOs that your customers may be interested in that are important to them but not within your control. For example, they may be concerned about the correctness or freshness of the data produced to Kafka. Do not agree to support SLOs that you are not responsible for, as that will only lead to taking on work that dilutes the core job of keeping Kafka running properly. Make sure to connect them with the proper group to set up understanding, and agreements, around these additional requirements.

Table 13-2. Types of SLIs

Availability	Is the client able to make a request and get a response?
Latency	How quickly is the response returned?
Quality	Does the response include a proper response?
Security	Are the request and response appropriately protected, whether that is authorization or encryption?
Throughput	Can the client get enough data, fast enough?

Keep in mind that it is usually better for your SLIs to be based on a counter of events that fall inside the thresholds of the SLO. This means that ideally, each event would be individually checked to see if it meets the threshold of the SLO. This rules out quantile metrics as good SLIs, as those will only tell you that 90% of your events were below a given value without allowing you to control what that value is. However, aggregating values into buckets (e.g., “less than 10 ms,” “10–50 ms,” “50–100 ms,” etc.) can be useful when working with SLOs, especially when you are not yet sure what a good threshold is. This will give you a view into the distribution of the events within the range of the SLO, and you can configure the buckets so that the boundaries are reasonable values for the SLO threshold.

Using SLOs in Alerting

In short, SLOs should inform your primary alerts. The reason for this is that the SLOs describe problems from your customers’ point of view, and those are the ones that you should be concerned about first. Generally speaking, if a problem does not impact your clients, it does not need to wake you up at night. SLOs will also tell you about the problems that you don’t know how to detect because you’ve never seen them before. They won’t tell you what those problems are, but they will tell you that they exist.

The challenge is that it's very difficult to use an SLO directly as an alert. SLOs are best for long timescales, such as a week, as we want to report them to management and customers in a way that can be consumed. In addition, by the time the SLO alert fires, it's too late—you're already operating outside of the SLO. Some will use a derivative value to provide an early warning, but the best way to approach using SLOs for alerting is to observe the rate at which you are burning through your SLO over its timeframe.

As an example, let's assume that your Kafka cluster receives one million requests per week, and you have an SLO defined that states that 99.9% of requests must send out the first byte of response within 10 ms. This means that over the week, you can have up to one thousand requests that respond slower than this and everything will still be OK. Normally, you see one request like this every hour, which is about 168 bad requests a week, measured from Sunday to Saturday. You have a metric that shows this as the SLO burn rate, and one request an hour at one million requests a week is a burn rate of 0.1% per hour.

On Tuesday at 10 a.m., your metric changes and now shows that the burn rate is 0.4% per hour. This isn't great, but it's still not a problem because you'll be well within the SLO by the end of the week. You open a ticket to take a look at the problem but go back to some higher-priority work. On Wednesday at 2 p.m., the burn rate jumps to 2% per hour and your alerts go off. You know that at this rate, you'll breach the SLO by lunchtime on Friday. Dropping everything, you diagnose the problem, and after about 4 hours you have the burn rate back down to 0.4% per hour, and it stays there for the rest of the week. By using the burn rate, you were able to avoid breaching the SLO for the week.

For more information on utilizing SLOs and the burn rate for alerting, you will find that *Site Reliability Engineering* and *The Site Reliability Workbook*, both edited by Betsy Beyer et al. (O'Reilly), are excellent resources.

Kafka Broker Metrics

There are many Kafka broker metrics. Many of them are low-level measurements, added by developers when investigating a specific issue or in anticipation of needing information for debugging purposes later. There are metrics providing information about nearly every function within the broker, but the most common ones provide the information needed to run Kafka on a daily basis.



Who Watches the Watchers?

Many organizations use Kafka for collecting application metrics, system metrics, and logs for consumption by a central monitoring system. This is an excellent way to decouple the applications from the monitoring system, but it presents a specific concern for Kafka itself. If you use this same system for monitoring Kafka itself, it is very likely that you will never know when Kafka is broken because the data flow for your monitoring system will be broken as well.

There are many ways that this can be addressed. One way is to use a separate monitoring system for Kafka that does not have a dependency on Kafka. Another way, if you have multiple datacenters, is to make sure that the metrics for the Kafka cluster in datacenter A are produced to datacenter B, and vice versa. However you decide to handle it, make sure that the monitoring and alerting for Kafka does not depend on Kafka working.

In this section, we'll start by discussing the high-level workflow for diagnosing problems with your Kafka cluster, referencing the metrics that are useful. Those, and other metrics, are described in more detail later in the chapter. This is by no means an exhaustive list of broker metrics, but rather several "must have" metrics for checking on the health of the broker and the cluster. We'll wrap up with a discussion on logging before moving on to client metrics.

Diagnosing Cluster Problems

When it comes to problems with a Kafka cluster, there are three major categories:

- Single-broker problems
- Overloaded clusters
- Controller problems

Issues with individual brokers are, by far, the easiest to diagnose and respond to. These will show up as outliers in the metrics for the cluster and are frequently related to slow or failing storage devices or compute restraints from other applications on the system. To detect them, make sure you are monitoring the availability of the individual servers, as well as the status of the storage devices, utilizing the operating system (OS) metrics.

Absent a problem identified at the OS or hardware level, however, the cause is almost always an imbalance in the load of the Kafka cluster. While Kafka attempts to keep the data within the cluster evenly spread across all brokers, this does not mean that client access to that data is evenly distributed. It also does not detect issues such as hot partitions. It is highly recommended that you utilize an external tool for keeping

the cluster balanced at all times. One such tool is [Cruise Control](#), an application that continually monitors the cluster and rebalances partitions within it. It also provides a number of other administrative functions, such as adding and removing brokers.



Preferred Replica Elections

The first step before trying to diagnose a problem further is to ensure that you have run a preferred replica election (see [Chapter 12](#)) recently. Kafka brokers do not automatically take partition leadership back (unless auto leader rebalance is enabled) after they have released leadership (e.g., when the broker has failed or been shut down). This means that it's very easy for leader replicas to become unbalanced in a cluster. The preferred replica election is safe and easy to run, so it's a good idea to do that first and see if the problem goes away.

Overloaded clusters are another problem that is easy to detect. If the cluster is balanced, and many of the brokers are showing elevated latency for requests or a low request handler pool idle ratio, you are reaching the limits of your brokers to serve traffic for this cluster. You may find upon deeper inspection that you have a client that has changed its request pattern and is now causing problems. Even when this happens, however, there may be little you can do about changing the client. The solutions available to you are either to reduce the load to the cluster or increase the number of brokers.

Problems with the controller in the Kafka cluster are much more difficult to diagnose and often fall into the category of bugs in Kafka itself. These issues manifest as broker metadata being out of sync, offline replicas when the brokers appear to be fine, and topic control actions like creation not happening properly. If you're scratching your head over a problem in the cluster and saying "That's really weird," there is a very good chance that it is because the controller did something unpredictable and bad. There are not a lot of ways to monitor the controller, but monitoring the active controller count as well as the controller queue size will give you a high-level indicator if there is a problem.

The Art of Under-Replicated Partitions

One of the most popular metrics to use when monitoring Kafka is under-replicated partitions. This measurement, provided on each broker in a cluster, gives a count of the number of partitions for which the broker is the leader replica, where the follower replicas are not caught up. This single measurement provides insight into a number of problems with the Kafka cluster, from a broker being down to resource exhaustion. With the wide variety of problems that this metric can indicate, it is worthy of an in-depth look at how to respond to a value other than zero. Many of the metrics used

in diagnosing these types of problems will be described later in this chapter. See [Table 13-3](#) for more details on under-replicated partitions.

Table 13-3. Metrics and their corresponding under-replicated partitions

Metric name	Under-replicated partitions
JMX MBean	kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions
Value range	Integer, zero or greater



The URP Alerting Trap

In the previous edition of this book, as well as in many conference talks, the authors have spoken at length about the fact that the under-replicated partitions (URP) metric should be your primary alerting metric because of how many problems it describes. This approach has a significant number of problems, not the least of which is that the URP metric can frequently be nonzero for benign reasons. This means that as someone operating a Kafka cluster, you will receive false alerts, which lead to the alert being ignored. It also requires a significant amount of knowledge to be able to understand what the metric is telling you. For this reason, we no longer recommend the use of URP for alerting. Instead, you should depend on SLO-based alerting to detect unknown problems.

A steady (unchanging) number of under-replicated partitions reported by many of the brokers in a cluster normally indicates that one of the brokers in the cluster is off-line. The count of under-replicated partitions across the entire cluster will equal the number of partitions that are assigned to that broker, and the broker that is down will not report a metric. In this case, you will need to investigate what has happened to that broker and resolve that situation. This is often a hardware failure, but it could also be an OS or Java issue that has caused the problem.

If the number of under-replicated partitions is fluctuating, or if the number is steady but there are no brokers offline, this typically indicates a performance issue in the cluster. These types of problems are much harder to diagnose due to their variety, but there are several steps you can work through to narrow it down to the most likely causes. The first step is to try and determine if the problem relates to a single broker or to the entire cluster. This can sometimes be a difficult question to answer. If the under-replicated partitions are on a single broker, as in the following example, then that broker is typically the problem. The error shows that other brokers are having a problem replicating messages from that one.

If several brokers have under-replicated partitions, it could be a cluster problem, but it might still be a single broker. In that case, it would be because a single broker is having problems replicating messages from everywhere, and you'll have to figure out which broker it is. One way to do this is to get a list of under-replicated partitions for the cluster and see if there is a specific broker that is common to all of the partitions that are under-replicated. Using the `kafka-topics.sh` tool (discussed in detail in [Chapter 12](#)), you can get a list of under-replicated partitions to look for a common thread.

For example, list under-replicated partitions in a cluster:

```
# kafka-topics.sh --bootstrap-server kafka1.example.com:9092/kafka-cluster  
--describe --under-replicated  
Topic: topicOne Partition: 5 Leader: 1 Replicas: 1,2 Isr: 1  
Topic: topicOne Partition: 6 Leader: 3 Replicas: 2,3 Isr: 3  
Topic: topicTwo Partition: 3 Leader: 4 Replicas: 2,4 Isr: 4  
Topic: topicTwo Partition: 7 Leader: 5 Replicas: 5,2 Isr: 5  
Topic: topicSix Partition: 1 Leader: 3 Replicas: 2,3 Isr: 3  
Topic: topicSix Partition: 2 Leader: 1 Replicas: 1,2 Isr: 1  
Topic: topicSix Partition: 5 Leader: 6 Replicas: 2,6 Isr: 6  
Topic: topicSix Partition: 7 Leader: 7 Replicas: 7,2 Isr: 7  
Topic: topicNine Partition: 1 Leader: 1 Replicas: 1,2 Isr: 1  
Topic: topicNine Partition: 3 Leader: 3 Replicas: 2,3 Isr: 3  
Topic: topicNine Partition: 4 Leader: 3 Replicas: 3,2 Isr: 3  
Topic: topicNine Partition: 7 Leader: 3 Replicas: 2,3 Isr: 3  
Topic: topicNine Partition: 0 Leader: 3 Replicas: 2,3 Isr: 3  
Topic: topicNine Partition: 5 Leader: 6 Replicas: 6,2 Isr: 6  
#
```

In this example, the common broker is number 2. This indicates that this broker is having a problem with message replication and will lead us to focus our investigation on that one broker. If there is no common broker, there is likely a cluster-wide problem.

Cluster-level problems

Cluster problems usually fall into one of two categories:

- Unbalanced load
- Resource exhaustion

The first problem, unbalanced partitions or leadership, is the easiest to find even though fixing it can be an involved process. In order to diagnose this problem, you will need several metrics from the brokers in the cluster:

- Partition count
- Leader partition count

- All topics messages in rate
- All topics bytes in rate
- All topics bytes out rate

Examine these metrics. In a perfectly balanced cluster, the numbers will be even across all brokers in the cluster, as in [Table 13-4](#).

Table 13-4. Utilization metrics

Broker	Partitions	Leaders	Messages in	Bytes in	Bytes out
1	100	50	13130 msg/s	3.56 MBps	9.45 MBps
2	101	49	12842 msg/s	3.66 MBps	9.25 MBps
3	100	50	13086 msg/s	3.23 MBps	9.82 MBps

This indicates that all the brokers are taking approximately the same amount of traffic. Assuming you have already run a preferred replica election, a large deviation indicates that the traffic is not balanced within the cluster. To resolve this, you will need to move partitions from the heavily loaded brokers to the less heavily loaded brokers. This is done using the `kafka-reassign-partitions.sh` tool described in [Chapter 12](#).



Helpers for Balancing Clusters

The Kafka broker itself does not provide for automatic reassignment of partitions in a cluster. This means that balancing traffic within a Kafka cluster can be a mind-numbing process of manually reviewing long lists of metrics and trying to come up with a replica assignment that works. To help with this, some organizations have developed automated tools for performing this task. One example is the `kafka-assigner` tool that LinkedIn has released in the open source [kafka-tools repository on GitHub](#). Some enterprise offerings for Kafka support also provide this feature.

Another common cluster performance issue is exceeding the capacity of the brokers to serve requests. There are many possible bottlenecks that could slow things down: CPU, disk IO, and network throughput are a few of the most common. Disk utilization is not one of them, as the brokers will operate properly right up until the disk is filled, and then this disk will fail abruptly. In order to diagnose a capacity problem, there are many metrics you can track at the OS level, including:

- CPU utilization
- Inbound network throughput
- Outbound network throughput

- Disk average wait time
- Disk percent utilization

Exhausting any of these resources will typically show up as the same problem: under-replicated partitions. It's critical to remember that the broker replication process operates in exactly the same way that other Kafka clients do. If your cluster is having problems with replication, then your customers are having problems with producing and consuming messages as well. It makes sense to develop a baseline for these metrics when your cluster is operating correctly and then set thresholds that indicate a developing problem long before you run out of capacity. You will also want to review the trend for these metrics as the traffic to your cluster increases over time. As far as Kafka broker metrics are concerned, the `All Topics Bytes In Rate` is a good guideline to show cluster usage.

Host-level problems

If the performance problem with Kafka is not present in the entire cluster and can be isolated to one or two brokers, it's time to examine that server and see what makes it different from the rest of the cluster. These types of problems fall into several general categories:

- Hardware failures
- Networking
- Conflicts with another process
- Local configuration differences



Typical Servers and Problems

A server and its OS is a complex machine with thousands of components, any of which could have problems and cause either a complete failure or just a performance degradation. It's impossible for us to cover everything that can fail in this book—numerous volumes have been written, and will continue to be, on this subject. But we can discuss some of the most common problems that are seen. This section will focus on issues with a typical server running a Linux OS.

Hardware failures are sometimes obvious, like when the server just stops working, but it's the less obvious problems that cause performance issues. These are usually soft failures that allow the system to keep running but degrade operation. This could be a bad bit of memory, where the system has detected the problem and bypassed that segment (reducing the overall available memory). The same can happen with a CPU

failure. For problems such as these, you should be using the facilities that your hardware provides, such as an intelligent platform management interface (IPMI) to monitor hardware health. When there's an active problem, looking at the kernel ring buffer using `dmesg` will help you to see log messages that are getting thrown to the system console.

The more common type of hardware failure that leads to a performance degradation in Kafka is a disk failure. Apache Kafka is dependent on the disk for persistence of messages, and producer performance is directly tied to how fast your disks commit those writes. Any deviation in this will show up as problems with the performance of the producers and the replica fetchers. The latter is what leads to under-replicated partitions. As such, it is important to monitor the health of the disks at all times and address any problems quickly.



One Bad Egg

A single disk failure on a single broker can destroy the performance of an entire cluster. This is because the producer clients will connect to all brokers that lead partitions for a topic, and if you have followed best practices, those partitions will be evenly spread over the entire cluster. If one broker starts performing poorly and slowing down produce requests, this will cause back pressure in the producers, slowing down requests to all brokers.

To begin with, make sure you are monitoring hardware status information for the disks from the IPMI, or the interface provided by your hardware. In addition, within the OS you should be running SMART (Self-Monitoring, Analysis and Reporting Technology) tools to both monitor and test the disks on a regular basis. This will alert you to a failure that is about to happen. It is also important to keep an eye on the disk controller, especially if it has RAID functionality, whether you are using hardware RAID or not. Many controllers have an onboard cache that is only used when the controller is healthy and the battery backup unit (BBU) is working. A failure of the BBU can result in the cache being disabled, degrading disk performance.

Networking is another area where partial failures will cause problems. Some of these problems are hardware issues, such as a bad network cable or connector. Some are configuration issues, which is usually a change in the speed or duplex settings for the connection, either on the server side or upstream on the networking hardware. Network configuration problems could also be OS issues, such as having the network buffers undersized or too many network connections taking up too much of the overall memory footprint. One of the key indicators of problems in this area will be the number of errors detected on the network interfaces. If the error count is increasing, there is probably an unaddressed issue.

If there are no hardware problems, another common problem to look for is another application running on the system that is consuming resources and putting pressure on the Kafka broker. This could be something that was installed in error, or it could be a process that is supposed to be running, such as a monitoring agent, but is having problems. Use the tools on your system, such as `top`, to identify if there is a process that is using more CPU or memory than expected.

If the other options have been exhausted and you have not yet found the source of the discrepancy on the host, a configuration difference has likely crept in, either with the broker or the system itself. Given the number of applications that are running on any single server and the number of configuration options for each of them, it can be a daunting task to find a discrepancy. This is why it is crucial that you utilize a configuration management system, such as [Chef](#) or [Puppet](#), in order to maintain consistent configurations across your OSes and applications (including Kafka).

Broker Metrics

In addition to under-replicated partitions, there are other metrics that are present at the overall broker level that should be monitored. While you may not be inclined to set alert thresholds for all of them, they provide valuable information about your brokers and your cluster. They should be present in any monitoring dashboard you create.

Active controller count

The *active controller count* metric indicates whether the broker is currently the controller for the cluster. The metric will either be 0 or 1, with 1 showing that the broker is currently the controller. At all times, only one broker should be the controller, and one broker must always be the controller in the cluster. If two brokers say that they are currently the controller, this means that you have a problem where a controller thread that should have exited has become stuck. This can cause problems with not being able to execute administrative tasks, such as partition moves, properly. To remedy this, you will need to restart both brokers at the very least. However, when there is an extra controller in the cluster, there will often be problems performing a safe shutdown of a broker, and you will need to force stop the broker instead. See [Table 13-5](#) for more details on active controller count.

Table 13-5. Active controller count metric details

Metric name	Active controller count
JMX MBean	<code>kafka.controller:type=KafkaController,name=ActiveControllerCount</code>
Value range	Zero or one

If no broker claims to be the controller in the cluster, the cluster will fail to respond properly in the face of state changes, including topic or partition creation, or broker failures. In this situation, you must investigate further to find out why the controller threads are not working properly. For example, a network partition from the ZooKeeper cluster could result in a problem like this. Once that underlying problem is fixed, it is wise to restart all the brokers in the cluster in order to reset state for the controller threads.

Controller queue size

The *controller queue size* metric indicates how many requests the controller is currently waiting to process for the brokers. The metric will be 0 or more, with the value fluctuating frequently as new requests from brokers come in and administrative actions, such as creating partitions, moving partitions, and processing leader changes happen. Spikes in the metric are to be expected, but if this value continuously increases, or stays steady at a high value and does not drop, it indicates that the controller may be stuck. This can cause problems with not being able to execute administrative tasks properly. To remedy this, you will need to move the controller to a different broker, which requires shutting down the broker that is currently the controller. However, when the controller is stuck, there will often be problems performing a controlled shutdown of any broker. See [Table 13-6](#) for more details on controller queue size.

Table 13-6. Controller queue size metric details

Metric name	Controller queue size
JMX MBean	kafka.controller:type=ControllerEventManager ,name=EventQueueSize
Value range	Integer, zero or more

Request handler idle ratio

Kafka uses two thread pools for handling all client requests: *network threads* and *request handler threads* (also called *I/O threads*). The network threads are responsible for reading and writing data to the clients across the network. This does not require significant processing, which means that exhaustion of the network threads is less of a concern. The request handler threads, however, are responsible for servicing the client request itself, which includes reading or writing the messages to disk. As such, as the brokers get more heavily loaded, there is a significant impact on this thread pool. See [Table 13-7](#) for more details on the request handler idle ratio.

Table 13-7. Request handler idle ratio details

Metric name	Request handler average idle percentage
JMX MBean	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent
Value range	Float, between zero and one inclusive



Intelligent Thread Usage

While it may seem like you will need hundreds of request handler threads, in reality you do not need to configure any more threads than you have CPUs in the broker. Apache Kafka is very smart about the way it uses the request handlers, making sure to offload to purgatory those requests that will take a long time to process. This is used, for example, when requests are being quoted or when more than one acknowledgment of produce requests is required.

The request handler idle ratio metric indicates the percentage of time the request handlers are not in use. The lower this number, the more loaded the broker is. Experience tells us that idle ratios lower than 20% indicate a potential problem, and lower than 10% is usually an active performance problem. Besides the cluster being undersized, there are two reasons for high thread utilization in this pool. The first is that there are not enough threads in the pool. In general, you should set the number of request handler threads equal to the number of processors in the system (including hyperthreaded processors).

The other common reason for high request handler thread utilization is that the threads are doing unnecessary work for each request. Prior to Kafka 0.10, the request handler thread was responsible for decompressing every incoming message batch, validating the messages and assigning offsets, and then recompressing the message batch with offsets before writing it to disk. To make matters worse, the compression methods were all behind a synchronous lock. As of version 0.10, there is a new message format that allows for relative offsets in a message batch. This means that newer producers will set relative offsets prior to sending the message batch, which allows the broker to skip recompression of the message batch. One of the single largest performance improvements you can make is to ensure that all producer and consumer clients support the 0.10 message format, and to change the message format version on the brokers to 0.10 as well. This will greatly reduce the utilization of the request handler threads.

All topics bytes in

The *all topics bytes in* rate, expressed in bytes per second, is useful as a measurement of how much message traffic your brokers are receiving from producing clients. This is a good metric to trend over time to help you determine when you need to expand the cluster or do other growth-related work. It is also useful for evaluating if one broker in a cluster is receiving more traffic than the others, which would indicate that it is necessary to rebalance the partitions in the cluster. See [Table 13-8](#) for more details.

Table 13-8. All topics bytes in metric details

Metric name	Bytes in per second
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
Value range	Rates as doubles, count as integer

As this is the first rate metric discussed, it is worth a short discussion of the attributes that are provided by these types of metrics. All of the rate metrics have seven attributes, and choosing which ones to use depends on what type of measurement you want. The attributes provide a discrete count of events, as well as an average of the number of events over various periods of time. Make sure to use the metrics appropriately, or you will end up with a flawed view of the broker.

The first two attributes are not measurements, but they will help you understand the metric you are looking at:

EventType

This is the unit of measurement for all the attributes. In this case, it is “bytes.”

RateUnit

For the rate attributes, this is the time period for the rate. In this case, it is “seconds.”

These two descriptive attributes tell us that the rates, regardless of the period of time they average over, are presented as a value of bytes per second. There are four rate attributes provided with different granularities:

OneMinuteRate

An average over the previous 1 minute

FiveMinuteRate

An average over the previous 5 minutes

FifteenMinuteRate

An average over the previous 15 minutes

MeanRate

An average since the broker was started

The OneMinuteRate will fluctuate quickly and provides more of a “point in time” view of the measurement. This is useful for seeing short spikes in traffic. The MeanRate will not vary much at all and provides an overall trend. Though MeanRate has its uses, it is probably not the metric you want to be alerted on. The FiveMinuteRate and FifteenMinuteRate provide a compromise between the two.

In addition to the rate attributes, there is a Count attribute as well. This is a constantly increasing value for the metric since the time the broker was started. For this metric, all topics bytes in, the Count represents the total number of bytes produced to the broker since the process was started. Utilized with a metrics system that supports countermetrics, this can give you an absolute view of the measurement instead of an averaged rate.

All topics bytes out

The *all topics bytes out* rate, similar to the bytes in rate, is another overall growth metric. In this case, the bytes out rate shows the rate at which consumers are reading messages out. The outbound bytes rate may scale differently than the inbound bytes rate, thanks to Kafka’s capacity to handle multiple consumers with ease. There are many deployments of Kafka where the outbound rate can easily be six times the inbound rate! This is why it is important to observe and trend the outbound bytes rate separately. See [Table 13-9](#) for more details.

Table 13-9. All topics bytes out metric details

Metric name	Bytes out per second
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
Value range	Rates as doubles, count as integer



Replica Fetchers Included

The outbound bytes rate *also* includes the replica traffic. This means that if all of the topics are configured with a replication factor of 2, you will see a bytes out rate equal to the bytes in rate when there are no consumer clients. If you have one consumer client reading all the messages in the cluster, then the bytes out rate will be twice the bytes in rate. This can be confusing when looking at the metrics if you’re not aware of what is counted.

All topics messages in

While the byte rates described previously show the broker traffic in absolute terms of bytes, the *messages in* rate shows the number of individual messages, regardless of their size, produced per second. This is useful as a growth metric as a different measure of producer traffic. It can also be used in conjunction with the bytes in rate to determine an average message size. You may also see an imbalance in the brokers, just like with the bytes in rate, that will alert you to necessary maintenance work. See [Table 13-10](#) for more details.

Table 13-10. All topics messages in metric details

Metric name	Messages in per second
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
Value range	Rates as doubles, count as integer



Why No Messages Out?

People often ask why there is no messages out metric for the Kafka broker. The reason is that when messages are consumed, the broker just sends the next batch to the consumer without expanding it to find out how many messages are inside. Therefore, the broker doesn't really know how many messages were sent out. The only metric that can be provided is the number of fetches per second, which is a request rate, not a messages count.

Partition count

The *partition count* for a broker generally doesn't change that much, as it is the total number of partitions assigned to that broker. This includes every replica the broker has, regardless of whether it is a leader or follower for that partition. Monitoring this is often more interesting in a cluster that has automatic topic creation enabled, as that can leave the creation of topics outside of the control of the person running the cluster. See [Table 13-11](#) for more details.

Table 13-11. Partition count metric details

Metric name	Partition count
JMX MBean	kafka.server:type=ReplicaManager,name=PartitionCount
Value range	Integer, zero or greater

Leader count

The *leader count* metric shows the number of partitions that the broker is currently the leader for. As with most other measurements in the brokers, this one should be generally even across the brokers in the cluster. It is much more important to check

the leader count on a regular basis, possibly alerting on it, as it will indicate when the cluster is imbalanced even if the number of replicas are perfectly balanced in count and size across the cluster. This is because a broker can drop leadership for a partition for many reasons, such as a ZooKeeper session expiration, and it will not automatically take leadership back once it recovers (except if you have enabled automatic leader rebalancing). In these cases, this metric will show fewer leaders, or often zero, which indicates that you need to run a preferred replica election to rebalance leadership in the cluster. See [Table 13-12](#) for more details.

Table 13-12. Leader count metric details

Metric name	Leader count
JMX MBean	kafka.server:type=ReplicaManager, name=LeaderCount
Value range	Integer, zero or greater

A useful way to consume this metric is to use it along with the partition count to show a percentage of partitions that the broker is the leader for. In a well-balanced cluster that is using a replication factor of 2, all brokers should be leaders for approximately 50% of their partitions. If the replication factor in use is 3, this percentage drops to 33%.

Offline partitions

Along with the under-replicated partitions count, the *offline partitions* count is a critical metric for monitoring (see [Table 13-13](#)). This measurement is only provided by the broker that is the controller for the cluster (all other brokers will report 0) and shows the number of partitions in the cluster that currently have no leader. Partitions without leaders can happen for two main reasons:

- All brokers hosting replicas for this partition are down
- No in-sync replica can take leadership due to message-count mismatches (with unclean leader election disabled)

Table 13-13. Offline partitions count metric details

Metric name	Offline partitions count
JMX MBean	kafka.controller:type=KafkaController, name=OfflinePartitionsCount
Value range	Integer, zero or greater

In a production Kafka cluster, an offline partition may be impacting the producer clients, losing messages or causing back pressure in the application. This is most often a “site down” type of problem and will need to be addressed immediately.

Request metrics

The Kafka protocol, described in [Chapter 6](#), has many different requests. Metrics are provided for how each of those requests performs. As of version 2.5.0, the following requests have metrics provided:

Table 13-14. Request metrics names

AddOffsetsToTxn	AddPartitionsToTxn	AlterConfigs
AlterPartitionReassignments	AlterReplicaLogDirs	ApiVersions
ControlledShutdown	CreateAcls	CreateDelegationToken
CreatePartitions	CreateTopics	DeleteAcls
DeleteGroups	DeleteRecords	DeleteTopics
DescribeAcls	DescribeConfigs	DescribeDelegationToken
DescribeGroups	DescribeLogDirs	ElectLeaders
EndTxn	ExpireDelegationToken	Fetch
FetchConsumer	FetchFollower	FindCoordinator
Heartbeat	IncrementalAlterConfigs	InitProducerId
JoinGroup	LeaderAndIsr	LeaveGroup
ListGroups	ListOffsets	ListPartitionReassignments
Metadata	OffsetCommit	OffsetDelete
OffsetFetch	OffsetsForLeaderEpoch	Produce
RenewDelegationToken	SaslAuthenticate	SaslHandshake
StopReplica	SyncGroup	TxnOffsetCommit
UpdateMetadata	WriteTxnMarkers	

For each of these requests, there are eight metrics provided, providing insight into each phase of the request processing. For example, for the Fetch request, the metrics shown in [Table 13-15](#) are available.

Table 13-15. Fetch request metrics

Name	JMX MBean
Total time	kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Fetch
Request queue time	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=Fetch
Local time	kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Fetch

Name	JMX MBean
Remote time	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Fetch
Throttle time	kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=Fetch
Response queue time	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Fetch
Response send time	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=Fetch
Requests per second	kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Fetch

The requests per second metric is a rate metric, as discussed earlier, and shows the total number of that type of request that has been received and processed over the time unit. This provides a view into the frequency of each request type, though it should be noted that many of the requests, such as `StopReplica` and `UpdateMetaData`, are infrequent.

The seven *time* metrics each provide a set of percentiles for requests, as well as a discrete `Count` attribute, similar to rate metrics. The metrics are all calculated since the broker was started, so keep that in mind when looking at metrics that do not change for long periods of time; the longer your broker has been running, the more stable the numbers will be. The parts of request processing they represent are:

Total time

The total amount of time the broker spends processing the request, from receiving it to sending the response back to the requester

Request queue time

The amount of time the request spends in queue after it has been received but before processing starts

Local time

The amount of time the partition leader spends processing a request, including sending it to disk (but not necessarily flushing it)

Remote time

The amount of time spent waiting for the followers before request processing can complete

Throttle time

The amount of time the response must be held in order to slow the requestor down to satisfy client quota settings

Response queue time

The amount of time the response to the request spends in the queue before it can be sent to the requestor

Response send time

The amount of time spent actually sending the response

The attributes provided for each metric are:

Count

Absolute count of number of requests since process start

Min

Minimum value for all requests

Max

Maximum value for all requests

Mean

Average value for all requests

StdDev

The standard deviation of the request timing measurements as a whole

Percentiles

50thPercentile, 75thPercentile, 95thPercentile, 98thPercentile,
99thPercentile, 999thPercentile



What Is a Percentile?

Percentiles are a common way of looking at timing measurement. A 99th percentile measurement tells us that 99% of all values in the sample group (request timings, in this case) are less than the value of the metric. This means that 1% of the values are greater than the value specified. A common pattern is to view the average value and the 99% or 99.9% value. In this way, you can understand how the average request performs and what the outliers are.

Out of all of these metrics and attributes for requests, which are the important ones to monitor? At a minimum, you should collect at least the average and one of the higher percentiles (either 99% or 99.9%) for the total time metric, as well as the requests per second metric, for every request type. This gives a view into the overall performance of requests to the Kafka broker. If you can, you should also collect those measurements for the other six timing metrics for each request type, as this will allow you to narrow down any performance problems to a specific phase of request processing.

For setting alert thresholds, the timing metrics can be difficult. The timing for a Fetch request, for example, can vary wildly depending on many factors, including settings on the client for how long it will wait for messages, how busy the particular topic being fetched is, and the speed of the network connection between the client

and the broker. It can be very useful, however, to develop a baseline value for the 99.9th percentile measurement for at least the total time, especially for Produce requests, and alert on this. Much like the under-replicated partitions metric, a sharp increase in the 99.9th percentile for Produce requests can alert you to a wide range of performance problems.

Topic and Partition Metrics

In addition to the many metrics available on the broker that describe the operation of the Kafka broker in general, there are topic- and partition-specific metrics. In larger clusters these can be numerous, and it may not be possible to collect all of them into a metrics system as a matter of normal operations. However, they are quite useful for debugging specific issues with a client. For example, the topic metrics can be used to identify a specific topic that is causing a large increase in traffic to the cluster. It also may be important to provide these metrics so that users of Kafka (the producer and consumer clients) are able to access them. Regardless of whether you are able to collect these metrics regularly, you should be aware of what is useful.

For all the examples in [Table 13-16](#), we will be using the example topic name *TOPIC NAME*, as well as partition 0. When accessing the metrics described, make sure to substitute the topic name and partition number that are appropriate for your cluster.

Per-topic metrics

For all the per-topic metrics, the measurements are very similar to the broker metrics described previously. In fact, the only difference is the provided topic name, and that the metrics will be specific to the named topic. Given the sheer number of metrics available, depending on the number of topics present in your cluster, these will almost certainly be metrics that you will not want to set up monitoring and alerts for. They are useful to provide to clients, however, so that they can evaluate and debug their own usage of Kafka.

Table 13-16. Metrics for each topic

Name	JMX MBean
Bytes in rate	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic= <i>TOPIC NAME</i>
Bytes out rate	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic= <i>TOPIC NAME</i>
Failed fetch rate	kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic= <i>TOPICNAME</i>
Failed produce rate	kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic= <i>TOPICNAME</i>

Name	JMX MBean
Messages in rate	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic= <i>TOPICNAME</i>
Fetch request rate	kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic= <i>TOPICNAME</i>
Produce request rate	kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic= <i>TOPICNAME</i>

Per-partition metrics

The per-partition metrics tend to be less useful on an ongoing basis than the per-topic metrics. Additionally, they are quite numerous as hundreds of topics can easily be thousands of partitions. Nevertheless, they can be useful in some limited situations. In particular, the partition-size metric indicates the amount of data (in bytes) that is currently being retained on disk for the partition (Table 13-17). Combined, these will indicate the amount of data retained for a single topic, which can be useful in allocating costs for Kafka to individual clients. A discrepancy between the size of two partitions for the same topic can indicate a problem where the messages are not evenly distributed across the key that is being used when producing. The log-segment count metric shows the number of log-segment files on disk for the partition. This may be useful along with the partition size for resource tracking.

Table 13-17. Metrics for each partition

Name	JMX MBean
Partition size	kafka.log:type=Log,name=Size,topic= <i>TOPICNAME</i> ,partition=0
Log segment count	kafka.log:type=Log,name=NumLogSegments,topic= <i>TOPICNAME</i> ,partition=0
Log end offset	kafka.log:type=Log,name=LogEndOffset,topic= <i>TOPICNAME</i> ,partition=0
Log start offset	kafka.log:type=Log,name=LogStartOffset,topic= <i>TOPICNAME</i> ,partition=0

The log end offset and log start offset metrics are the highest and lowest offsets for messages in that partition, respectively. It should be noted, however, that the difference between these two numbers does not necessarily indicate the number of messages in the partition, as log compaction can result in “missing” offsets that have been removed from the partition due to newer messages with the same key. In some environments, it could be useful to track these offsets for a partition. One such use case is to provide a more granular mapping of timestamp to offset, allowing for consumer clients to easily roll back offsets to a specific time (though this is less important with time-based index searching, introduced in Kafka 0.10.1).



Under-Replicated Partition Metrics

There is a per-partition metric provided to indicate whether or not the partition is under-replicated. In general, this is not very useful in day-to-day operations, as there are too many metrics to gather and watch. It is much easier to monitor the broker-wide under-replicated partition count and then use the command-line tools (described in [Chapter 12](#)) to determine the specific partitions that are under-replicated.

JVM Monitoring

In addition to the metrics provided by the Kafka broker, you should be monitoring a standard suite of measurements for all of your servers, as well as the Java Virtual Machine (JVM) itself. These will be useful to alert you to a situation, such as increasing garbage collection activity, that will degrade the performance of the broker. They will also provide insight into why you see changes in metrics downstream in the broker.

Garbage collection

For the JVM, the critical thing to monitor is the status of garbage collection (GC). The particular beans that you must monitor for this information will vary depending on the particular Java Runtime Environment (JRE) that you are using, as well as the specific GC settings in use. For an Oracle Java 1.8 JRE running with G1 garbage collection, the beans to use are shown in [Table 13-18](#).

Table 13-18. G1 garbage collection metrics

Name	JMX MBean
Full GC cycles	<code>java.lang:type=GarbageCollector,name=G1 Old Generation</code>
Young GC cycles	<code>java.lang:type=GarbageCollector,name=G1 Young Generation</code>

Note that in the semantics of GC, “Old” and “Full” are the same thing. For each of these metrics, the two attributes to watch are `CollectionCount` and `CollectionTime`. The `CollectionCount` is the number of GC cycles of that type (Full or Young) since the JVM was started. The `CollectionTime` is the amount of time, in milliseconds, spent in that type of GC cycle since the JVM was started. As these measurements are counters, they can be used by a metrics system to tell you an absolute number of GC cycles and time spent in GC per unit of time. They can also be used to provide an average amount of time per GC cycle, though this is less useful in normal operations.

Each of these metrics also has a `LastGcInfo` attribute. This is a composite value, made up of five fields, that gives you information on the last GC cycle for the type of GC described by the bean. The important value to look at is the `duration` value, as

this tells you how long, in milliseconds, the last GC cycle took. The other values in the composite (`GcThreadCount`, `id`, `startTime`, and `endTime`) are informational and not very useful. It's important to note that you will not be able to see the timing of every GC cycle using this attribute, as young GC cycles in particular can happen frequently.

Java OS monitoring

The JVM can provide you with some information on the OS through the `java.lang:type=OperatingSystem` bean. However, this information is limited and does not represent everything you need to know about the system running your broker. The two attributes that can be collected here that are of use, which are difficult to collect in the OS, are the `MaxFileDescriptorCount` and `OpenFileDescriptorCount` attributes. `MaxFileDescriptorCount` will tell you the maximum number of file descriptors (FDs) that the JVM is allowed to have open. The `OpenFileDescriptorCount` attribute tells you the number of FDs that are currently open. There will be FDs open for every log segment and network connection, and they can add up quickly. A problem closing network connections properly could cause the broker to rapidly exhaust the number allowed.

OS Monitoring

The JVM cannot provide us with all the information that we need to know about the system it is running on. For this reason, we must not only collect metrics from the broker but also from the OS itself. Most monitoring systems will provide agents that will collect more OS information than you could possibly be interested in. The main areas that are necessary to watch are CPU usage, memory usage, disk usage, disk I/O, and network usage.

For CPU utilization, you will want to look at the system load average at the very least. This provides a single number that will indicate the relative utilization of the processors. In addition, it may also be useful to capture the percent usage of the CPU, broken down by type. Depending on the method of collection and your particular OS, you may have some or all of the following CPU percentage breakdowns (provided with the abbreviation used):

`us`

The time spent in user space

`sy`

The time spent in kernel space

`ni`

The time spent on low-priority processes

id

The time spent idle

wa

The time spent in wait (on disk)

hi

The time spent handling hardware interrupts

si

The time spent handling software interrupts

st

The time waiting for the hypervisor



What Is System Load?

While many know that system load is a measure of CPU usage on a system, most people misunderstand how it is measured. The load average is a count of the number of processes that are runnable and are waiting for a processor to execute on. Linux also includes threads that are in an uninterruptible sleep state, such as waiting for the disk. The load is presented as three numbers, which is the count averaged over the last minute, 5 minutes, and 15 minutes. In a single CPU system, a value of 1 would mean the system is 100% loaded, with a thread always waiting to execute. This means that on a multiple CPU system, the load average number that indicates 100% is equal to the number of CPUs in the system. For example, if there are 24 processors in the system, 100% would be a load average of 24.

The Kafka broker uses a significant amount of processing for handling requests. For this reason, keeping track of the CPU utilization is important when monitoring Kafka. Memory is less important to track for the broker itself, as Kafka will normally be run with a relatively small JVM heap size. It will use a small amount of memory outside of the heap for compression functions, but most of the system memory will be left to be used for cache. All the same, you should keep track of memory utilization to make sure other applications do not infringe on the broker. You will also want to make sure that swap memory is not being used by monitoring the amount of total and free swap memory.

Disk is by far the most important subsystem when it comes to Kafka. All messages are persisted to disk, so the performance of Kafka depends heavily on the performance of the disks. Monitoring usage of both disk space and inodes (*inodes* are the file and directory metadata objects for Unix filesystems) is important, as you need to assure that you are not running out of space. This is especially true for the partitions where

Kafka data is being stored. It is also necessary to monitor the disk I/O statistics, as this will tell us that the disk is being used efficiently. For at least the disks where Kafka data is stored, monitor the reads and writes per second, the average read and write queue sizes, the average wait time, and the utilization percentage of the disk.

Finally, monitor the network utilization on the brokers. This is simply the amount of inbound and outbound network traffic, normally reported in bits per second. Keep in mind that every bit inbound to the Kafka broker will be a number of bits outbound equal to the replication factor of the topics, not including consumers. Depending on the number of consumers, outbound network traffic could easily be an order of magnitude larger than inbound traffic. Keep this in mind when setting thresholds for alerts.

Logging

No discussion of monitoring is complete without a word about logging. Like many applications, the Kafka broker will fill disks with log messages in minutes if you let it. In order to get useful information from logging, it is important to enable the right loggers at the right levels. By simply logging all messages at the `INFO` level, you will capture a significant amount of important information about the state of the broker. It is useful to separate a couple of loggers from this, however, in order to provide a cleaner set of log files.

There are two loggers writing to separate files on disk. The first is `kafka.controller`, still at the `INFO` level. This logger is used to provide messages specifically regarding the cluster controller. At any time, only one broker will be the controller, and therefore only one broker will be writing to this logger. The information includes topic creation and modification, broker status changes, and cluster activities such as preferred replica elections and partition moves. The other logger to separate is `kafka.server.ClientQuotaManager`, also at the `INFO` level. This logger is used to show messages related to produce and consume quota activities. While this is useful information, it is better to not have it in the main broker log file.

It is also helpful to log information regarding the status of the log compaction threads. There is no single metric to show the health of these threads, and it is possible for failure in compaction of a single partition to halt the log compaction threads entirely, and silently. Enabling the `kafka.log.LogCleaner`, `kafka.log.Cleaner`, and `kafka.log.LogCleanerManager` loggers at the `DEBUG` level will output information about the status of these threads. This will include information about each partition being compacted, including the size and number of messages in each. Under normal operations, this is not a lot of logging, which means that it can be enabled by default without overwhelming you.

There is also some logging that may be useful to turn on when debugging issues with Kafka. One such logger is `kafka.request.logger`, turned on at either the DEBUG or TRACE levels. This logs information about every request sent to the broker. At the DEBUG level, the log includes connection end points, request timings, and summary information. At the TRACE level, it will also include topic and partition information—nearly all request information short of the message payload itself. At either level, this logger generates a significant amount of data, and it is not recommended to enable it unless necessary for debugging.

Client Monitoring

All applications need monitoring. Those that instantiate a Kafka client, either a producer or consumer, have metrics specific to the client that should be captured. This section covers the official Java client libraries, though other implementations should have their own measurements available.

Producer Metrics

The Kafka producer client has greatly compacted the metrics available by making them available as attributes on a small number of JMX MBeans. In contrast, the previous version of the producer client (which is no longer supported) used a larger number of MBeans but had more detail in many of the metrics (providing a greater number of percentile measurements and different moving averages). As a result, the overall number of metrics provided covers a wider surface area, but it can be more difficult to track outliers.

All of the producer metrics have the client ID of the producer client in the bean names. In the examples provided, this has been replaced with *CLIENTID*. Where a bean name contains a broker ID, this has been replaced with *BROKERID*. Topic names have been replaced with *TOPICNAME*. See [Table 13-19](#) for an example.

Table 13-19. Kafka producer metric MBeans

Name	JMX MBean
Overall producer	<code>kafka.producer:type=producer-metrics,client-id=CLIENTID</code>
Per-broker	<code>kafka.producer:type=producer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID</code>
Per-topic	<code>kafka.producer:type=producer-topic-metrics,client-id=CLIENTID,topic=TOPICNAME</code>

Each of the metric beans in [Table 13-19](#) has multiple attributes available to describe the state of the producer. The particular attributes that are of the most use are

described in the next section. Before proceeding, be sure you understand the semantics of how the producer works, as described in [Chapter 3](#).

Overall producer metrics

The overall producer metrics bean provides attributes describing everything from the sizes of the message batches to the memory buffer utilization. While all of these measurements have their place in debugging, there are only a handful needed on a regular basis, and only a couple of those that should be monitored and have alerts. Note that while we will discuss several metrics that are averages (ending in `-avg`), there are also maximum values for each metric (ending in `-max`) that have limited usefulness.

The `record-error-rate` is one attribute that you will definitely want to set an alert for. This metric should always be zero, and if it is anything greater than that, the producer is dropping messages it is trying to send to the Kafka brokers. The producer has a configured number of retries and a backoff between those, and once that has been exhausted, the messages (called `records` here) will be dropped. There is also a `record-retry-rate` attribute that can be tracked, but it is less critical than the error rate because retries are normal.

The other metric to alert on is the `request-latency-avg`. This is the average amount of time a produce request sent to the brokers takes. You should be able to establish a baseline value for what this number should be in normal operations, and set an alert threshold above that. An increase in the request latency means that produce requests are getting slower. This could be due to networking issues, or it could indicate problems on the brokers. Either way, it's a performance issue that will cause back pressure and other problems in your producing application.

In addition to these critical metrics, it is always good to know how much message traffic your producer is sending. Three attributes will provide three different views of this. The `outgoing-byte-rate` describes the messages in absolute size in bytes per second. The `record-send-rate` describes the traffic in terms of the number of messages produced per second. Finally, the `request-rate` provides the number of produce requests sent to the brokers per second. A single request contains one or more batches. A single batch contains one or more messages. And, of course, each message is made up of some number of bytes. These metrics are all useful to have on an application dashboard.

There are also metrics that describe the size of records, requests, and batches. The `request-size-avg` metric provides the average size of the produce requests being sent to the brokers in bytes. The `batch-size-avg` provides the average size of a single message batch (which, by definition, is comprised of messages for a single topic partition) in bytes. The `record-size-avg` shows the average size of a single record in

bytes. For a single-topic producer, this provides useful information about the messages being produced. For multiple-topic producers, such as MirrorMaker, it is less informative. Besides these three metrics, there is a `records-per-request-avg` metric that describes the average number of messages that are in a single produce request.

The last overall producer metric attribute that is recommended is `record-queue-time-avg`. This measurement is the average amount of time, in milliseconds, that a single message waits in the producer, after the application sends it, before it is actually produced to Kafka. After an application calls the producer client to send a message (by calling the `send` method), the producer waits until one of two things happens:

- It has enough messages to fill a batch based on the `batch.size` configuration.
- It has been long enough since the last batch was sent based on the `linger.ms` configuration.

Either of these two will cause the producer client to close the current batch it is building and send it to the brokers. The easiest way to understand it is that for busy topics, the first condition will apply, whereas for slow topics, the second will apply. The `record-queue-time-avg` measurement will indicate how long messages take to be produced, and therefore is helpful when tuning these two configurations to meet the latency requirements for your application.

Per-broker and per-topic metrics

In addition to the overall producer metrics, there are metric beans that provide a limited set of attributes for the connection to each Kafka broker, as well as for each topic that is being produced. These measurements are useful for debugging problems in some cases, but they are not metrics that you are going to want to review on an ongoing basis. All of the attributes on these beans are the same as the attributes for the overall producer beans described previously and have the same meaning as described previously (except that they apply either to a specific broker or a specific topic).

The most useful metric provided by the per-broker producer metrics is the `request-latency-avg` measurement. This is because this metric will be mostly stable (given stable batching of messages) and can still show a problem with connections to a specific broker. The other attributes, such as `outgoing-byte-rate` and `request-latency-avg`, tend to vary depending on what partitions each broker is leading. This means that what these measurements “should” be at any point in time can quickly change, depending on the state of the Kafka cluster.

The topic metrics are a little more interesting than the per-broker metrics, but they will only be useful for producers that are working with more than one topic. They will also only be usable on a regular basis if the producer is not working with a lot of

topics. For example, a MirrorMaker could be producing hundreds, or thousands, of topics. It is difficult to review all of those metrics, and nearly impossible to set reasonable alert thresholds on them. As with the per-broker metrics, the per-topic measurements are best used when investigating a specific problem. The `record-send-rate` and `record-error-rate` attributes, for example, can be used to isolate dropped messages to a specific topic (or validated to be across all topics). In addition, there is a `byte-rate` metric that provides the overall messages rate in bytes per second for the topic.

Consumer Metrics

Similar to the producer client, the consumer in Kafka consolidates many of the metrics into attributes on just a few metric beans. These metrics have also eliminated the percentiles for latencies and the moving averages for rates, which were present in the deprecated Scala consumer, similar to the producer client. In the consumer, because the logic around consuming messages is a little more complex than just firing messages into the Kafka brokers, there are a few more metrics to deal with as well. See [Table 13-20](#).

Table 13-20. Kafka consumer metric MBeans

Name	JMX MBean
Overall consumer	<code>kafka.consumer:type=consumer-metrics,client-id=CLIENTID</code>
Fetch manager	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID</code>
Per-topic	<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENT ID,topic=TOPICNAME</code>
Per-broker	<code>kafka.consumer:type=consumer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID</code>
Coordinator	<code>kafka.consumer:type=consumer-coordinator-metrics,client-id=CLIENTID</code>

Fetch manager metrics

In the consumer client, the overall consumer metric bean is less useful for us because the metrics of interest are located in the *fetch manager* beans instead. The overall consumer bean has metrics regarding the lower-level network operations, but the fetch manager bean has metrics regarding bytes, request, and record rates. Unlike the producer client, the metrics provided by the consumer are useful to look at but not useful for setting up alerts on.

For the fetch manager, the one attribute you may want to set up monitoring and alerts for is `fetch-latency-avg`. As with the equivalent `request-latency-avg` in the producer client, this metric tells us how long fetch requests to the brokers take. The problem with alerting on this metric is that the latency is governed by the consumer configurations `fetch.min.bytes` and `fetch.max.wait.ms`. A slow topic will have

erratic latencies, as sometimes the broker will respond quickly (when there are messages available), and sometimes it will not respond for `fetch.max.wait.ms` (when there are no messages available). When consuming topics that have more regular, and abundant, message traffic, this metric may be more useful to look at.



Wait! No Lag?

The best advice for all consumers is that you must monitor the consumer lag. So why do we not recommend monitoring the `records-lag-max` attribute on the fetch manager bean? This metric shows the current lag (the difference between the consumer's offset and the broker's log-end offset) for the partition that is the most behind.

The problem with this is twofold: it only shows the lag for one partition, and it relies on proper functioning of the consumer. If you have no other option, use this attribute for lag and set up alerting for it. But the best practice is to use external lag monitoring, as will be described in “[Lag Monitoring](#)” on page 377.

To know how much message traffic your consumer client is handling, you should capture the `bytes-consumed-rate` or the `records-consumed-rate`, or preferably both. These metrics describe the message traffic consumed by this client instance in bytes per second and messages per second, respectively. Some users set minimum thresholds on these metrics for alerting so that they are notified if the consumer is not doing enough work. You should be careful when doing this, however. Kafka is intended to decouple the consumer and producer clients, allowing them to operate independently. The rate at which the consumer is able to consume messages is often dependent on whether or not the producer is working correctly, so monitoring these metrics on the consumer makes assumptions about the state of the producer. This can lead to false alerts on the consumer clients.

It is also good to understand the relationship among bytes, messages, and requests, and the fetch manager provides metrics to help with this. The `fetch-rate` measurement tells us the number of fetch requests per second that the consumer is performing. The `fetch-size-avg` metric gives the average size of those fetch requests in bytes. Finally, the `records-per-request-avg` metric gives us the average number of messages in each fetch request. Note that the consumer does not provide an equivalent to the producer `record-size-avg` metric to let us know what the average size of a message is. If this is important, you will need to infer it from the other metrics available or capture it in your application after receiving messages from the consumer client library.

Per-broker and per-topic metrics

The metrics that are provided by the consumer client for each of the broker connections and each of the topics being consumed, as with the producer client, are useful for debugging issues with consumption, but will probably not be measurements that you review daily. As with the fetch manager, the `request-latency-avg` attribute provided by the per-broker metrics bean has limited usefulness, depending on the message traffic in the topics you are consuming. The `incoming-byte-rate` and `request-rate` metrics break down the consumed message metrics provided by the fetch manager into per-broker bytes per second and requests per second measurements, respectively. These can be used to help isolate problems that the consumer is having with the connection to a specific broker.

Per-topic metrics provided by the consumer client are useful if more than one topic is being consumed. Otherwise, these metrics will be the same as the fetch manager's metrics and redundant to collect. On the other end of the spectrum, if the client is consuming many topics (Kafka MirrorMaker, for example) these metrics will be difficult to review. If you plan on collecting them, the most important metrics to gather are the `bytes-consumed-rate`, the `records-consumed-rate`, and the `fetch-size-avg`. The `bytes-consumed-rate` shows the absolute size in bytes consumed per second for the specific topic, while the `records-consumed-rate` shows the same information in terms of the number of messages. The `fetch-size-avg` provides the average size of each fetch request for the topic in bytes.

Consumer coordinator metrics

As described in [Chapter 4](#), consumer clients generally work together as part of a consumer group. This group has coordination activities, such as group members joining, and heartbeat messages to the brokers to maintain group membership. The consumer coordinator is the part of the consumer client that is responsible for handling this work, and it maintains its own set of metrics. As with all metrics, there are many numbers provided but only a few key ones that you should monitor regularly.

The biggest problem that consumers can run into due to coordinator activities is a pause in consumption while the consumer group synchronizes. This is when the consumer instances in a group negotiate which partitions will be consumed by which individual client instances. Depending on the number of partitions that are being consumed, this can take some time. The coordinator provides the metric attribute `sync-time-avg`, which is the average amount of time, in milliseconds, that the sync activity takes. It is also useful to capture the `sync-rate` attribute, which is the number of group syncs that happen every second. For a stable consumer group, this number should be zero most of the time.

The consumer needs to commit offsets to checkpoint its progress in consuming messages, either automatically on a regular interval or by manual checkpoints triggered

in the application code. These commits are essentially just produce requests (though they have their own request type), in that the offset commit is a message produced to a special topic. The consumer coordinator provides the `commit-latency-avg` attribute, which measures the average amount of time that offset commits take. You should monitor this value just as you would the request latency in the producer. It should be possible to establish a baseline expected value for this metric, and set reasonable thresholds for alerting above that value.

One final coordinator metric that can be useful to collect is `assigned-partitions`. This is a count of the number of partitions that the consumer client (as a single instance in the consumer group) has been assigned to consume. This is helpful because, when compared to this metric from other consumer clients in the group, it is possible to see the balance of load across the entire consumer group. We can use this to identify imbalances that might be caused by problems in the algorithm used by the consumer coordinator for distributing partitions to group members.

Quotas

Apache Kafka has the ability to throttle client requests in order to prevent one client from overwhelming the entire cluster. This is configurable for both producer and consumer clients and is expressed in terms of the permitted amount of traffic from an individual client ID to an individual broker in bytes per second. There is a broker configuration, which sets a default value for all clients, as well as per-client overrides that can be dynamically set. When the broker calculates that a client has exceeded its quota, it slows the client down by holding the response back to the client for enough time to keep the client under the quota.

The Kafka broker does not use error codes in the response to indicate that the client is being throttled. This means that it is not obvious to the application that throttling is happening without monitoring the metrics that are provided to show the amount of time that the client is being throttled. The metrics that must be monitored are shown in [Table 13-21](#).

Table 13-21. Metrics to monitor

Client	Bean name
Consumer	<code>bean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID,attribute fetch-throttle-time-avg</code>
Producer	<code>bean kafka.producer:type=producer-metrics,client-id=CLIENTID,attribute produce-throttle-time-avg</code>

Quotas are not enabled by default on the Kafka brokers, but it is safe to monitor these metrics irrespective of whether or not you are currently using quotas. Monitoring

them is a good practice as they may be enabled at some point in the future, and it's easier to start with monitoring them as opposed to adding metrics later.

Lag Monitoring

For Kafka consumers, the most important thing to monitor is the consumer lag. Measured in number of messages, this is the difference between the last message produced in a specific partition and the last message processed by the consumer. While this topic would normally be covered in the previous section on consumer client monitoring, it is one of the cases where external monitoring far surpasses what is available from the client itself. As mentioned previously, there is a lag metric in the consumer client, but using it is problematic. It only represents a single partition, the one that has the most lag, so it does not accurately show how far behind the consumer is. In addition, it requires proper operation of the consumer, because the metric is calculated by the consumer on each fetch request. If the consumer is broken or offline, the metric is either inaccurate or not available.

The preferred method of consumer lag monitoring is to have an external process that can watch both the state of the partition on the broker, tracking the offset of the most recently produced message, and the state of the consumer, tracking the last offset the consumer group has committed for the partition. This provides an objective view that can be updated regardless of the status of the consumer itself. This checking must be performed for every partition that the consumer group consumes. For a large consumer, like MirrorMaker, this may mean tens of thousands of partitions.

Chapter 12 provided information on using the command-line utilities to get consumer group information, including committed offsets and lag. Monitoring lag like this, however, presents its own problems. First, you must understand for each partition what is a reasonable amount of lag. A topic that receives 100 messages an hour will need a different threshold than a topic that receives 100,000 messages per second. Then, you must be able to consume all of the lag metrics into a monitoring system and set alerts on them. If you have a consumer group that consumes 100,000 partitions over 1,500 topics, you may find this to be a daunting task.

One way to monitor consumer groups reduce this complexity is to use **Burrow**. This is an open source application, originally developed by LinkedIn, that provides consumer status monitoring by gathering lag information for all consumer groups in a cluster and calculating a single status for each group saying whether the consumer group is working properly, falling behind, or is stalled or stopped entirely. It does this without requiring thresholds by monitoring the progress that the consumer group is making on processing messages, though you can also get the message lag as an absolute number. There is an in-depth discussion of the reasoning and methodology behind how Burrow works on the [LinkedIn Engineering blog](#). Deploying Burrow can be an easy way to provide monitoring for all consumers in a cluster, as well as in

multiple clusters, and it can be easily integrated with your existing monitoring and alerting system.

If there is no other option, the `records-lag-max` metric from the consumer client will provide at least a partial view of the consumer status. It is strongly suggested, however, that you utilize an external monitoring system like Burrow.

End-to-End Monitoring

Another type of external monitoring that is recommended to determine if your Kafka clusters are working properly is an end-to-end monitoring system that provides a client point of view on the health of the Kafka cluster. Consumer and producer clients have metrics that can indicate that there might be a problem with the Kafka cluster, but this can be a guessing game as to whether increased latency is due to a problem with the client, the network, or Kafka itself. In addition, it means that if you are responsible for running the Kafka cluster, and not the clients, you would now have to monitor all of the clients as well. What you really need to know is:

- Can I produce messages to the Kafka cluster?
- Can I consume messages from the Kafka cluster?

In an ideal world, you would be able to monitor this for every topic individually. However, in most situations it is not reasonable to inject synthetic traffic into every topic in order to do this. We can, however, at least provide those answers for every broker in the cluster, and that is what [Xinfra Monitor \(formerly known as Kafka Monitor\) does](#). This tool, open sourced by the Kafka team at LinkedIn, continually produces and consumes data from a topic that is spread across all brokers in a cluster. It measures the availability of both produce and consume requests on each broker, as well as the total produce to consume latency. This type of monitoring is invaluable to be able to externally verify that the Kafka cluster is operating as intended, since just like consumer lag monitoring, the Kafka broker cannot report whether or not clients are able to use the cluster properly.

Summary

Monitoring is a key aspect of running Apache Kafka properly, which explains why so many teams spend a significant amount of their time perfecting that part of operations. Many organizations use Kafka to handle petabyte-scale data flows. Assuring that the data does not stop, and that messages are not lost, this is a critical business requirement. It is also our responsibility to assist users with monitoring how their applications use Kafka by providing the metrics that they need to do this.

In this chapter we covered the basics of how to monitor Java applications, and specifically the Kafka applications. We reviewed a subset of the numerous metrics available in the Kafka broker, also touching on Java and OS monitoring, as well as logging. We then detailed the monitoring available in the Kafka client libraries, including quota monitoring. Finally, we discussed the use of external monitoring systems for consumer lag monitoring and end-to-end cluster availability. While certainly not an exhaustive list of the metrics that are available, this chapter reviewed the most critical ones to keep an eye on.

Stream Processing

Kafka was traditionally seen as a powerful message bus, capable of delivering streams of events but without processing or transformation capabilities. Kafka's reliable stream delivery capabilities make it a perfect source of data for stream processing systems. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza, and many more stream processing systems were built with Kafka often being their only reliable data source.

With the increased popularity of Apache Kafka, first as a simple message bus and later as a data integration system, many companies had a system containing many streams of interesting data, stored for long amounts of time and perfectly ordered, just waiting for some stream processing framework to show up and process them. In other words, in the same way that data processing was significantly more difficult before databases were invented, stream processing was held back by the lack of a stream processing platform.

Starting from version 0.10.0, Kafka does more than provide a reliable source of data streams to every popular stream processing framework. Now Kafka includes a powerful stream processing library as part of its collection of client libraries, called Kafka Streams (or sometimes Streams API). This allows developers to consume, process, and produce events in their own apps, without relying on an external processing framework.

We'll begin the chapter by explaining what we mean by stream processing (since this term is frequently misunderstood), then discuss some of the basic concepts of stream processing and the design patterns that are common to all stream processing systems. We'll then dive into Apache Kafka's stream processing library—its goals and architecture. We'll give a small example of how to use Kafka Streams to calculate a moving average of stock prices. We'll then discuss other examples of good stream processing

use cases and finish off the chapter by providing a few criteria you can use when choosing which stream processing framework (if any) to use with Apache Kafka.

This chapter is intended as just a quick introduction to the large and fascinating world of stream processing and Kafka Streams. There are entire books written on these subjects.

Some books cover the basic concepts of stream processing from a data architecture perspective:

- *Making Sense of Stream Processing* by Martin Kleppmann (O'Reilly) discusses the benefits of rethinking applications as stream processing applications and how to reorient data architectures around the idea of event streams.
- *Streaming Systems* by Tyler Akidau, Slava Chernyak, and Reuven Lax (O'Reilly) is a great general introduction to the topic of stream processing and some of the basic ideas in the space.
- *Flow Architectures* by James Urquhart (O'Reilly) is targeted at CTOs and discusses the implications of stream processing to the business.

Other books go into specific details of specific frameworks:

- *Mastering Kafka Streams and ksqlDB* by Mitch Seymour (O'Reilly)
- *Kafka Streams in Action* by William P. Bejeck Jr. (Manning)
- *Event Streaming with Kafka Streams and ksqlDB* by William P. Bejeck Jr. (Manning)
- *Stream Processing with Apache Flink* by Fabian Hueske and Vasiliki Kalavri (O'Reilly)
- *Stream Processing with Apache Spark* by Gerard Maas and Francois Garillot (O'Reilly)

Finally, Kafka Streams is still an evolving framework. Every major release deprecates APIs and modifies semantics. This chapter documents APIs and semantics as of Apache Kafka 2.8. We avoided using any API that was planned for deprecation in release 3.0, but our discussion of join semantics and timestamp handling does not include any of the changes planned for release 3.0.

What Is Stream Processing?

There is a lot of confusion about what stream processing means. Many definitions mix up implementation details, performance requirements, data models, and many other aspects of software engineering. A similar thing has happened in the world of relational databases—the abstract definitions of the relational model are getting

forever entangled in the implementation details and specific limitations of the popular database engines.

The world of stream processing is still evolving, and just because a specific popular implementation does things in specific ways or has specific limitations doesn't mean that those details are an inherent part of processing streams of data.

Let's start at the beginning: What is a data stream (also called an *event stream* or *streaming data*)? First and foremost, a *data stream* is an abstraction representing an unbounded dataset. *Unbounded* means infinite and ever growing. The dataset is unbounded because over time, new records keep arriving. This definition is used by [Google](#), [Amazon](#), and pretty much everyone else.

Note that this simple model (a stream of events) can be used to represent just about every business activity we care to analyze. We can look at a stream of credit card transactions, stock trades, package deliveries, network events going through a switch, events reported by sensors in manufacturing equipment, emails sent, moves in a game, etc. The list of examples is endless because pretty much everything can be seen as a sequence of events.

There are a few other attributes of the event streams model, in addition to its unbounded nature:

Event streams are ordered

There is an inherent notion of which events occur before or after other events. This is clearest when looking at financial events. A sequence in which you first put money in your account and later spend the money is very different from a sequence at which you first spend the money and later cover your debt by depositing money back. The latter will incur overdraft charges, while the former will not. Note that this is one of the differences between an event stream and a database table—records in a table are always considered unordered, and the “order by” clause of SQL is not part of the relational model; it was added to assist in reporting.

Immutable data records

Events, once occurred, can never be modified. A financial transaction that is canceled does not disappear. Instead, an additional event is written to the stream, recording a cancellation of a previous transaction. When a customer returns merchandise to a shop, we don't delete the fact that the merchandise was sold to them earlier, rather we record the return as an additional event. This is another difference between a data stream and a database table—we can delete or update records in a table, but those are all additional transactions that occur in the database, and as such can be recorded in a stream of events that records all transactions. If you are familiar with binlogs, WALs, or redo logs in databases, you can see that if we insert a record into a table and later delete it, the table will no

longer contain the record, but the redo log will contain two transactions—the insert and the delete.

Event streams are replayable

This is a desirable property. While it is easy to imagine nonreplayable streams (TCP packets streaming through a socket are generally nonreplayable), for most business applications, it is critical to be able to replay a raw stream of events that occurred months (and sometimes years) earlier. This is required to correct errors, try new methods of analysis, or perform audits. This is the reason we believe Kafka made stream processing so successful in modern businesses—it allows capturing and replaying a stream of events. Without this capability, stream processing would not be more than a lab toy for data scientists.

It is worth noting that neither the definition of event streams nor the attributes we later listed say anything about the data contained in the events or the number of events per second. The data differs from system to system—events can be tiny (sometimes only a few bytes) or very large (XML messages with many headers); they can also be completely unstructured key-value pairs, semi-structured JSON, or structured Avro or Protobuf messages. While it is often assumed that data streams are “big data” and involve millions of events per second, the same techniques we’ll discuss apply equally well (and often better) to smaller streams of events with only a few events per second or minute.

Now that we know what event streams are, it’s time to make sure we understand stream processing. Stream processing refers to the ongoing processing of one or more event streams. Stream processing is a programming paradigm—just like request-response and batch processing. Let’s look at how different programming paradigms compare to get a better understanding of how stream processing fits into software architectures:

Request-response

This is the lowest-latency paradigm, with response times ranging from submilliseconds to a few milliseconds, usually with the expectation that response times will be highly consistent. The mode of processing is usually blocking—an app sends a request and waits for the processing system to respond. In the database world, this paradigm is known as *online transaction processing* (OLTP). Point-of-sale systems, credit card processing, and time-tracking systems typically work in this paradigm.

Batch processing

This is the high-latency/high-throughput option. The processing system wakes up at set times—every day at 2:00 a.m., every hour on the hour, etc. It reads all required input (either all data available since the last execution, all data from the beginning of month, etc.), writes all required output, and goes away until the next time it is scheduled to run. Processing times range from minutes to hours,

and users expect to read stale data when they are looking at results. In the database world, these are the data warehouse and business intelligence systems—data is loaded in huge batches once a day, reports are generated, and users look at the same reports until the next data load occurs. This paradigm often has great efficiency and economy of scale, but in recent years, businesses need the data available in shorter timeframes in order to make decision-making more timely and efficient. This puts huge pressure on systems that were written to exploit economy of scale—not to provide low-latency reporting.

Stream processing

This is a continuous and nonblocking option. Stream processing fills the gap between the request-response world, where we wait for events that take two milliseconds to process, and the batch processing world, where data is processed once a day and takes eight hours to complete. Most business processes don't require an immediate response within milliseconds but can't wait for the next day either. Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response within milliseconds. Business processes such as alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand, or tracking deliveries of packages are all a natural fit for continuous but nonblocking processing.

It is important to note that the definition doesn't mandate any specific framework, API, or feature. As long as we are continuously reading data from an unbounded dataset, doing something to it, and emitting output, we are doing stream processing. But the processing has to be continuous and ongoing. A process that starts every day at 2:00 a.m., reads 500 records from the stream, outputs a result, and goes away doesn't quite cut it as far as stream processing goes.

Stream Processing Concepts

Stream processing is very similar to any type of data processing—we write code that receives data, does something with the data (a few transformations, aggregates, enrichments, etc.), and then place the result somewhere. However, there are some key concepts that are unique to stream processing and often cause confusion when someone who has data processing experience first attempts to write stream processing applications. Let's take a look at a few of those concepts.

Topology

A stream processing application includes one or more processing topologies. A processing topology starts with one or more source streams that are passed through a graph of stream processors connected through event streams, until results are written

to one or more sink streams. Each stream processor is a computational step applied to the stream of events in order to transform the events. Examples of some stream processors we'll use in our examples are filter, count, group-by, and left-join. We often visualize stream processing applications by drawing the processing nodes and connecting them with arrows to show how events flow from one node to the next as the application is processing data.

Time

Time is probably the most important concept in stream processing and often the most confusing. For an idea of how complex time can get when discussing distributed systems, we recommend Justin Sheehy's excellent paper, "[There Is No Now](#)". In the context of stream processing, having a common notion of time is critical because most stream applications perform operations on time windows. For example, our stream application might calculate a moving five-minute average of stock prices. In that case, we need to know what to do when one of our producers goes offline for two hours due to network issues and returns with two hours' worth of data—most of the data will be relevant for five-minute time windows that have long passed and for which the result was already calculated and stored.

Stream processing systems typically refer to the following notions of time:

Event time

This is the time the events we are tracking occurred and the record was created—the time a measurement was taken, an item was sold at a shop, a user viewed a page on our website, etc. In version 0.10.0 and later, Kafka automatically adds the current time to producer records at the time they are created. If this does not match the application's notion of *event time*, such as in cases where the Kafka record is created based on a database record sometime after the event occurred, then we recommend adding the event time as a field in the record itself so that both timestamps will be available for later processing. Event time is usually the time that matters most when processing stream data.

Log append time

This is the time the event arrived at the Kafka broker and was stored there, also called *ingestion time*. In version 0.10.0 and later, Kafka brokers will automatically add this time to records they receive if Kafka is configured to do so or if the records arrive from older producers and contain no timestamps. This notion of time is typically less relevant for stream processing, since we are usually interested in the times the events occurred. For example, if we calculate the number of devices produced per day, we want to count devices that were actually produced on that day, even if there were network issues and the event only arrived to Kafka the following day. However, in cases where the real event time was not recorded, log append time can still be used consistently because it does not change after the

record was created, and assuming no delays in the pipeline, it can be a reasonable approximation of event time.

Processing time

This is the time at which a stream processing application received the event in order to perform some calculation. This time can be milliseconds, hours, or days after the event occurred. This notion of time assigns different timestamps to the same event depending on exactly when each stream processing application happened to read the event. It can even differ for two threads in the same application! Therefore, this notion of time is highly unreliable and best avoided.

Kafka Streams assigns time to each event based on the `TimestampExtractor` interface. Developers of Kafka Streams applications can use different implementations of this interface, which can use either of the three time semantics explained previously or a completely different choice of timestamp, including extracting a timestamp from the contents of the event itself.

When Kafka Streams writes output to a Kafka topic, it assigns a timestamp to each event based on the following rules:

- When the output record maps directly to an input record, the output record will use the same timestamp as the input.
- When the output record is a result of an aggregation, the timestamp of the output record will be the maximum timestamp used in the aggregation.
- When the output record is a result of joining two streams, the timestamp of the output record is the largest of the two records being joined. When a stream and a table are joined, the timestamp from the stream record is used.
- Finally, if the output record was generated by a Kafka Streams function that generates data in a specific schedule regardless of input, such as `punctuate()`, the output timestamp will depend on the current internal times of the stream processing app.

When using the Kafka Streams lower-level processing API rather than the DSL, Kafka Streams includes APIs for manipulating the timestamps of records directly, so developers can implement timestamp semantics that match the required business logic of the application.



Mind the Time Zone

When working with time, it is important to be mindful of time zones. The entire data pipeline should standardize on a single time zone; otherwise, results of stream operations will be confusing and often meaningless. If you must handle data streams with different time zones, you need to make sure you can convert events to a single time zone before performing operations on time windows. Often this means storing the time zone in the record itself.

State

As long as we only need to process each event individually, stream processing is a very simple activity. For example, if all we need to do is read a stream of online shopping transactions from Kafka, find the transactions over \$10,000, and email the relevant salesperson, we can probably write this in just few lines of code using a Kafka consumer and SMTP library.

Stream processing becomes really interesting when we have operations that involve multiple events: counting the number of events by type, moving averages, joining two streams to create an enriched stream of information, etc. In those cases, it is not enough to look at each event by itself; we need to keep track of more information—how many events of each type did we see this hour, all events that require joining, sums, averages, etc. We call this information a *state*.

It is often tempting to store the state in variables that are local to the stream processing app, such as a simple hash table to store moving counts. In fact, we did just that in many examples in this book. However, this is not a reliable approach for managing state in stream processing because when the stream processing application is stopped or crashes, the state is lost, which changes the results. This is usually not the desired outcome, so care should be taken to persist the most recent state and recover it when restarting the application.

Stream processing refers to several types of state:

Local or internal state

State that is accessible only by a specific instance of the stream processing application. This state is usually maintained and managed with an embedded, in-memory database running within the application. The advantage of local state is that it is extremely fast. The disadvantage is that we are limited to the amount of memory available. As a result, many of the design patterns in stream processing focus on ways to partition the data into substreams that can be processed using a limited amount of local state.

External state

State that is maintained in an external data store, often a NoSQL system like Cassandra. The advantages of an external state are its virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications. The downside is the extra latency and complexity introduced with an additional system, as well as availability—the application needs to handle the possibility that the external system is not available. Most stream processing apps try to avoid having to deal with an external store, or at least limit the latency overhead by caching information in the local state and communicating with the external store as rarely as possible. This usually introduces challenges with maintaining consistency between the internal and external state.

Stream-Table Duality

We are all familiar with database tables. A *table* is a collection of records, each identified by its primary key and containing a set of attributes as defined by a schema. Table records are mutable (i.e., tables allow update and delete operations). Querying a table allows checking the state of the data at a specific point in time. For example, by querying the CUSTOMERS_CONTACTS table in a database, we expect to find current contact details for all our customers. Unless the table was specifically designed to include history, we will not find their past contacts in the table.

Unlike tables, streams contain a history of changes. A *stream* is a string of events wherein each event caused a change. A table contains a current state of the world, which is the result of many changes. From this description, it is clear that streams and tables are two sides of the same coin—the world always changes, and sometimes we are interested in the events that caused those changes, whereas other times we are interested in the current state of the world. Systems that allow us to transition back and forth between the two ways of looking at data are more powerful than systems that support just one.

To convert a table to a stream, we need to capture the changes that modify the table. Take all those `insert`, `update`, and `delete` events and store them in a stream. Most databases offer change data capture (CDC) solutions for capturing these changes, and there are many Kafka connectors that can pipe those changes into Kafka where they will be available for stream processing.

To convert a stream to a table, we need to apply all the changes that the stream contains. This is also called *materializing* the stream. We create a table, either in memory, in an internal state store, or in an external database, and start going over all the events in the stream from beginning to end, changing the state as we go. When we finish, we have a table representing a state at a specific time that we can use.

Suppose we have a store selling shoes. A stream representation of our retail activity can be a stream of events:

“Shipment arrived with red, blue, and green shoes.”
“Blue shoes sold.”
“Red shoes sold.”
“Blue shoes returned.”
“Green shoes sold.”

If we want to know what our inventory contains right now or how much money we made until now, we need to materialize the view. [Figure 14-1](#) shows that we currently have 299 red shoes. If we want to know how busy the store is, we can look at the entire stream and see that there were four customer events today. We may also want to investigate why the blue shoes were returned.

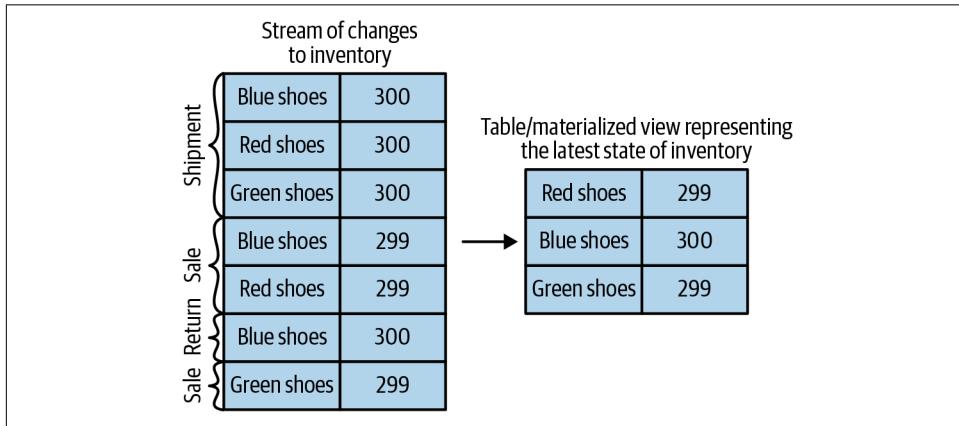


Figure 14-1. Materializing inventory changes

Time Windows

Most operations on streams are windowed operations, operating on slices of time: moving averages, top products sold this week, 99th percentile load on the system, etc. Join operations on two streams are also windowed—we join events that occurred at the same slice of time. Very few people stop and think about the type of window they want for their operations. For example, when calculating moving averages, we want to know:

Size of the window

Do we want to calculate the average of all events in every five-minute window? Every 15-minute window? Or the entire day? Larger windows are smoother but they lag more—if the price increases, it will take longer to notice than with a smaller window. Kafka Streams also includes a *session window*, where the size of

the window is defined by a period of inactivity. The developer defines a session gap, and all events that arrive continuously with gaps smaller than the defined session gap belong to the same session. A gap in arrivals will define a new session, and all events arriving after the gap, but before the next gap, will belong to the new session.

How often the window moves (advance interval)

Five-minute averages can update every minute, second, or every time there is a new event. Windows for which the size is a fixed time interval are called *hopping windows*. When the advance interval is equal to the window size, it is called a *tumbling window*.

How long the window remains updatable (grace period)

Our five-minute moving average calculated the average for the 00:00–00:05 window. Now, an hour later, we are getting a few more input records with their *event time* showing 00:02. Do we update the result for the 00:00–00:05 period? Or do we let bygones be bygones? Ideally, we'll be able to define a certain time period during which events will get added to their respective time slice. For example, if the events were delayed up to four hours, we should recalculate the results and update. If events arrive later than that, we can ignore them.

Windows can be aligned to clock time—i.e., a five-minute window that moves every minute will have the first slice as 00:00–00:05 and the second as 00:01–00:06. Or it can be unaligned and simply start whenever the app started, and then the first slice can be 03:17–03:22. See [Figure 14-2](#) for the difference between these two types of windows.

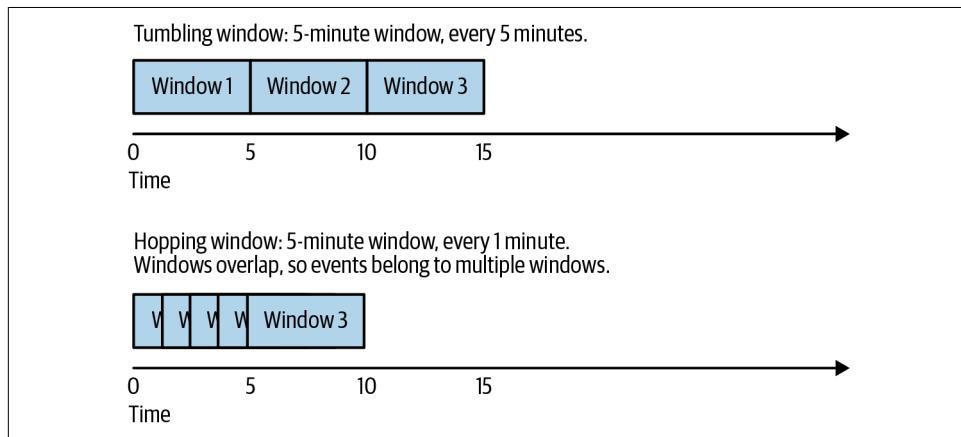


Figure 14-2. Tumbling window versus hopping window

Processing Guarantees

A key requirement for stream processing applications is the ability to process each record exactly once, regardless of failures. Without exactly-once guarantees, stream processing can't be used in cases where accurate results are needed. As discussed in detail in [Chapter 8](#), Apache Kafka has support for exactly-once semantics with a transactional and idempotent producer. Kafka Streams uses Kafka's transactions to implement exactly-once guarantees for stream processing applications. Every application that uses the Kafka Streams library can enable exactly-once guarantees by setting `processing.guarantee` to `exactly_once`. Kafka Streams version 2.6 or later includes a more efficient exactly-once implementation that requires Kafka brokers of version 2.5 or later. This efficient implementation can be enabled by setting `processing.guarantee` to `exactly_once_beta`.

Stream Processing Design Patterns

Every stream processing system is different—from the basic combination of a consumer, processing logic, and producer, to involved clusters like Spark Streaming with its machine learning libraries, and much in between. But there are some basic design patterns, which are known solutions to common requirements of stream processing architectures. We'll review a few of those well-known patterns and show how they are used with a few examples.

Single-Event Processing

The most basic pattern of stream processing is the processing of each event in isolation. This is also known as a *map/filter pattern* because it is commonly used to filter unnecessary events from the stream or transform each event. (The term *map* is based on the map/reduce pattern in which the map stage transforms events and the reduce stage aggregates them.)

In this pattern, the stream processing app consumes events from the stream, modifies each event, and then produces the events to another stream. An example is an app that reads log messages from a stream and writes ERROR events into a high-priority stream and the rest of the events into a low-priority stream. Another example is an application that reads events from a stream and modifies them from JSON to Avro. Such applications do not need to maintain state within the application because each event can be handled independently. This means that recovering from app failures or load-balancing is incredibly easy as there is no need to recover state; we can simply hand off the events to another instance of the app to process.

This pattern can be easily handled with a simple producer and consumer, as seen in [Figure 14-3](#).

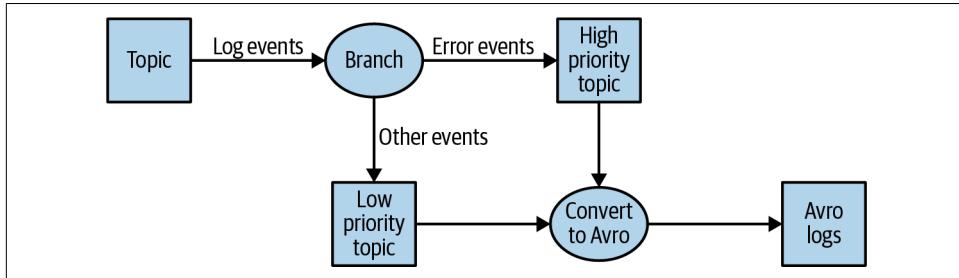


Figure 14-3. Single-event processing topology

Processing with Local State

Most stream processing applications are concerned with aggregating information, especially window aggregation. An example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.

These aggregations require maintaining a *state*. In our example, in order to calculate the minimum and average price each day, we need to store the minimum value, the sum, and the number of records we've seen up until the current time.

All this can be done using *local state* (rather than a shared state) because each operation in our example is a *group by aggregate*. That is, we perform the aggregation per stock symbol, not on the entire stock market in general. We use a Kafka partitioner to make sure that all events with the same stock symbol are written to the same partition. Then, each instance of the application will get all the events from the partitions that are assigned to it (this is a Kafka consumer guarantee). This means that each instance of the application can maintain state for the subset of stock symbols that are written to the partitions that are assigned to it. See [Figure 14-4](#).

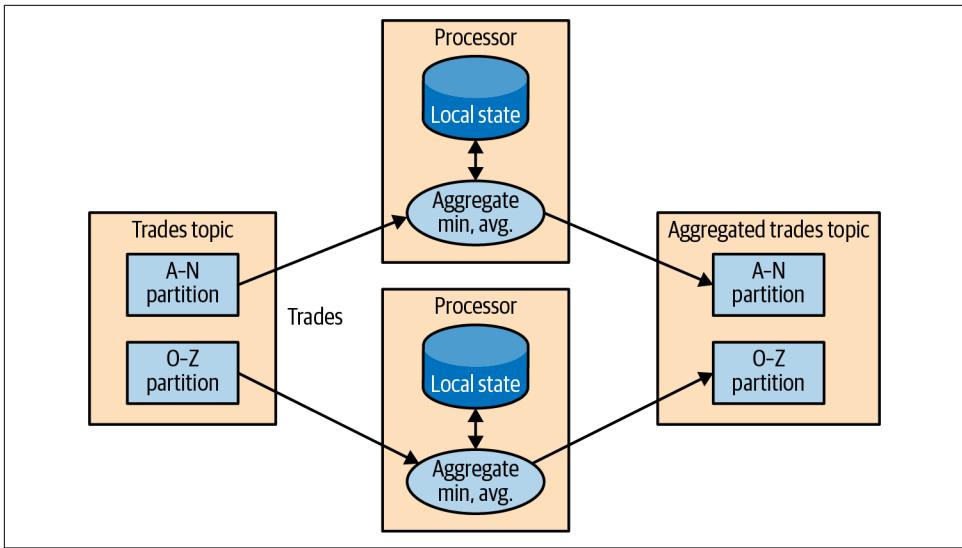


Figure 14-4. Topology for event processing with local state

Stream processing applications become significantly more complicated when the application has local state. There are several issues a stream processing application must address:

Memory usage

The local state ideally fits into the memory available to the application instance. Some local stores allow spilling to disk, but this has significant performance impact.

Persistence

We need to make sure the state is not lost when an application instance shuts down and that the state can be recovered when the instance starts again or is replaced by a different instance. This is something that Kafka Streams handles very well—local state is stored in-memory using embedded RocksDB, which also persists the data to disk for quick recovery after restarts. But all the changes to the local state are also sent to a Kafka topic. If a stream's node goes down, the local state is not lost—it can be easily re-created by rereading the events from the Kafka topic. For example, if the local state contains “current minimum for IBM = 167.19,” we store this in Kafka so that later we can repopulate the local cache from this data. Kafka uses log compaction for these topics to make sure they don't grow endlessly and that re-creating the state is always feasible.

Rebalancing

Partitions sometimes get reassigned to a different consumer. When this happens, the instance that loses the partition must store the last good state, and the instance that receives the partition must know to recover the correct state.

Stream processing frameworks differ in how much they help the developer manage the local state they need. If our application requires maintaining local state, we make sure to check the framework and its guarantees. We'll include a short comparison guide at the end of the chapter, but as we all know, software changes quickly and stream processing frameworks doubly so.

Multiphase Processing/Repartitioning

Local state is great if we need a *group by* type of aggregate. But what if we need a result that uses all available information? For example, suppose we want to publish the top 10 stocks each day—the 10 stocks that gained the most from opening to closing during each day of trading. Obviously, nothing we do locally on each application instance is enough because all the top 10 stocks could be in partitions assigned to other instances. What we need is a two-phase approach. First, we calculate the daily gain/loss for each stock symbol. We can do this on each instance with a local state. Then we write the results to a new topic with a single partition. This partition will be read by a single application instance that can then find the top 10 stocks for the day. The second topic, which contains just the daily summary for each stock symbol, is obviously much smaller with significantly less traffic than the topics that contain the trades themselves, and therefore it can be processed by a single instance of the application. Sometimes more steps are needed to produce the result. See [Figure 14-5](#).

This type of multiphase processing is very familiar to those who write MapReduce code, where you often have to resort to multiple reduce phases. If you've ever written map-reduce code, you'll remember that you needed a separate app for each reduce step. Unlike MapReduce, most stream processing frameworks allow including all steps in a single app, with the framework handling the details of which application instance (or worker) will run each step.

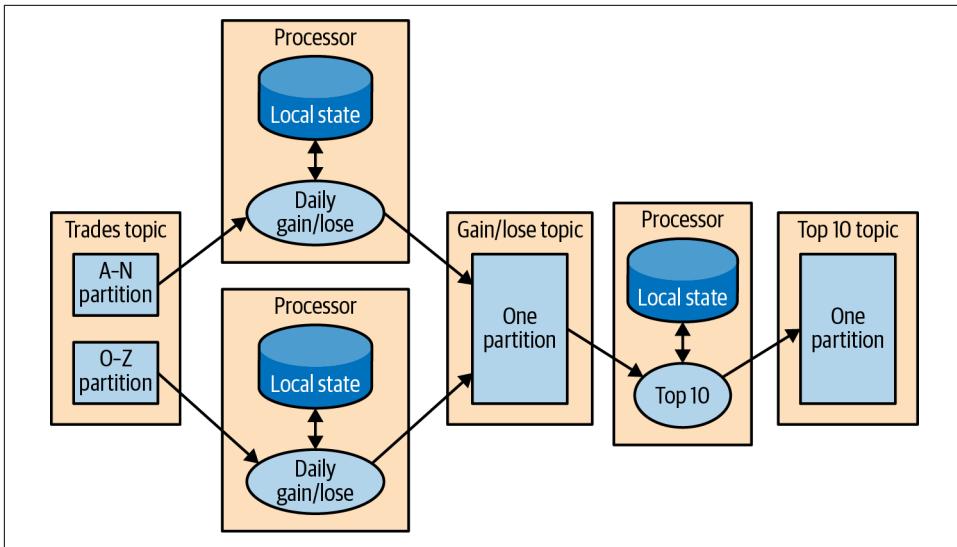


Figure 14-5. Topology that includes both local state and repartitioning steps

Processing with External Lookup: Stream-Table Join

Sometimes stream processing requires integration with data external to the stream—validating transactions against a set of rules stored in a database or enriching click-stream information with data about the users who clicked.

The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click, plus the user age and gender, to another topic. See [Figure 14-6](#).

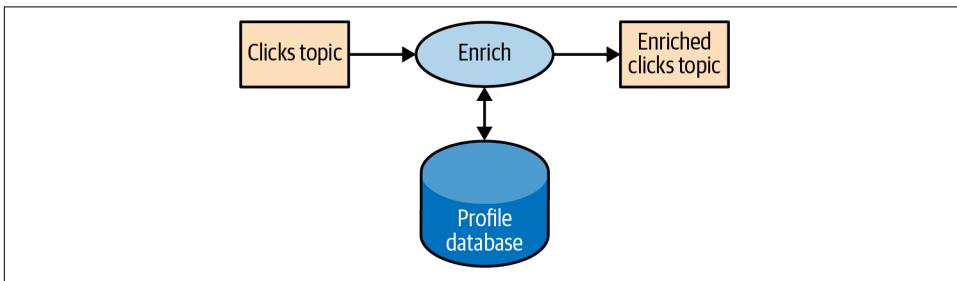


Figure 14-6. Stream processing that includes an external data source

The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record—usually between 5 and 15 milliseconds. In many cases, this is not feasible. Often the additional load this places on the external data store is also not acceptable—stream processing systems can often handle 100K–500K

events per second, but the database can only handle perhaps 10K events per second at reasonable performance. There is also added complexity around availability—our application will need to handle situations when the external DB is not available.

To get good performance and availability, we need to cache the information from the database in our stream processing application. Managing this cache can be challenging though—how do we prevent the information in the cache from getting stale? If we refresh events too often, we are still hammering the database, and the cache isn't helping much. If we wait too long to get new events, we are doing stream processing with stale information.

But if we can capture all the changes that happen to the database table in a stream of events, we can have our stream processing job listen to this stream and update the cache based on database change events. Capturing changes to the database as events in a stream is known as *change data capture* (CDC), and Kafka Connect has multiple connectors capable of performing CDC and converting database tables to a stream of change events. This allows us to keep our own private copy of the table and be notified whenever there is a database change event so we can update our own copy accordingly. See [Figure 14-7](#).

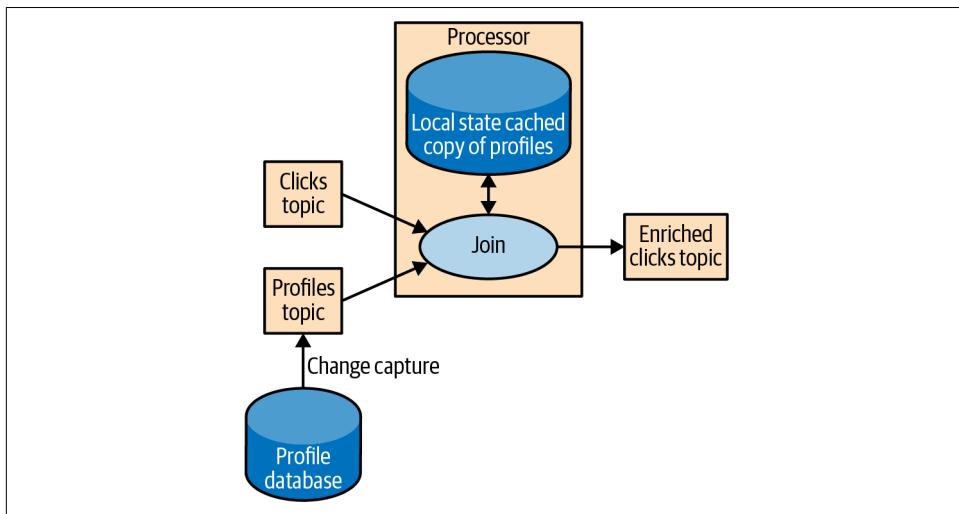


Figure 14-7. Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing

Then, when we get click events, we can look up the `user_id` at our local state and enrich the event. And because we are using a local state, this scales a lot better and will not affect the database and other apps using it.

We refer to this as a *stream-table join* because one of the streams represents changes to a locally cached table.

Table-Table Join

In the previous section we discussed how a table and a stream of update events are equivalent. We've discussed in detail how this works when joining a stream and a table. There is no reason why we can't have those materialized tables in both sides of the join operation.

Joining two tables is always nonwindowed and joins the current state of both tables at the time the operation is performed. With Kafka Streams, we can perform an **equijoin**, in which both tables have the same key that is partitioned in the same way, and therefore the join operation can be efficiently distributed between a large number of application instances and machines.

Kafka Streams also supports **foreign-key join** of two tables—the key of one stream or table is joined with an arbitrary field from another stream or table. You can learn more about how it works in [“Crossing the Streams”](#), a talk from Kafka Summit 2020, or the more in-depth [blog post](#).

Streaming Join

Sometimes we want to join two real event streams rather than a stream with a table. What makes a stream “real”? If you recall the discussion at the beginning of the chapter, streams are unbounded. When we use a stream to represent a table, we can ignore most of the history in the stream because we only care about the current state in the table. But when we join two streams, we are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time windows. This is why a streaming join is also called a *windowed join*.

For example, let's say that we have one stream with search queries that people entered into our website and another stream with clicks, which include clicks on search results. We want to match search queries with the results they clicked on so that we will know which result is most popular for which query. Obviously, we want to match results based on the search term but only match them within a certain time window. We assume the result is clicked seconds after the query was entered into our search engine. So we keep a small, few-seconds-long window on each stream and match the results from each window. See [Figure 14-8](#).

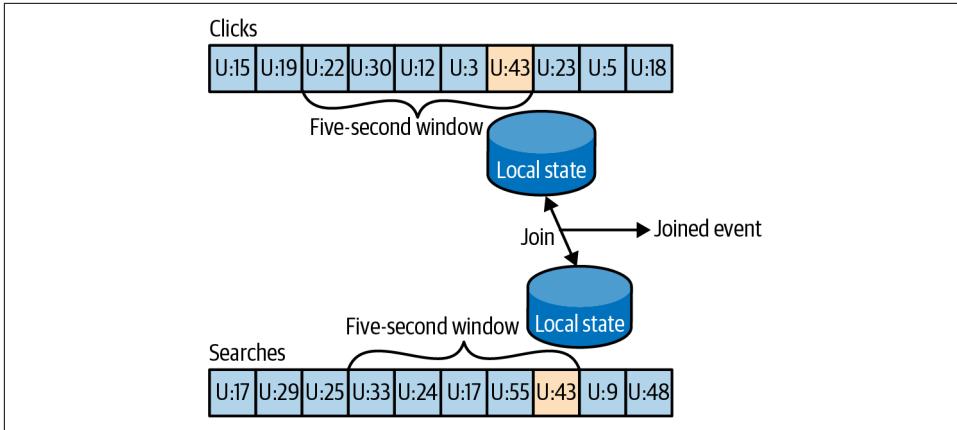


Figure 14-8. Joining two streams of events; these joins always involve a moving time window

Kafka Streams supports **equi-joins**, where streams, queries, and clicks are partitioned on the same keys, which are also the join keys. This way, all the click events from `user_id:42` end up in partition 5 of the clicks topic, and all the search events for `user_id:42` end up in partition 5 of the search topic. Kafka Streams then makes sure that partition 5 of both topics is assigned to the same task. So this task sees all the relevant events for `user_id:42`. It maintains the join window for both topics in its embedded RocksDB state store, and this is how it can perform the join.

Out-of-Sequence Events

Handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems. Out-of-sequence events happen quite frequently and expectedly in IoT scenarios ([Figure 14-9](#)). For example, a mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects. This also happens when monitoring network equipment (a faulty switch doesn't send diagnostics signals until it is repaired) or manufacturing (network connectivity in plants is notoriously unreliable, especially in developing countries).

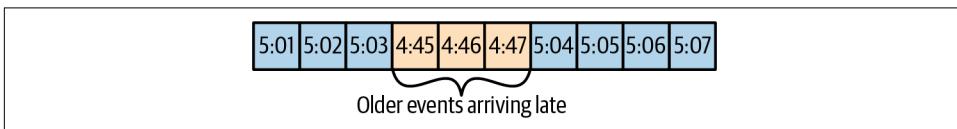


Figure 14-9. Out-of-sequence events

Our streams applications need to be able to handle those scenarios. This typically means the application has to do the following:

- Recognize that an event is out of sequence—this requires that the application examines the event time and discovers that it is older than the current time.
- Define a time period during which it will attempt to reconcile out-of-sequence events. Perhaps a three-hour delay should be reconciled, and events over three weeks old can be thrown away.
- Have an in-band capability to reconcile this event. This is the main difference between streaming apps and batch jobs. If we have a daily batch job and a few events arrived after the job completed, we can usually just rerun yesterday's job and update the events. With stream processing, there is no “rerun yesterday's job”—the same continuous process needs to handle both old and new events at any given moment.
- Be able to update results. If the results of the stream processing are written into a database, a *put* or *update* is enough to update the results. If the stream app sends results by email, updates may be trickier.

Several stream processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time, and the ability to handle events with event times that are older or newer than the current processing time. This is typically done by maintaining multiple aggregation windows available for update in the local state and giving developers the ability to configure how long to keep those window aggregates available for updates. Of course, the longer the aggregation windows are kept available for updates, the more memory is required to maintain the local state.

The Kafka Streams API always writes aggregation results to result topics. Those are usually **compacted topics**, which means that only the latest value for each key is preserved. In case the results of an aggregation window need to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will effectively replace the previous result.

Reprocessing

The last important pattern is reprocessing events. There are two variants of this pattern:

- We have an improved version of our stream processing application. We want to run the new version of the application on the same event stream as the old, produce a new stream of results that does not replace the first version, compare the results between the two versions, and at some point move clients to use the new results instead of the existing ones.

- The existing stream processing app is buggy. We fix the bug, and we want to reprocess the event stream and recalculate our results

The first use case is made simple by the fact that Apache Kafka stores the event streams in their entirety for long periods of time in a scalable data store. This means that having two versions of a stream processing application writing two result streams only requires the following:

- Spinning up the new version of the application as a new consumer group
- Configuring the new version to start processing from the first offset of the input topics (so it will get its own copy of all events in the input streams)
- Letting the new application continue processing, and switching the client applications to the new result stream when the new version of the processing job has caught up

The second use case is more challenging—it requires “resetting” an existing app to start processing back at the beginning of the input streams, resetting the local state (so we won’t mix results from the two versions of the app), and possibly cleaning the previous output stream. While Kafka Streams has a tool for resetting the state for a stream processing app, our recommendation is to try to use the first method whenever sufficient capacity exists to run two copies of the app and generate two result streams. The first method is much safer—it allows switching back and forth between multiple versions and comparing results between versions, and doesn’t risk losing critical data or introducing errors during the cleanup process.

Interactive Queries

As discussed previously, stream processing applications have state, and this state can be distributed among many instances of the application. Most of the time the users of stream processing applications get the results of the processing by reading them from an output topic. In some cases, however, it is desirable to take a shortcut and read the results from the state store itself. This is common when the result is a table (e.g., the top 10 best-selling books) and the stream of results is really a stream of updates to this table—it is much faster and easier to just read the table directly from the stream processing application state.

Kafka Streams includes flexible APIs for [querying the state of a stream processing application](#).

Kafka Streams by Example

To demonstrate how these patterns are implemented in practice, we'll show a few examples using the Apache Kafka Streams API. We are using this specific API because it is relatively simple to use and it ships with Apache Kafka, which we already have access to. It is important to remember that the patterns can be implemented in any stream processing framework and library—the patterns are universal, but the examples are specific.

Apache Kafka has two stream APIs—a low-level Processor API and a high-level Streams DSL. We will use the Kafka Streams DSL in our examples. The DSL allows us to define the stream processing application by defining a chain of transformations to events in the streams. Transformations can be as simple as a filter or as complex as a stream-to-stream join. The lower-level API allows us to create our own transformations. To learn more about the low-level Processor API, the [developer guide](#) has detailed information, and the presentation “[Beyond the DSL](#)” is a great introduction.

An application that uses the DSL API always starts with using the `StreamsBuilder` to create a processing *topology*—a directed acyclic graph (DAG) of transformations that are applied to the events in the streams. Then we create a `KafkaStreams` execution object from the topology. Starting the `KafkaStreams` object will start multiple threads, each applying the processing topology to events in the stream. The processing will conclude when we close the `KafkaStreams` object.

We'll look at a few examples that use Kafka Streams to implement some of the design patterns we just discussed. A simple word count example will be used to demonstrate the map/filter pattern and simple aggregates. Then we'll move to an example where we calculate different statistics on stock market trades, which will allow us to demonstrate window aggregations. Finally, we'll use ClickStream enrichment as an example to demonstrate streaming joins.

Word Count

Let's walk through an abbreviated word count example for Kafka Streams. You can find the full example on [GitHub](#).

The first thing you do when creating a stream processing app is configure Kafka Streams. Kafka Streams has a large number of possible configurations, which we won't discuss here, but you can find them in the [documentation](#). In addition, you can configure the producer and consumer embedded in Kafka Streams by adding any producer or consumer config to the `Properties` object:

```

public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
                  "wordcount"); ①
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
                  "localhost:9092"); ②
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
                  Serdes.String().getClass().getName()); ③
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
                  Serdes.String().getClass().getName());
    }
}

```

- ① Every Kafka Streams application must have an application ID. It is used to coordinate the instances of the application and also when naming the internal local stores and the topics related to them. This name must be unique for each Kafka Streams application working with the same Kafka cluster.
- ② The Kafka Streams application always reads data from Kafka topics and writes its output to Kafka topics. As we'll discuss later, Kafka Streams applications also use Kafka for coordination. So we had better tell our app where to find Kafka.
- ③ When reading and writing data, our app will need to serialize and deserialize, so we provide default Serde classes. If needed, we can override these defaults later when building the streams topology.

Now that we have the configuration, let's build our streams topology:

```

StreamsBuilder builder = new StreamsBuilder(); ①

KStream<String, String> source =
    builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream<String, String> counts = source.flatMapValues(value->
    Arrays.asList(pattern.split(value.toLowerCase())))) ②
    .map((key, value) -> new KeyValue<String,
          String>(value, value))
    .filter((key, value) -> (!value.equals("the")))) ③
    .groupByKey() ④
    .count().mapValues(value->
        Long.toString(value)).toStream(); ⑤
counts.to("wordcount-output"); ⑥

```

- ① We create a `StreamsBuilder` object and start defining a stream by pointing at the topic we'll use as our input.

- ② Each event we read from the source topic is a line of words; we split it up using a regular expression into a series of individual words. Then we take each word (currently a value of the event record) and put it in the event record key so it can be used in a group-by operation.
- ③ We filter out the word *the*, just to show how easy filtering is.
- ④ And we group by key, so we now have a collection of events for each unique word.
- ⑤ We count how many events we have in each collection. The result of counting is a `Long` data type. We convert it to a `String` so it will be easier for humans to read the results.
- ⑥ Only one thing left—write the results back to Kafka.

Now that we have defined the flow of transformations that our application will run, we just need to...run it:

```
KafkaStreams streams = new KafkaStreams(builder.build(), props); ①
streams.start(); ②

// usually the stream application would be running forever,
// in this example we just let it run for some time and stop
Thread.sleep(5000L);

streams.close(); ③
```

- ① Define a `KafkaStreams` object based on our topology and the properties we defined.
- ② Start Kafka Streams.
- ③ After a while, stop it.

That's it! In just a few short lines, we demonstrated how easy it is to implement a single event processing pattern (we applied a map and a filter on the events). We repartitioned the data by adding a group-by operator and then maintained simple local state when we counted the number of records that have each word as a key. Then we maintained simple local state when we counted the number of times each word appeared.

At this point, we recommend running the full example. The [README in the GitHub repository](#) contains instructions on how to run the example.

Note that we can run the entire example on our machine without installing anything except Apache Kafka. If our input topic contains multiple partitions, we can run

multiple instances of the WordCount application (just run the app in several different terminal tabs), and we have our first Kafka Streams processing cluster. The instances of the WordCount application talk to one another and coordinate the work. One of the biggest barriers to entry for some stream processing frameworks is that local mode is very easy to use, but then to run a production cluster, we need to install YARN or Mesos, then install the processing framework on all those machines, and then learn how to submit our app to the cluster. With the Kafka's Streams API, we just start multiple instances of our app—and we have a cluster. The exact same app is running on our development machine and in production.

Stock Market Statistics

The next example is more involved—we will read a stream of stock market trading events that include the stock ticker, ask price, and ask size. In stock market trades, *ask price* is what a seller is asking for, whereas *bid price* is what the buyer is suggesting to pay. *Ask size* is the number of shares the seller is willing to sell at that price. For simplicity of the example, we'll ignore bids completely. We also won't include a timestamp in our data; instead, we'll rely on event time populated by our Kafka producer.

We will then create output streams that contain a few windowed statistics:

- Best (i.e., minimum) ask price for every five-second window
- Number of trades for every five-second window
- Average ask price for every five-second window

All statistics will be updated every second.

For simplicity, we'll assume our exchange only has 10 stock tickers trading in it. The setup and configuration are very similar to those we used in [“Word Count” on page 402](#):

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, Constants.BROKER);
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    TradeSerde.class.getName());
```

The main difference is the Serde classes used. In [“Word Count” on page 402](#), we used strings for both key and value, and therefore used the `Serdes.String()` class as a serializer and deserializer for both. In this example, the key is still a string, but the value is a `Trade` object that contains the ticker symbol, ask price, and ask size. In order to serialize and deserialize this object (and a few other objects we used in this small app), we used the Gson library from Google to generate a JSON serializer and

deserializer from our Java object. Then we created a small wrapper that created a Serde object from those. Here is how we created the Serde:

```
static public final class TradeSerde extends WrapperSerde<Trade> {  
    public TradeSerde() {  
        super(new JsonSerializer<Trade>(),  
              new JsonDeserializer<Trade>(Trade.class));  
    }  
}
```

Nothing fancy, but remember to provide a Serde object for every object you want to store in Kafka—input, output, and, in some cases, intermediate results. To make this easier, we recommend generating these Serdes through a library like Gson, Avro, Protobuf, or something similar.

Now that we have everything configured, it's time to build our topology:

```
KStream<Windowed<String>, TradeStats> stats = source  
    .groupByKey() ❶  
    .windowedBy(TimeWindows.of(Duration.ofMillis(windowSize))  
                .advanceBy(Duration.ofSeconds(1))) ❷  
    .aggregate( ❸  
        () -> new TradeStats(),  
        (k, v, tradestats) -> tradestats.add(v), ❹  
        Materialized.<String, TradeStats, WindowStore<Bytes, byte[]>>  
            as("trade-aggregates") ❺  
            .withValueSerde(new TradeStatsSerde()) ❻  
    .toStream() ❼  
    .mapValues((trade) -> trade.computeAvgPrice()); ❽
```



```
stats.to("stockstats-output",  
        Produced.keySerde(  
            WindowedSerdes.timeWindowedSerdeFrom(String.class, windowSize))); ❾
```

- ❶ We start by reading events from the input topic and performing a `groupByKey()` operation. Despite its name, this operation does not do any grouping. Rather, it ensures that the stream of events is partitioned based on the record key. Since we wrote the data into a topic with a key and didn't modify the key before calling `groupByKey()`, the data is still partitioned by its key—so this method does nothing in this case.
- ❷ We define the window—in this case, a window of five seconds, advancing every second.
- ❸ After we ensure correct partitioning and windowing, we start the aggregation. The `aggregate` method will split the stream into overlapping windows (a five-second window every second) and then apply an aggregate method on all the events in the window. The first parameter this method takes is a new object that will contain the results of the aggregation—`Tradestats`, in our case. This is an

object we created to contain all the statistics we are interested in for each time window—minimum price, average price, and number of trades.

- ④ We then supply a method for actually aggregating the records—in this case, an `add` method of the `Tradestats` object is used to update the minimum price, number of trades, and total prices in the window with the new record.
- ⑤ As mentioned in “[Stream Processing Design Patterns](#)” on page 392, windowing aggregation requires maintaining a state and a local store in which the state will be maintained. The last parameter of the aggregate method is the configuration of the state store. `Materialized` is the store configuration object, and we configure the store name as `trade-aggregates`. This can be any unique name.
- ⑥ As part of the state store configuration, we also provide a Serde object for serializing and deserializing the results of the aggregation (the `Tradestats` object).
- ⑦ The results of the aggregation is a *table* with the ticker and the time window as the primary key and the aggregation result as the value. We are turning the table back into a stream of events.
- ⑧ The last step is to update the average price—right now the aggregation results include the sum of prices and number of trades. We go over these records and use the existing statistics to calculate average price so we can include it in the output stream.
- ⑨ And finally, we write the results back to the `stockstats-output` stream. Since the results are part of a windowing operation, we create a `WindowedSerde` that stores the result in a windowed data format that includes the window timestamp. The window size is passed as part of the Serde, even though it isn’t used in the serialization (deserialization requires the window size, because only the start time of the window is stored in the output topic).

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in “[Word Count](#)” on page 402.

This example shows how to perform windowed aggregation on a stream—probably the most popular use case of stream processing. One thing to notice is how little work was needed to maintain the local state of the aggregation—just provide a Serde and name the state store. Yet this application will scale to multiple instances and automatically recover from a failure of each instance by shifting processing of some partitions to one of the surviving instances. We will see more on how it is done in “[Kafka Streams: Architecture Overview](#)” on page 410.

As usual, you can find the complete example, including instructions for running it, on [GitHub](#).

ClickStream Enrichment

The last example will demonstrate streaming joins by enriching a stream of clicks on a website. We will generate a stream of simulated clicks, a stream of updates to a fictional profile database table, and a stream of web searches. We will then join all three streams to get a 360-degree view into each user activity. What did the users search for? What did they click as a result? Did they change their “interests” in their user profile? These kinds of joins provide a rich data collection for analytics. Product recommendations are often based on this kind of information—the user searched for bikes, clicked on links for “Trek,” and is interested in travel, so we can advertise bikes from Trek, helmets, and bike tours to exotic locations like Nebraska.

Since configuring the app is similar to the previous examples, let’s skip this part and take a look at the topology for joining multiple streams:

```
KStream<Integer, PageView> views =
    builder.stream(Constants.PAGE_VIEW_TOPIC,
        Consumed.with(Serdes.Integer(), new PageViewSerde())); ①
KStream<Integer, Search> searches =
    builder.stream(Constants.SEARCH_TOPIC,
        Consumed.with(Serdes.Integer(), new SearchSerde()));
KTable<Integer, UserProfile> profiles =
    builder.table(Constants.USER_PROFILE_TOPIC,
        Consumed.with(Serdes.Integer(), new ProfileSerde())); ②

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ③
    (page, profile) -> {
        if (profile != null)
            return new UserActivity(
                profile.getUserID(), profile.getUserName(),
                profile.getZipcode(), profile.getInterests(),
                "", page.getPage()); ④
        else
            return new UserActivity(
                -1, "", "", null, "", page.getPage());
    });

KStream<Integer, UserActivity> userActivityKStream =
    viewsWithProfile.leftJoin(searches, ⑤
        (userActivity, search) -> {
            if (search != null)
                userActivity.updateSearch(search.getSearchTerms()); ⑥
            else
                userActivity.updateSearch("");
            return userActivity;
    },
```

```
JoinWindows.of(Duration.ofSeconds(1)).before(Duration.ofSeconds(0)), ⑦  
    StreamJoined.with(Serdes.Integer(), ⑧  
        new UserActivitySerde(),  
        new SearchSerde()));
```

- ➊ First, we create a streams objects for the two streams we want to join—clicks and searches. When we create the stream object, we pass the input topic and the key and value Serde that will be used when consuming records out of the topic and deserializing them into input objects.
- ➋ We also define a KTable for the user profiles. A KTable is a materialized store that is updated through a stream of changes.
- ➌ Then we enrich the stream of clicks with user profile information by joining the stream of events with the profile table. In a stream-table join, each event in the stream receives information from the cached copy of the profile table. We are doing a left-join, so clicks without a known user will be preserved.
- ➍ This is the `join` method—it takes two values, one from the stream and one from the record, and returns a third value. Unlike in databases, we get to decide how to combine the two values into one result. In this case, we created one `activity` object that contains both the user details and the page viewed.
- ➎ Next, we want to `join` the click information with searches performed by the same user. This is still a left `join`, but now we are joining two streams, not streaming to a table.
- ➏ This is the `join` method—we simply add the search terms to all the matching page views.
- ➐ This is the interesting part—a *stream-to-stream join* is a join with a time window. Joining all clicks and searches for each user doesn’t make much sense—we want to join each search with clicks that are related to it, that is, clicks that occurred a short period of time after the search. So we define a join window of one second. We invoke `of` to create a window of one second before and after each search, and then we call `before` with a zero-seconds interval to make sure we only join clicks that happen one second after each search and not before. The results will include relevant clicks, search terms, and the user profile. This will allow a full analysis of searches and their results.
- ➑ We define the Serde of the join result here. This includes a Serde for the key that both sides of the join have in common and the Serde for both values that will be included in the result of the join. In this case, the key is the user ID, so we use a simple `Integer` Serde.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in “Word Count” on page 402.

This example shows two different join patterns possible in stream processing. One joins a stream with a table to enrich all streaming events with information in the table. This is similar to joining a fact table with a dimension when running queries on a data warehouse. The second example joins two streams based on a time window. This operation is unique to stream processing.

As usual, you can find the complete example, including instructions for running it, on [GitHub](#).

Kafka Streams: Architecture Overview

The examples in the previous section demonstrated how to use the Kafka Streams API to implement a few well-known stream processing design patterns. But to understand better how Kafka’s Streams library actually works and scales, we need to peek under the covers and understand some of the design principles behind the API.

Building a Topology

Every streams application implements and executes one *topology*. Topology (also called DAG, or directed acyclic graph, in other stream processing frameworks) is a set of operations and transitions that every event moves through from input to output. Figure 14-10 shows the topology in “Word Count” on page 402.

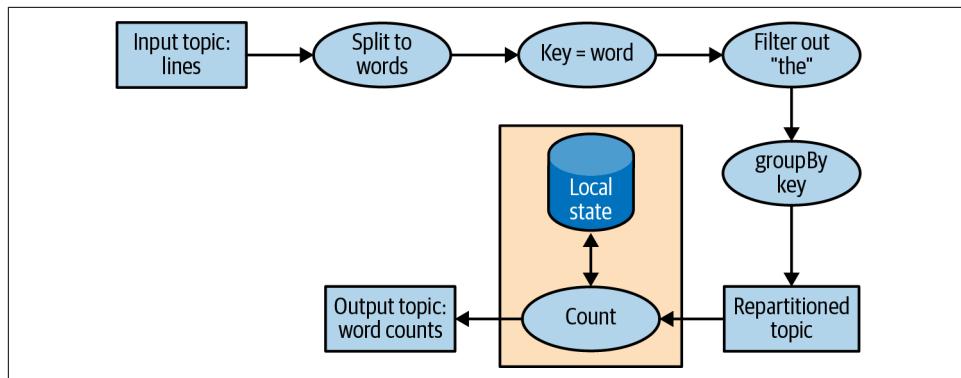


Figure 14-10. Topology for the word-count stream processing example

Even a simple app has a nontrivial topology. The topology is made up of processors—those are the nodes in the topology graph (represented by circles in our diagram). Most processors implement an operation of the data—filter, map, aggregate, etc. There are also source processors, which consume data from a topic and pass it on, and sink processors, which take data from earlier processors and produce it to a

topic. A topology always starts with one or more source processors and finishes with one or more sink processors.

Optimizing a Topology

By default, Kafka Streams executes applications that were built with the DSL API by mapping each DSL method independently to a lower-level equivalent. By evaluating each DSL method independently, opportunities to optimize the overall resulting topology were missed.

However, note that the execution of a Kafka Streams application is a three-step process:

1. The logical topology is defined by creating `KStream` and `KTable` objects and performing DSL operations, such as filter and join, on them.
2. `StreamsBuilder.build()` generates a physical topology from the logical topology.
3. `KafkaStreams.start()` executes the topology—this is where data is consumed, processed, and produced.

The second step, where the physical topology is generated from the logical definitions, is where overall optimizations to the plan can be applied.

Currently, Apache Kafka only contains a few optimizations, mostly around reusing topics where possible. These can be enabled by setting `StreamsConfig.TOPOLOGY_OPTIMIZATION` to `StreamsConfig.OPTIMIZE` and calling `build(props)`. If you only call `build()` without passing the config, optimization is still disabled. It is recommended to test applications with and without optimizations and to compare execution times and volumes of data written to Kafka, and of course, validate that the results are identical in various known scenarios.

Testing a Topology

Generally speaking, we want to test software before using it in scenarios where its successful execution is important. Automated testing is considered the gold standard. Repeatable tests that are evaluated every time a change is made to a software application or library enable fast iterations and easier troubleshooting.

We want to apply the same kind of methodology to our Kafka Streams applications. In addition to automated end-to-end tests that run the stream processing application against a staging environment with generated data, we'll want to also include faster, lighter-weight, and easier-to-debug unit and integration tests.

The main testing tool for Kafka Streams applications is `TopologyTestDriver`. Since its introduction in version 1.1.0, its API has undergone significant improvements, and versions since 2.4 are convenient and easy to use. These tests look like normal unit tests. We define input data, produce it to mock input topics, run the topology with the test driver, read the results from mock output topics, and validate the result by comparing it to expected values.

We recommend using the `TopologyTestDriver` for testing stream processing applications, but since it does not simulate Kafka Streams caching behavior (an optimization not discussed in this book, entirely unrelated to the state store itself, which is simulated by this framework), there are entire classes of errors that it will not detect.

Unit tests are typically complemented by integration tests, and for Kafka Streams, there are two popular integration test frameworks: `EmbeddedKafkaCluster` and `Testcontainers`. The former runs Kafka brokers inside the JVM that runs the tests, while the latter runs Docker containers with Kafka brokers (and many other components, as needed for the tests). `Testcontainers` is recommended, since by using Docker it fully isolates Kafka, its dependencies, and its resource usage from the application we are trying to test.

This is just a short overview of Kafka Streams testing methodologies. We recommend reading the “[Testing Kafka Streams—A Deep Dive](#)” blog post for deeper explanations and detailed code examples of topologies and tests.

Scaling a Topology

Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application. We can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the work involved in data processing.

The Streams engine parallelizes execution of a topology by splitting it into tasks. The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes. Each task is responsible for a subset of the partitions: the task will subscribe to those partitions and consume events from them. For every event it consumes, the task will execute all the processing steps that apply to this partition in order before eventually writing the result to the sink. Those tasks are the basic unit of parallelism in Kafka Streams, because each task can execute independently of others. See [Figure 14-11](#).

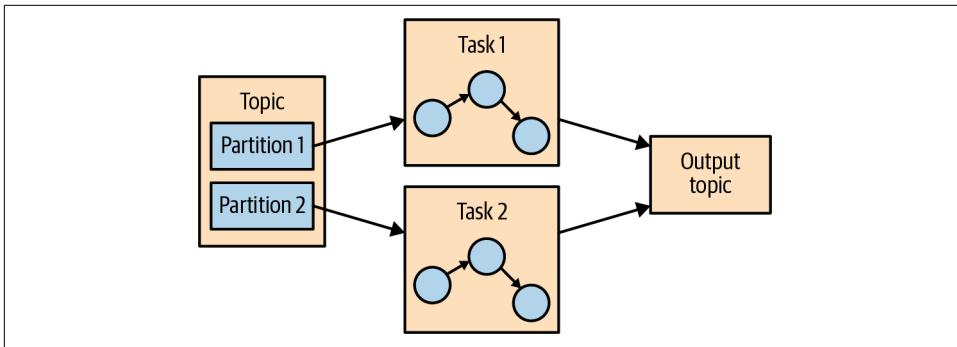


Figure 14-11. Two tasks running the same topology—one for each partition in the input topic

The developer of the application can choose the number of threads each application instance will execute. If multiple threads are available, every thread will execute a subset of the tasks that the application creates. If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server. This is the way streaming applications scale: we will have as many tasks as we have partitions in the topics we are processing. If we want to process faster, add more threads. If we run out of resources on the server, start another instance of the application on another server. Kafka will automatically coordinate work—it will assign each task its own subset of partitions, and each task will independently process events from those partitions and maintain its own local state with relevant aggregates if the topology requires this. See [Figure 14-12](#).

Sometimes a processing step may require input from multiple partitions, which could create dependencies between tasks. For example, if we join two streams, as we did in the ClickStream example in [“ClickStream Enrichment” on page 408](#), we need data from a partition in each stream before we can emit a result. Kafka Streams handles this situation by assigning all the partitions needed for one join to the same task so that the task can consume from all the relevant partitions and perform the join independently. This is why Kafka Streams currently requires that all topics that participate in a join operation have the same number of partitions and be partitioned based on the join key.

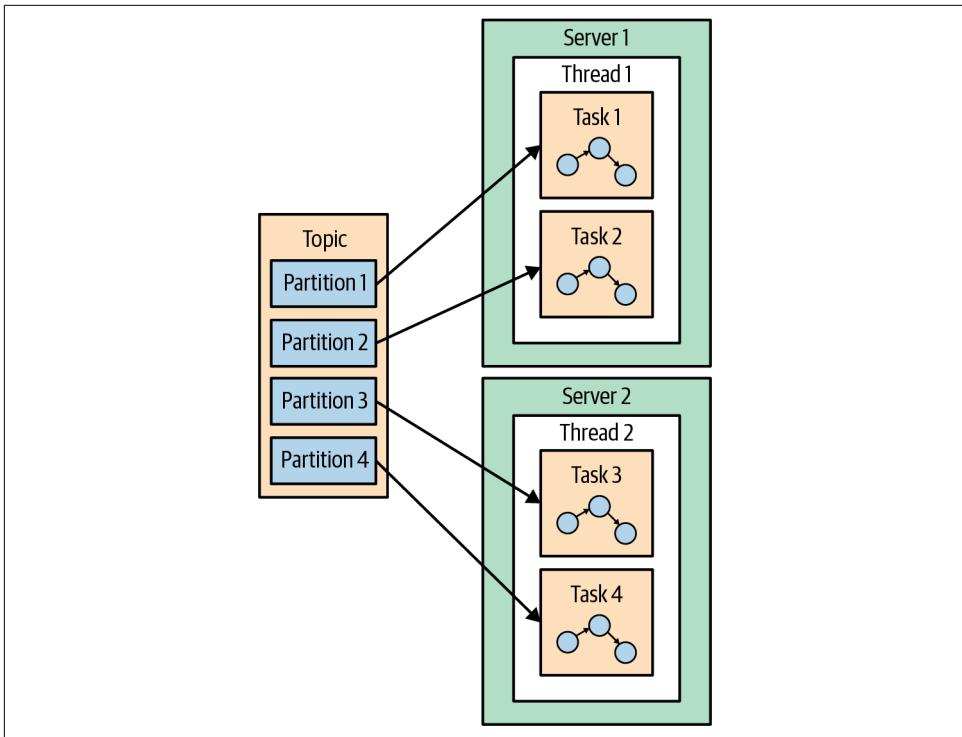


Figure 14-12. The stream processing tasks can run on multiple threads and multiple servers

Another example of dependencies between tasks is when our application requires repartitioning. For instance, in the ClickStream example, all our events are keyed by the user ID. But what if we want to generate statistics per page? Or per zip code? Kafka Streams will repartition the data by zip code and run an aggregation of the data with the new partitions. If task 1 processes the data from partition 1 and reaches a processor that repartitions the data (`groupBy` operation), it will need to *shuffle*, or send events to other tasks. Unlike other stream processor frameworks, Kafka Streams repartitions by writing the events to a new topic with new keys and partitions. Then another set of tasks reads events from the new topic and continues processing. The repartitioning steps break our topology into two subtopologies, each with its own tasks. The second set of tasks depends on the first, because it processes the results of the first subtopology. However, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data into a topic at its own rate and the second set consumes from the topic and processes the events on its own. There is no communication and no shared resources between the tasks, and they don't need to run on the same threads or servers. This is one of the more useful

things Kafka does—reduce dependencies between different parts of a pipeline. See Figure 14-13.

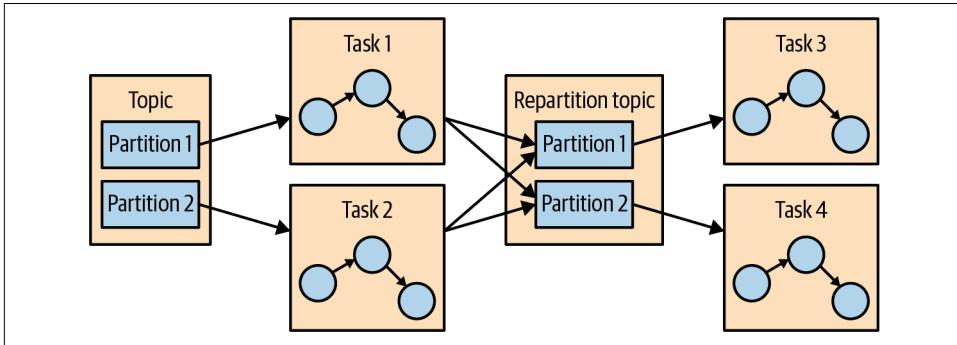


Figure 14-13. Two sets of tasks processing events with a topic for repartitioning events between them

Surviving Failures

The same model that allows us to scale our application also allows us to gracefully handle failures. First, Kafka is highly available, and therefore the data we persist to Kafka is also highly available. So if the application fails and needs to restart, it can look up its last position in the stream from Kafka and continue its processing from the last offset it committed before failing. Note that if the local state store is lost (e.g., because we needed to replace the server it was stored on), the streams application can always re-create it from the change log it stores in Kafka.

Kafka Streams also leverages Kafka's consumer coordination to provide high availability for tasks. If a task failed but there are threads or other instances of the streams application that are active, the task will restart on one of the available threads. This is similar to how consumer groups handle the failure of one of the consumers in the group by assigning partitions to one of the remaining consumers. Kafka Streams benefited from improvements in Kafka's consumer group coordination protocol, such as static group membership and cooperative rebalancing (described in Chapter 4), as well as improvements to Kafka's exactly-once semantics (described in Chapter 8).

While the high-availability methods described here work well in theory, reality introduces some complexity. One important concern is the speed of recovery. When a thread has to start processing a task that used to run on a failed thread, it first needs to recover its saved state—the current aggregation windows, for instance. Often this is done by rereading internal topics from Kafka in order to warm up Kafka Streams state stores. During the time it takes to recover the state of a failed task, the stream processing job will not make progress on that subset of its data, leading to reduced availability and stale data.

Therefore, reducing recovery time often boils down to reducing the time it takes to recover the state. A key technique is to make sure all Kafka Streams topics are configured for aggressive compaction—by setting a low `min.compaction.lag.ms` and configuring the segment size to 100 MB instead of the default 1 GB (recall that the last segment in each partition, the active segment, is not compacted).

For an even faster recovery, we recommend configuring `standby replica`—those are tasks that simply shadow active tasks in a stream processing application and keep the current state warm on a different server. When failover occurs, they already have the most current state and are ready to continue processing with almost no downtime.

More information on both scalability and high availability in Kafka Streams is available in a [a blog post](#) and a [Kafka summit talk on the topic](#).

Stream Processing Use Cases

Throughout this chapter we've learned how to do stream processing—from general concepts and patterns to specific examples in Kafka Streams. At this point it may be worth looking at the common stream processing use cases. As explained in the beginning of the chapter, stream processing—or continuous processing—is useful in cases where we want our events to be processed in quick order rather than wait for hours until the next batch, but also where we are not expecting a response to arrive in milliseconds. This is all true but also very abstract. Let's look at a few real scenarios that can be solved with stream processing:

Customer service

Suppose that we just reserved a room at a large hotel chain, and we expect an email confirmation and receipt. A few minutes after reserving, when the confirmation still hasn't arrived, we call customer service to confirm our reservation. Suppose the customer service desk tells us, "I don't see the order in our system, but the batch job that loads the data from the reservation system to the hotels and the customer service desk only runs once a day, so please call back tomorrow. You should see the email within 2–3 business days." This doesn't sound like very good service, yet we've had this conversation more than once with a large hotel chain. What we really want is for every system in the hotel chain to get an update about a new reservation seconds or minutes after the reservation is made, including the customer service center, the hotel, the system that sends email confirmations, the website, etc. We also want the customer service center to be able to immediately pull up all the details about any of our past visits to any of the hotels in the chain, and the reception desk at the hotel to know that we are a loyal customer so they can give us an upgrade. Building all those systems using stream processing applications allows them to receive and process updates in near real time, which makes for a better customer experience. With such a system, the customer would receive a confirmation email within minutes, their credit card

would be charged on time, the receipt would be sent, and the service desk could immediately answer their questions regarding the reservation.

Internet of Things

IoT can mean many things—from a home device for adjusting temperature and ordering refills of laundry detergent, to real-time quality control of pharmaceutical manufacturing. A very common use case when applying stream processing to sensors and devices is to try to predict when preventive maintenance is needed. This is similar to application monitoring but applied to hardware and is common in many industries, including manufacturing, telecommunications (identifying faulty cellphone towers), cable TV (identifying faulty box-top devices before users complain), and many more. Every case has its own pattern, but the goal is similar: process events arriving from devices at a large scale and identify patterns that signal that a device requires maintenance. These patterns can be dropped packets for a switch, more force required to tighten screws in manufacturing, or users restarting the box more frequently for cable TV.

Fraud detection

Also known as *anomaly detection*, this is a very wide field that focuses on catching “cheaters” or bad actors in the system. Examples of fraud-detection applications include detecting credit card fraud, stock trading fraud, video-game cheaters, and cybersecurity risks. In all these fields, there are large benefits to catching fraud as early as possible, so a near real-time system that is capable of responding to events quickly—perhaps stopping a bad transaction before it is even approved—is much preferred to a batch job that detects fraud three days after the fact, when cleanup is much more complicated. This is, again, a problem of identifying patterns in a large-scale stream of events.

In cybersecurity, there is a method known as *beaconing*. When the hacker plants malware inside the organization, it will occasionally reach outside to receive commands. It can be difficult to detect this activity since it can happen at any time and any frequency. Typically, networks are well defended against external attacks but more vulnerable to someone inside the organization reaching out. By processing the large stream of network connection events and recognizing a pattern of communication as abnormal (for example, detecting that this host typically doesn’t access those specific IPs), the security organization can be alerted early, before more harm is done.

How to Choose a Stream Processing Framework

When choosing a stream processing framework, it is important to consider the type of application you are planning on writing. Different types of applications call for different stream processing solutions:

Ingest

Where the goal is to get data from one system to another, with some modification to the data to conform to the target system.

Low milliseconds actions

Any application that requires almost immediate response. Some fraud-detection use cases fall within this bucket.

Asynchronous microservices

These microservices perform a simple action on behalf of a larger business process, such as updating the inventory of a store. These applications may need to maintain local state caching events as a way to improve performance.

Near real-time data analytics

These streaming applications perform complex aggregations and joins in order to slice and dice the data and generate interesting, business-relevant insights.

The stream processing system you will choose will depend a lot on the problem you are solving:

- If you are trying to solve an ingest problem, you should reconsider whether you want a stream processing system or a simpler ingest-focused system like Kafka Connect. If you are sure you want a stream processing system, you need to make sure it has both a good selection of connectors and high-quality connectors for the systems you are targeting.
- If you are trying to solve a problem that requires low milliseconds actions, you should also reconsider your choice of streams. Request-response patterns are often better suited to this task. If you are sure you want a stream processing system, then you need to opt for one that supports an event-by-event low-latency model rather than one that focuses on microbatches.
- If you are building asynchronous microservices, you need a stream processing system that integrates well with your message bus of choice (Kafka, hopefully), has change capture capabilities that easily deliver upstream changes to the microservice local state, and has the good support of a local store that can serve as a cache or materialized view of the microservice data.
- If you are building a complex analytics engine, you also need a stream processing system with great support for a local store—this time, not for maintenance of local caches and materialized views but rather to support advanced aggregations, windows, and joins that are otherwise difficult to implement. The APIs should include support for custom aggregations, window operations, and multiple join types.

In addition to use case-specific considerations, there are a few global considerations you should take into account:

Operability of the system

Is it easy to deploy to production? Is it easy to monitor and troubleshoot? Is it easy to scale up and down when needed? Does it integrate well with your existing infrastructure? What if there is a mistake and you need to reprocess data?

Usability of APIs and ease of debugging

I've seen orders-of-magnitude differences in the time it takes to write a high-quality application among different versions of the same framework. Development time and time-to-market are important, so you need to choose a system that makes you efficient.

Makes hard things easy

Almost every system will claim they can do advanced windowed aggregations and maintain local stores, but the question is: do they make it easy for you? Do they handle gritty details around scale and recovery, or do they supply leaky abstractions and make you handle most of the mess? The more a system exposes clean APIs and abstractions and handles the gritty details on its own, the more productive developers will be.

Community

Most stream processing applications you consider are going to be open source, and there's no replacement for a vibrant and active community. Good community means you get new and exciting features on a regular basis, the quality is relatively good (no one wants to work on bad software), bugs get fixed quickly, and user questions get answers in a timely manner. It also means that if you get a strange error and Google it, you will find information about it because other people are using this system and seeing the same issues.

Summary

We started the chapter by explaining stream processing. We gave a formal definition and discussed the common attributes of the stream processing paradigm. We also compared it to other programming paradigms.

We then discussed important stream processing concepts. Those concepts were demonstrated with three example applications written with Kafka Streams.

After going over all the details of these example applications, we gave an overview of the Kafka Streams architecture and explained how it works under the covers. We conclude the chapter, and the book, with several examples of stream processing use cases and advice on how to compare different stream processing frameworks.

APPENDIX A

Installing Kafka on Other Operating Systems

Apache Kafka is primarily a Java application and therefore should be able to run on any system where you are able to install a JRE. It has, however, been optimized for Linux-based operating systems, so that is where it will perform best. Running on other operating systems may result in bugs specific to the OS. For this reason, when using Kafka for development or test purposes on a common desktop OS, it is a good idea to consider running in a virtual machine that matches your eventual production environment.

Installing on Windows

As of Microsoft Windows 10, there are now two ways that you can run Kafka. The traditional way is using a native Java installation. Windows 10 users also have the option to use the Windows Subsystem for Linux. The latter method is highly preferred because it provides a much simpler setup that more closely matches the typical production environment, so we will review it first.

Using Windows Subsystem for Linux

If you are running Windows 10, you can install native Ubuntu support under Windows using Windows Subsystem for Linux (WSL). At the time of publication, Microsoft still considers WSL to be an experimental feature. Though it acts similar to a virtual machine, it does not require the resources of a full VM and provides richer integration with the Windows OS.

To install WSL, follow the instructions available from the Microsoft Developer Network at the “[What Is the Windows Subsystem for Linux?](#)” page. Once that is done, you will need to install a JDK using apt (assuming that you have installed the Ubuntu system package for WSL):

```
$ sudo apt install openjdk-16-jre-headless  
[sudo] password for username:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
[...]  
done.  
$
```

Once you have installed the JDK, you can proceed to install Apache Kafka using the instructions in [Chapter 2](#).

Using Native Java

For older versions of Windows, or if you prefer not to use the WSL environment, you can run Kafka natively with a Java environment for Windows. Be aware, however, that this can introduce bugs specific to the Windows environment. These bugs may not get the attention in the Apache Kafka development community as similar problems on Linux might.

Before installing ZooKeeper and Kafka, you must have a Java environment set up. You should install the latest version of Oracle Java 16, which can be found on the [Oracle Java SE download page](#). Download a full JDK package so that you have all the Java tools available, and follow the instructions for installation.



Be Careful with Paths

When installing Java and Kafka, it is highly recommended that you stick to installation paths that do not contain spaces. While Windows allows spaces in paths, applications that are designed to run in Unix environments are not set up this way, and specifying paths will be difficult. When installing Java, make sure to set the installation path with this in mind. For example, if installing JDK 16.0.1, a good choice would be to use the path `C:\Java\jdk-16.0.1`.

Once Java is installed, set up the environment variables so that it can be used. This is done in the Control Panel for Windows, though the exact location will depend on your version of the OS. In Windows 10, you must:

1. Select “System and Security”
2. Select System

3. Select “Advanced system settings,” which will open the System Properties window
4. On the Advanced tab, click the “Environment Variables” button

Use this section to add a new user variable named `JAVA_HOME` ([Figure A-1](#)) and set it to the path where you installed Java. Then edit the system variable named `Path` and add a new entry that is `%JAVA_HOME%\bin`. Save these settings, and exit out of the Control Panel.

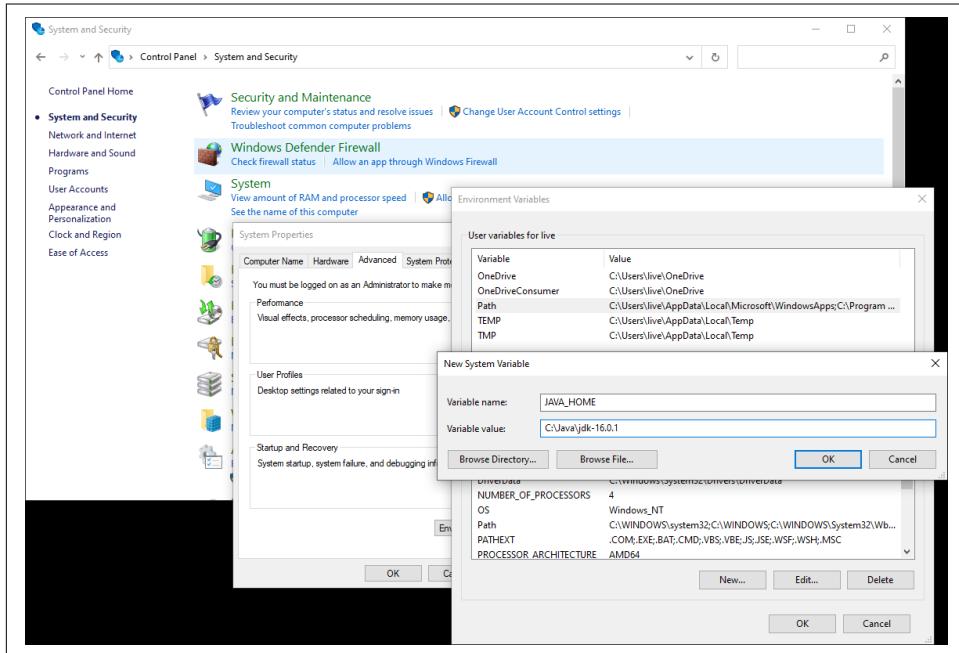


Figure A-1. Adding the `JAVA_HOME` variable

Now you can proceed to install Apache Kafka. The installation includes ZooKeeper, so you do not have to install it separately. The [current release of Kafka](#) can be downloaded online. At publication time, that version is 2.8.0 running under Scala version 2.13.0. The downloaded file will be gzip compressed and packaged with the `tar` utility, so you will need to use a Windows application such as 8 Zip to uncompress it. Similar to installing on Linux, you must choose a directory to extract Kafka into. For this example, we will assume Kafka is extracted into `C:\kafka_2.13-2.8.0`.

Running ZooKeeper and Kafka under Windows is a little different, as you must use the batch files designed for Windows rather than the shell scripts for other platforms. These batch files also do not support backgrounding the application, so you will need a separate shell for each application. First, start ZooKeeper:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> bin\windows\zookeeper-server-start.bat C:
\kafka_2.13-2.8.0\config\zookeeper.properties
[2021-07-18 17:37:12,917] INFO Reading configuration from: C:
\kafka_2.13-2.8.0\config\zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[...]
[2021-07-18 17:37:13,135] INFO PrepRequestProcessor (sid:0) started, reconfigEnabled=false (org.apache.zookeeper.server.PrepRequestProcessor)
[2021-07-18 17:37:13,144] INFO Using checkIntervalMs=60000 maxPerMinute=10000 (org.apache.zookeeper.server.ContainerManager)
```

Once ZooKeeper is running, you can open another window to start Kafka:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> .\bin\windows\kafka-server-start.bat C:
\kafka_2.13-2.8.0\config\server.properties
[2021-07-18 17:39:46,098] INFO Registered kafka:type=kafka.Log4jControllerMBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 17:39:47,918] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
[2021-07-18 17:39:48,009] INFO [broker-0-to-controller-send-thread]: Recorded new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null) (kafka.server.BrokerToControllerRequestThread)
```

Installing on macOS

macOS runs on Darwin, a Unix OS that is derived, in part, from FreeBSD. This means that many of the expectations of running on a Unix OS hold true, and installing applications designed for Unix, like Apache Kafka, is not too difficult. You can either keep the installation simple by using a package manager (like Homebrew), or you can install Java and Kafka manually for greater control over versions.

Using Homebrew

If you have already installed [Homebrew](#) for macOS, you can use it to install Kafka in one step. This will ensure that you have Java installed first, and it will then install Apache Kafka 2.8.0 (as of the time of writing).

If you have not yet installed Homebrew, do that first by following the directions on the [installation page](#). Then you can install Kafka itself. The Homebrew package manager will ensure that you have all the dependencies installed first, including Java:

```
$ brew install kafka
==> Installing dependencies for kafka: openjdk, openssl@1.1 and zookeeper
==> Installing kafka dependency: openjdk
==> Pouring openjdk--16.0.1.big_sur.bottle.tar.gz
[...]
==> Summary
```

```
/usr/local/Cellar/kafka/2.8.0: 200 files, 68.2MB  
$
```

Homebrew will install Kafka under `/usr/local/Cellar`, but the files will be linked into other directories:

- Binaries and scripts will be in `/usr/local/bin`.
- Kafka configurations will be in `/usr/local/etc/kafka`.
- ZooKeeper configurations will be in `/usr/local/etc/zookeeper`.
- The `log.dirs` configuration (the location for Kafka data) will be set to `/usr/local/var/lib/kafka-logs`.

After installation is complete, you can start ZooKeeper and Kafka (this example starts Kafka in the foreground):

```
$ /usr/local/bin/zkServer start  
ZooKeeper JMX enabled by default  
Using config: /usr/local/etc/zookeeper/zoo.cfg  
Starting zookeeper ... STARTED  
$ /usr/local/bin/kafka-server-start /usr/local/etc/kafka/server.properties  
[2021-07-18 17:52:15,688] INFO Registered kafka:type=kafka.Log4jController  
MBean (kafka.utils.Log4jControllerRegistration$)  
[...]  
[2021-07-18 17:52:18,187] INFO [KafkaServer id=0] started (kafka.server.Kafka  
Server)  
[2021-07-18 17:52:18,232] INFO [broker-0-to-controller-send-thread]: Recorded  
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)  
(kafka.server.BrokerToControllerRequestThread)
```

Installing Manually

Similar to a manual installation for the Windows OS, when installing Kafka on macOS, you must first install a JDK. Use the same [Oracle Java SE download page](#) to get the proper version for macOS. You can then download Apache Kafka, similar to Windows again. For this example, we will assume that the Kafka download is expanded into the `/usr/local/kafka_2.13-2.8.0` directory.

Starting ZooKeeper and Kafka looks just like starting them when using Linux, though you will need to make sure your `JAVA_HOME` directory is set first:

```
$ export JAVA_HOME=/usr/libexec/java_home -v 16.0.1  
$ echo $JAVA_HOME  
/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home  
  
$ /usr/local/kafka_2.13-2.8.0/bin/zookeeper-server-start.sh -daemon /usr/local/  
kafka_2.13-2.8.0/config/zookeeper.properties  
$ /usr/local/kafka_2.13-2.8.0/bin/kafka-server-start.sh /usr/local/  
kafka_2.13-2.8.0/config/server.properties  
[2021-07-18 18:02:34,724] INFO Registered kafka:type=kafka.Log4jController
```

```
MBean (kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 18:02:36,873] INFO [KafkaServer id=0] started (kafka.server.Kafka
Server)
[2021-07-18 18:02:36,915] INFO [broker-0-to-controller-send-thread]: Recorded
new controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)((("macOS, installing Kafka on",
startref="ix_macOS")))((("operating systems", "other than Linux, installing
Kafka on", startref="ix_OSinstall")))
```

APPENDIX B

Additional Kafka Tools

The Apache Kafka community has created a robust ecosystem of tools and platforms that make the task of running and using Kafka far easier. While this is by no means an exhaustive list, several of the more popular tools are presented here to help get you started.



Caveat Emptor

While the authors are affiliated with some of the companies and projects that are included in this list, neither they nor O'Reilly specifically endorse one tool over others. Please be sure to do your own research on the suitability of these platforms and tools for the work that you need to do.

Comprehensive Platforms

Several companies offer fully integrated platforms for working with Apache Kafka. This includes managed deployments of all components, such that you can focus on using Kafka and not on how to run it. This can present an ideal solution for use cases where resources are not available (or you do not want to dedicate them) for learning how to properly operate Kafka and the infrastructure required around it. Several also provide tools, such as schema management, REST interfaces, and in some cases client library support, so that you can be assured components interoperate correctly.

Title	Confluent Cloud
URL	https://www.confluent.io/confluent-cloud
Description	It's only fitting that the company created by some of the original developers to develop and support Kafka provides a managed solution. Confluent Cloud combines a number of must-have tools—including schema management, clients, a RESTful interface, and monitoring—into a single offering. It's available on all three major cloud platforms (AWS, Microsoft Azure, and Google Cloud Platform) and is backed with support provided by a sizable portion of the core Apache Kafka contributors employed by Confluent. Many of the components that are included in the platform, such as the Schema Registry and the REST proxy, are available as standalone tools under the Confluent Community License , which does not restrict use cases.
Title	Aiven
URL	https://aiven.io
Description	Aiven provides managed solutions for many data platforms, including Kafka. To support this, it has developed Karapace , which is a schema registry and a REST proxy, both API-compatible with Confluent's components but supported under the Apache 2.0 license , which does not restrict use cases. In addition to the three major cloud providers, Aiven also supports DigitalOcean and UpCloud .
Title	CloudKarafka
URL	https://www.cloudkarafka.com
Description	CloudKarafka focuses on providing a managed Kafka solution with integrations for popular infrastructure services (such as DataDog or Splunk). It supports the use of Confluent's Schema Registry and REST proxy with its platform, but only the 5.0 version prior to the license changes by Confluent. CloudKarafka provides its services on both AWS and Google Cloud Platform.
Title	Amazon Managed Streaming for Apache Kafka (Amazon MSK)
URL	https://aws.amazon.com/msk
Description	Amazon also provides its own managed Kafka platform, supported only on AWS. Schema support is provided through integration with AWS Glue , while a REST proxy is not directly supported. Amazon promotes the use of community tools (such as Cruise Control, Burrow, and Confluent's REST proxy) but does not directly support them. As such, MSK is somewhat less integrated than other offers, but can still provide a core Kafka cluster.
Title	Azure HDInsight
URL	https://azure.microsoft.com/en-us/services/hdinsight
Description	Microsoft also provides a managed platform for Kafka in HDInsight, which also supports Hadoop, Spark, and other big data components. Similar to MSK, HDInsight focuses on the core Kafka cluster, leaving many of the other components (including a schema registry and REST proxy) up to the user to provide. Some third parties have provided templates for performing these deployments, but they are not supported by Microsoft.
Title	Cloudera
URL	https://www.cloudera.com/products/open-source/apache-hadoop/apache-kafka.html
Description	Cloudera has been a fixture in the Kafka community since the early days and provides managed Kafka as the stream data component of its overall Customer Data Platform (CDP) product. CDP focuses on more than just Kafka, however, and it operates in the public cloud environments as well as providing private options.

Cluster Deployment and Management

When running Kafka outside of a managed platform, you will need several things to assist you with running the cluster properly. This includes help with provisioning and deployment, balancing data, and visualizing your clusters.

Title	Strimzi
URL	https://strimzi.io
Description	Strimzi provides Kubernetes operators for deploying Kafka clusters to make it easier to set up Kafka in a Kubernetes environment. It does not provide managed services but instead makes it easy for you to get up and running in a cloud, whether public or private. It also provides the Strimzi Kafka Bridge, which is a REST proxy implementation supported under the Apache 2.0 license . At this time, Strimzi does not have support for a schema registry, due to concerns about licenses.
Title	AKHQ
URL	https://akhq.io
Description	AKHQ is a GUI for managing and interacting with Kafka clusters. It supports configuration management, including users and ACLs, and provides some support for components like the Schema Registry and Kafka Connect as well. It also provides tools for working with data in the cluster as an alternative to the console tools.
Title	JulieOps
URL	https://github.com/kafka-ops/julie
Description	JulieOps (formerly Kafka Topology Builder) provides for automated management of topics and ACLs using a GitOps model. More than viewing the state of the current configuration, JulieOps provides a means for declarative configuration and change control of topics, schemas, ACLs, and more over time.
Title	Cruise Control
URL	https://github.com/linkedin/cruise-control
Description	Cruise Control is LinkedIn's answer to how to manage hundreds of clusters with thousands of brokers. This tool began as a solution to automated rebalancing of data in clusters but has evolved to include anomaly detection and administrative operations, such as adding and removing brokers. For anything more than a testing cluster, it is a must-have for any Kafka operator.
Title	Conduktor
URL	https://www.conduktor.io
Description	While not open source, Conduktor is a popular desktop tool for managing and interacting with Kafka clusters. It supports many of the managed platforms (including Confluent, Aiven, and MSK) and many different components (such as Connect, kSQL, and Streams). It also allows you to interact with data in the clusters, as opposed to using the console tools. A free license is provided for development use that works with a single cluster.

Monitoring and Data Exploration

A critical part to running Kafka is to ensure that your cluster and your clients are healthy. Like many applications, Kafka exposes numerous metrics and other telemetry, but making sense of it can be challenging. Many of the larger monitoring platforms (such as [Prometheus](#)) can easily fetch metrics from Kafka brokers and clients. There are also a number of tools available to assist with making sense of all the data.

Title	Xintra Monitor
URL	https://github.com/linkedin/kafka-monitor
Description	Xintra Monitor (formerly Kafka Monitor) was developed by LinkedIn to monitor availability of Kafka clusters and brokers. It does this by using a set of topics to generate synthetic data through the cluster and measuring latency, availability, and completeness. It's a valuable tool for measuring your Kafka deployment's health without requiring direct interaction with your clients.
Title	Burrow
URL	https://github.com/linkedin/burrow
Description	Burrow is another tool originally created by LinkedIn, which provides holistic monitoring of consumer lag within Kafka clusters. It provides a view into the health of the consumers without needing to directly interact with them. Burrow is actively supported by the community and has its own ecosystem of tools to connect it with other components.
Title	Kafka Dashboard
URL	https://www.datadoghq.com/dashboards/kafka-dashboard
Description	For those who use DataDog for monitoring, it provides an excellent Kafka Dashboard to give you a head start on integrating Kafka clusters into your monitoring stack. It is designed to provide a single-pane view of your Kafka cluster, simplifying the view of many metrics.
Title	Streams Explorer
URL	https://github.com/bakdata/streams-explorer
Description	Streams Explorer is a tool for visualizing the flow of data through applications and connectors in a Kubernetes deployment. While it heavily relies on structuring your deployments using either Kafka Streams or Faust through bakdata's tools, it can then provide an easily comprehensible view of those applications and their metrics.
Title	kcat
URL	https://github.com/edenhill/kafkacat
Description	Kcat (formerly kafkacat) is a much-loved alternative to the console producer and consumer that are part of the core Apache Kafka project. It is small, fast, and written in C, so it does not have JVM overhead. It also supports limited views into cluster status by showing metadata output for the cluster.

Client Libraries

The Apache Kafka project provides client libraries for Java applications, but one language is never enough. There are many implementations of the Kafka client out there, with popular languages such as Python, Go, and Ruby having several options. In addition, REST proxies (such as those from Confluent, Strimzi, or Karapace) can cover a variety of use cases. Here are a few client implementations that have stood the test of time.

Title librdkafka

URL <https://github.com/edenhill/librdkafka>

Description librdkafka is a C library implementation of the Kafka client that is regarded as one of the best-performing libraries available. So good, in fact, that Confluent supports clients for Go, Python, and .NET that it created as wrappers around librdkafka. It is licensed simply under the [two-clause BSD license](#), which makes it easy to use in any application.

Title Sarama

URL <https://github.com/Shopify/sarama>

Description Shopify created the Sarama client as a native Golang implementation. It's released under the [MIT license](#).

Title kafka-python

URL <https://github.com/dpkp/kafka-python>

Description kafka-python is another native client implementation, this time in Python. It's released under the [Apache 2.0 license](#).

Stream Processing

While the Apache Kafka project includes Kafka Streams for building applications, it's not the only choice out there for stream processing of data from Kafka.

Title Samza

URL <https://samza.apache.org>

Description Apache Samza is a framework for stream processing that was specifically designed for Kafka. While it predates Kafka Streams, it was developed by many of the same people, and as a result the two share many concepts. However, unlike Kafka Streams, Samza runs on Yarn and provides a full framework for applications to run in.

Title Spark

URL <https://spark.apache.org>

Description Spark is another Apache project oriented toward batch processing of data. It handles streams by considering them to be fast microbatches. This means the latency is a little higher, but fault tolerance is simply handled through reprocessing batches, and Lambda architecture is easy. It also has the benefit of wide community support.

Title	Flink
URL	https://flink.apache.org
Description	Apache Flink is specifically oriented toward stream processing and operates with very low latency. Like Samza, it supports Yarn but also works with Mesos, Kubernetes, or standalone clusters. It also supports Python and R with provided high-level APIs.
Title	Beam
URL	https://beam.apache.org
Description	Apache Beam doesn't provide stream processing directly but instead promotes itself as a unified programming model for both batch and stream processing. It utilizes platforms like Samza, Spark, and Flink as runners for components in an overall processing pipeline.

Index

A

abortTransaction method, 199
acceptor threads, 143
access control, 267
access control lists (see ACLs)
ACID reliability guarantee, 162
acks parameter, 55, 144, 169
 acks=0, 170
 acks=1, 170
 acks=all, 170, 185, 246
 using correct configuration for reliability, 170
AclAuthorizer, 292-295
 enabling, 291
ACLs (access control lists)
 client ACLs, tool for interacting with, 335
 Deny ACLs, 287
 enabled for ZooKeeper authorization, 300
 for ZooKeeper authorization, 301
 group and role ACLs, 296
 managing for resources in large organizations, 297
 for MirrorMaker, 252
 topic migration with MirrorMaker, 250
 in ZooKeeper, 292
active controller, 139
 active controller count metric, 354
active segment, 153
active-active architecture, 238-240
 replication flow, 251
active-standby architecture, 240-245
 replication flow config in MirrorMaker, 249
activity tracking, 12, 14
 Kafka's features for, 15

ad hoc pipelines, 209
AddPartitionsToTxnRequest, 199
admin requests, 143
AdminClient, 113
 advanced admin operations, 127-130
 adding partitions to a topic, 127
 deleting records from a topic, 128
 leader election, 128
 reassigning replicas, 129
 checking cluster metadata, 127
 consumer group management with
 exploring consumer groups, 123
 modifying consumer groups, 125-126
 deleteRecords method, 158
 lifecycle, creating, configuring, and closing, 115-117
 metadata management and topic creation, 148
 overview, 114-115
 additional notes, 115
 asynchronous and eventually consistent API, 114
 flat hierarchy, 115
 Options, 114
 topic management with, 118-121
administering Kafka, 305-337
 authorizing admin operations, 305
 consumer groups, 312-315
 dynamic configuration changes, 315-320
 other tools, 334
 partition management, 326-334
 producing and consuming, 320-326
 topic operations, 305-312
 unsafe operations, 335-337

aggregation and/or joins in stream processing, 187
aggregations, 393
agility in data pipelines, 209
alerting
 metrics for, 341
 on lack of metrics from broker, 343
 under-replicated partitions alerting trap, 349
 using SLOs in, 345
aliases (DNS), 116
all topics bytes in rate, 357
all topics bytes out rate, 358
Allow ACLs, 296
allow.everyone.if.no.acl.found configuration, 295
AlterConfigResult object, 132
alterPartitionReassignments method, 129
Amazon Web Services (AWS)
 cloud services for Kafka, 16
 installing Kafka in, 36
anonymous connections, 270
Apache Avro, 15
Apache Kafka (see Kafka)
Apache Kafka MirrorMaker (see MirrorMaker)
Apache ZooKeeper (see ZooKeeper)
ApiVersionRequest, 149
application health checks, 343
applications
 reprocessing caused by crashes, 187
 reprocessing caused by zombie applications, 188
 validating, 177
 with multiple instances or multiple producers, sending duplicate messages, 185
asl.enabled.mechanisms parameter, 275
asynchronous commits (offsets), 97
 combining with synchronous commits, 99
at-least-once delivery, 205
atomic multipartition writes, 188
atomicity, 188
auditability, 267
auditing, 265, 298
AuthenticateCallbackHandler, 280
authentication, 265, 269-289
 for MirrorMaker, 252
 reauthentication, 286-287
SASL, 208, 275-286
 delegation tokens, 285-286
SASL/GSSAPI, 276-278
SASL/OAUTHBEARER, 283-285
SASL/PLAIN, 278-281
SASL/SCRAM, 281-282
SASL configuration for ZooKeeper, 299
security updates without downtime, 288
SSL, 270
authorization, 208, 265, 291-298
 AclAuthorizer, 292-295
 customizing, 295-297
 principal associated with MirrorMaker, 252
 security considerations, 297
 for ZooKeeper, 301
authorizer.class.name parameter, 291
auto.commit.interval.ms, 91, 95, 174
auto.create.topics.enable configuration, 27
auto.leader.rebalance.enable configuration, 27, 128
auto.leader.rebalance.enable setting
 true default setting, 141
auto.offset.reset parameter, 91, 125, 173
automatic commit (offsets), 95
automation, metrics consumed by, 342
availability, 267
 allowing out of sync replicas to become leaders, 167
 considerations in determining number of replicas per topic, 165
 high availability in Kafka Streams, 415
 mirroring, use for high availability and disaster recovery, 234
Avro
 serializing with, 64
 using Avro deserialization with Kafka consumer, 109
 using Avro records with Kafka, 65-68
AvroConverter, 212
AvroSerializer, 106
AWS (see Amazon Web Services)
Azure
 cloud services for Kafka, 16
 Kafka on, 36

B

bandwidth, limited, in WANs, 235
batch processing, 384
batch.size parameter, 59, 258
batches, 5
beginTransaction method, 199

bootstrap.servers configuration (Connect workers), 212
bootstrap.servers property (consumers), 84, 269
bootstrap.servers property (producers), 50
broker.id parameter, 25, 38
broker.rack configuration, 166
brokers, 8
 broker-consumer communication between clusters, 236
 clients setting upper and lower boundary on data returned, 146
 cluster membership, 135
 configuration, 24-33, 164-169
 general parameters, 25-27
 keeping replicas in sync, 168
 minimum in- sync replicas, 168
 persisting messages to disk, 169
 replicator factor, 165
 topic defaults, 27-33
 unclean leader election, 166
 configuration defaults, overriding, 318
 configuring broker-side buffers, 257
 configuring TLS for, 273
 configuring to authenticate to ZooKeeper using SASL with a JAAS configuration file, 300
 datacenter layout and, 43
 deciding how many for a cluster, 37
 failure of, 183
 health checks, 343
 installing, 23-24
 keeping open file handle to each partition segment, 153
 Kerberos principal in ZooKeeper, 300
 metrics, 346-370
 active controller count, 354
 all topics bytes in rate, 357
 all topics bytes out rate, 358
 all topics messages in, 359
 cluster-level problems, 350
 controller queue size, 355
 diagnosing cluster problems, 347
 host-level problems, 352
 JVM monitoring, 366-367
 leader count, 359
 logging, 369
 offline partitions count, 360
 partition count, 359
 request handler idle ratio, 355
 request metrics, 361
 topic and partition metrics, 364
 under-replicated partitions, 348
moving topics onto, 327
multiple, configuration for a cluster, 38
per-broker consumer metrics, 375
per-broker producer metrics, 372
quotas for messages, 73
registration with KRaft controller quorum, 139
retrieving passwords from third-party store, 301
SASL mechanisms supported, 275
throttling client's requests, 74
transaction coordinator, 198
buffer.memory parameter, 59
Burrow (consumer lag checker), 179, 377

C

CA (certificate authority), 270
 self-signed CA, 271
callbacks
 for asynchronous send by producers, 53
 send method call with, 51
canary, monitoring, 256
certificate authority (CA), 270
Chandy-Lamport snapshots, 198
change data capture and Debezium Project, 220
ChecksumMessageFormatter, 324
chroot path for Kafka cluster, 26
CLI (command-line interface) utilities
 checking Kafka version for, 306
 for making admin changes, 305
clickstream enrichment example (Kafka Streams), 408-410
client APIs, 47
client ID versus consumer groups, 318
client libraries, 431
client.dns.lookup configuration, 116
 DNS name with multiple IP addresses, 117
 use of DNS aliases, 116
client.id parameter, 55, 93
client.rack parameter, 93
clients
 ACLs, tool for, 335
 built-in Kafka client APIs, 47
 configuration defaults, overriding, 317
 configurations (keys) for, 317
 establishing authenticity of, 267

filtering of topics for subscriptions on client side, 85

Java, JMX metrics allowing client-side monitoring, 178

languages Kafka clients are written in, 142

monitoring, 370-376

- consumer metrics, 373-377
- producer metrics, 370-373

most common types of requests, 143

producer and consumer versus Kafka Connect, 210

quotas applied to, 73

retrieving passwords from third-party store, 301

SSL authentication, 271

third-party, 48

throttling of requests by brokers, 74

cloud computing

- cloud migrations, use of cross-cluster mirroring, 234
- cloud services for Kafka, 16
- installing Kafka in, 35

Cluster Linking, 262

clusters, 8

- checking metadata, 127
- cluster discovery, 245
- cluster-level problems, 350
 - exceeding capacity of brokers to serve requests, 351
 - helpers for balancing clusters, 351
 - unbalanced partitions or leadership, 350

configuring, 36-42

- broker configuration, 38
- Linux OS, tuning, 38-42
- number of brokers, 37

copying data from one Kafka cluster to another, 192

data flow in Kafka cluster, 266

diagnosing cluster problems, 347

impact of replica verification on, 334

membership in, 135

monitoring, 378

multicluster architectures, 235-247

multiple, 9

poorly balanced, throttling behavior in, 317

size of, 149

tiered storage, 150

tools for deployment and management of, 429

command-line interface (CLI) utilities

- checking Kafka version for, 306
- for making admin changes, 305

commit log, 13

commitAsync method, 98

- combining with commitSync, 99
- committing a specified offset, 100

commitSync method, 96, 103

committed messages, 162

commitTransaction method, 199

community-developed external secret config providers, 209

compact retention policy, 156

compacted topics, 400

compaction, 156-158

- how it works, 156
- logging information on log compaction threads, 369
- timing of topic compaction, 159

compression

- of encrypted messages, 291

compression.type parameter, 59

compromised users, 287

concurrent mark and sweep garbage collection, 43

ConfigResource, 121-123

Configurable interface, 72

configuration

- dynamic changes in, 315-320
- describing configuration overrides, 319
- overriding broker configuration defaults, 318
- overriding client and user configuration defaults, 317
- overriding topic configuration defaults, 315
- removing configuration overrides, 319
- validating for brokers and clients, 176

configuration management, 121-123

configuration management systems, 354

configuration prefixes in MirrorMaker, 251

Confluent, 16

- cross-datacenter mirroring solutions, 261

Confluent Control Center, 256

Confluent Hub, downloading connectors from, 216

Confluent Replicator, 261

- features supported, 261

Confluent Schema Registry, 65

Avro, Protobuf, and JSON Schema converters, 211
use of topic to store configuration, 118

Confluent Server, 261, 262

Connect API, 204
(see also Kafka Connect)
data model, 227

connection.failed.authentication.delay.ms, 274

connection.url parameter, 221

connections.max.reauth.ms parameter, 286, 287

connector plug-ins, 211

connectors, 211
(see also Kafka Connect)
building your own, 223
deleting, 216
responsibilities, 226
tasks in MirrorMaker, 250

console consumer, 322-326
 consuming offsets topics, 325
 message formatter options, 324
 using consumer configuration options, 323

console producer, 73, 320-322
 line-reader options, 322
 using producer configuration options, 321

ConsoleProducer\$LineMessageReader, 322

consume locally and produce remotely approach, 254

consume quotas, 73

consumer coordination, 415

consumer groups, 7, 78, 312-315
 client ID versus, 318
 deleting, 313
 listing and describing, 312
 fields provided for my-consumer group, 313
 managing, 123-126
 modifying consumer groups, 125-126
 offset management, 314
 exporting offsets, 314
 importing offsets, 314

ZooKeeper-based, 312

__consumer_offsets topic, 188, 242, 250, 325

consumer.assign method, 111

consumer.assignment method, 104

consumer.close method, 105

consumer.partitionsFor method, 111

consumer.poll method, 190

consumer.wakeup method, 105

ConsumerRebalanceListener class, 101-103
 onPartitionsAssigned method, 101
 onPartitionsLost method, 102
 onPartitionsRevoked method, 101
 committing offsets before losing partition ownership, 102

consumers, 6, 77-112
 commits and offsets, 94-101
 committing a specified offset, 100

concepts, 77-84
 consumer groups and partition balance, 80-83
 consumers and consumer groups, 77-80
 static group membership, 83

configuration validation with Verifiable-Consumer, 176

configuring, 88-94

consumer lag metric, 178

consuming records with specific offsets, 104

creating in Kafka, 84

deciding when to use Kafka Connect instead of, 210

defined, 7

deserializers, 106-110
 custom, 107-109
 using Avro deserialization, 109

exiting the poll loop, 105

metrics, 373-376
 consumer coordinator metrics, 375
 fetch manager metrics, 373
 per-broker and per-topic, 375
 quotas, 376

monitoring for MirrorMaker in production, 256

multiple, Kafka handling of, 10

offset migration in MirrorMaker, 250

poll loop, 86-87

in read_committed or read_uncommitted mode, 190

rebalance listeners, 101-103

standalone, using without a group, 110

stopping for import of offsets by consumer group, 315

subscribing to topics, 85

time-sensitivity requirements between producers and, Kafka decoupling, 204

tuning configuration in MirrorMaker, 258

using in a reliable system, 172-176

explicitly committing offsets in consumers, 174-176

important configuration properties for reliable processing, 173

zombie, 188

ZooKeeper and, 45

controller (cluster), 8, 136-139

controller election, testing, 177

diagnosing problems with, 348

moving, 335

new KRaft controller, 137

controller (disk), 353

controller queue size metric, 355

converters, use by Kafka Connect, 211, 212

converter-specific configuration parameters, 212

converters and Connect's data model, 227

cooperative rebalances, 81, 102, 415

CooperativeStickyAssignor, 92

cost

- considerations in determining number of replicas per topic, 166
- higher cost to communicate between clusters, 235

coupling

- decoupling data sources and targets in data pipelines, 209
- decoupling of writing and reading messages in Kafka, 5
- decoupling producers and consumers, 15

CPUs, 35

- capacity, considerations in clusters, 38
- failure of, 353
- usage metrics, 367

CreateTopicsResult object, 114

credentials for authentication, 208

cross-cluster data mirroring, 233-263

- MirrorMaker, 247-259
 - configuring MirrorMaker, 249
 - deploying MirrorMaker in production, 253-256
 - security, 252
 - tuning MirrorMaker, 257-259
- multicluster architectures, 235-247
 - active-active architecture, 238-240
 - active-standby architecture, 240-245
 - hub-and-spoke architecture, 236-238
 - realities of cross-datacenter communication, 235

other solutions, 259-263

- Confluent cross-datacenter mirroring solutions, 261
- LinkedIn Brooklin, 260
- Uber uReplicator, 259

use cases, 234

Cruise Control, 348

CSV files

- exporting offsets to, 314
- importing offsets for consumer group from, 315

customer service, use of stream processing, 416

D

DAGs (see directed acyclic graphs)

data ecosystem, 12-14

data formats, reconciling in data pipelines, 206

data ingestion tools, 229

data integration, putting in context, 204

data integrity, 267

data pipelines, building, 203-231

- alternatives to Kafka Connect, 229-230

considerations, 204-210

- coupling and agility, 209
- data formats, 206
- failure handling, 209
- high and varying throughput, 205
- reliability, 205
- security, 208
- timeliness, 204
- transformations, 207

deciding when to use Kafka Connect versus producer and consumer, 210

Kafka Connect, 211-229

- connector example, file source to file sink, 214-216
- connector example, MySQL to Elasticsearch, 216-222
- running, 211-214
- single message transformations, 223
- understanding how Kafka Connect works, 225-229

data privacy, 267

data transformations (see transformations)

data types, 206

data-driven enterprises, 1

databases

- reading from and writing to Kafka, then to another database, 192

reading from Kafka topic and writing to database, 191

datacenters

- 2.5 DC architecture for stretch clusters, 247
- layout concerns for production environment, 43
- multiple, 9
- realities of cross-datacenter communication, 235

dead letter queues, 225

Debezium Project

- blog post on outbox pattern, 192
- change capture connectors for variety of databases, 220

debugging

- metrics for, 341
- using log4j logs, 298
- using logging information for, 370

decorating messages, 13

decoupling producers and consumers, 203

default.api.timeout.ms parameter, 90

default.replication.factor parameter, 29, 165

DefaultEncoder, 320

DefaultMessageFormatter, 324

- options to use with --property option, 325

delegation tokens, 285-286

- configuring, 286
- security considerations, 286

delete retention policy, 156

delete.and.compact retention policy, 156

delete.topic.enable configuration, 27, 311

deleted events, 158

deleteRecords method, 128

deletion of topics, 120

delivery.timeout.ms parameter, 57, 171

Deny ACLs, 287, 298

deployment

- secure, guarantees by, 267

DES-MD5 encryption algorithm, 277

DescribeClusterOptions object, 114

DescribeTopicResult object, 119

directed acyclic graphs (DAGs), 402, 410

disaster recovery

- planning in active-standby architecture, 241-245
- after the failover, 245
- cluster discovery, 245
- data loss and inconsistencies in unplanned failover, 241

start offset for applications after failover, 242

stretch clusters and, 246

use of cross-cluster data mirroring, 234

disk faults, introducing, 178

disk performance

- using time-based segments, 32

disk space

- partition allocation and, 152

disks

- capacity, 34
- failure of, 353
- Linux, tuning for Kafka clusters, 41
- monitoring usage of disk space and inodes, 368
- throughput, 33

DNS

- client.dns.lookup configuration, 116
- DNS name with multiple IP addresses, 117
- use of DNS aliases, 116

discovery service, 245

- secure DNS service required by Kerberos, 276

DR (see disaster recovery)

DSL API (Kafka Streams), 402

DumpLogSegment tool, 155

duplication

- avoiding duplicates with producer retries, 184
- retrying to send failed messages leading to, 171

durability, 162

- considerations in determining number of replicas per topic, 165
- of messages in a crash, 163

E

eager rebalances, 81

edge clusters, aggregation of data from, 235

Elasticsearch, ingest frameworks for, 229

Elasticsearch, MySQL connector to (example), 216-222

ELT (Extract-Load-Transform), 207

EmbeddedKafkaCluster framework, 412

enable.auto.commit parameter, 91, 173

enable.idempotence parameter, 60, 61, 171

encryption, 265, 289-291

- compression of encrypted messages, 291
- and decryption, using gpg tool, 302

end-to-end, 289
SASL/DIGEST-MD5 use with TLS encryption, 299
SSL, 270
encryption algorithms
 symmetric, 289
 weak, avoiding, 277
end-to-end encryption, 289
end-to-end latency, 56, 165
end-to-end monitoring, 378
EndTransactionRequest, 199
ensembles (ZooKeeper), 21
entity-types of dynamic config changes, 315
environment setup, 19-23
 installing Java, 19
 installing ZooKeeper, 20-23
 operating system, 19
ephemeral nodes, 135
equi-joins, 398, 399
error handling
 additional, configuring for producers, 171
 connector configuration, 225
 idempotent producer, improvements in, 186
errors
 correct handling by producers, 170
 KafkaProducer, 53
 nonretrievable, 171
 retrievable, handling by producers, 171
ErrorsPerSec metric, 182
ETL (Extract-Transform-Load), 207
 building ETL system with Kafka, 208
 GUI-based ETL tools, 229
event time, 391
EventType, 357
eventual consistency, 114
exactly-once delivery, 205
exactly-once semantics, 61, 181-201
 how transactions guarantee, 188-191
 idempotent producers, 182-186
 transaction performance, 200
 transactions, 186-200
 how they work, 198-200
 how to use, 193-196
 problems not solved by, 191-193
 problems solved by, 187
 transactional IDs and fencing, 196-198
 use cases, 187
exactly_once_beta, 193, 392
exceptions
 producer catching, 52
 returned by KafkaProducer, 53
ExecutionException, 119
expected behavior, 178
Ext4 (fourth extended filesystem), 41
Extents File System (XFS), 41
external secret configuration, 208
Extract-Load-Transform (ELT), 207
Extract-Transform-Load (see ETL)
extreme processing, 210

F

failover to DR cluster in Kafka, 241-245
 after the failover, 245
 cluster discovery, 245
 data loss and inconsistencies in unplanned failover, 241
 start offset for applications after failover, 242
failure handling in data pipelines, 209
failures, surviving in Kafka Streams, 415
FencedProducer error, 189
fencing of zombie applications, 196
 (see also zombie applications)
fetch manager metrics, 373
Fetch requests, 143
 about, 145
 from follower replicas to leader, 141
 metrics on, 361
 timing for, 363
fetch session cache, 147
fetch.max.bytes parameter, 89, 259
fetch.max.wait.ms parameter, 88, 259
fetch.min.bytes parameter, 88, 259
FetchMessageConversionsPerSec, 155
file descriptors, 367
file source and file sink connector (example), 214-216
files
 format of stored data files, 153
 management of stored files, 152
FileStream connectors, 216
FileStreamSink class, 216
FileStreamSource class, 214
filesystem, choosing for Linux disk, 41
fire-and-forget messages, 51
Flink, 176
Fluentd, 229
Flume, 229

flush.messages parameter, 169
flush.ms parameter, 169
follower replicas
 about, 140
 reading from, 140
followers, 8
foreign-key joins, 398
formatting messages, 13
fraud detection, 417
Future objects
 returned by AdminClient methods, 114
 not using blocking get call on, 120
 wrapped in Result objects, 119
Future.get method, 53

G

G1GC (Garbage-First garbage collector), 42, 164
 metrics on, 366
garbage collection
 metrics on, 366
 out of sync replicas and, 164
garbage collector options (Java), 42
Generic Security Service Application Program Interface (GSS-API), 276
 (see also SASL/GSSAPI)
GenericRecord object (Avro), 68
Google, cloud services for Kafka, 16
gpg tool, 302
group by aggregations, 395
group coordinator (broker), 82
group-based access control, 296
group.id configuration (Connect workers), 212
group.id property (consumers), 84, 173
group.instance.id parameter, 93
Gson library, 405
GSS-API (Generic Security Service Application Program Interface), 276
 (see also SASL/GSSAPI)
GUI-based ETL tools, 229
gzip compression, 59

H

Hadoop, 14
hard disk drives (HDDs), 33
hardware
 failures of, 352
 selecting for Kafka broker, 33-35
hash partitioning, 70

hash-based message authentication code (HMAC), 285
HashiCorp Vault, 208
HDDs (hard disk drives), 33
headers
 message batch, 154
 messages in mysql.login topic, 225
 record headers, 71, 240
heartbeat.interval.ms parameter, 89
heartbeats sent by consumer group to broker, 82
high availability, 234
high availability in Kafka Streams, 415
high performance (Kafka), 11
historical metrics, 342
HMAC (hash-based message authentication code), 285
hopping windows, 391
hostname verification (server), 271
htpasswd tool, 279
hub-and-spoke architecture, 236-238
humans, metrics consumed by, 342

I

I/O threads (request handler threads), 143, 355
idempotent producers, 61, 182-186
 how they work, 182-184
 broker failure, 183
 producer restart, 183
 how to use, 185
 improvements in version 2.5, 186
 limitations of, 184
-if-not-exists argument (kafka-topics.sh), 307
in-sync replicas, 141, 163
 clients' consumption of messages from, 146
 keeping replicas in sync, 168
 min.insync.replicas broker configuration, 168
incremental rebalances, 81
 (see also cooperative rebalances)
indexes, 155
 validating index file for log segment, 333
ingest frameworks for other datastores, 229
ingestion time, 386
initTransaction method, 198
inodes, 368
installation, 19-46
 configuring Kafka clusters, 36
 environment setup, 19-23

Kafka broker, 23-24
Kafka in the cloud, 35
production concerns, 42-46
selecting hardware, 33-35
integrated platforms for working with Kafka, 427
intelligent platform management interface (IPMI), 353
inter.broker.listener.name parameter, 269
interactive queries, 401
interceptors (producer), 71
Internet of Things, use of stream processing, 417
interval.ms configuration (see auto.com-mit.interval.ms)
IntSerializer, 107
INVALID_CONFIG exception, 171
IPMI (intelligent platform management interface), 353
isolation.level configuration, 190

J

JAAS configuration file, 275, 276, 278
logins and options for ZooKeeper server, 299
Java
client libraries allowing interaction with topics, 320
clients including JMX metrics for monitoring, 178
conversion of Kafka byte arrays to Java objects, 106
garbage collection options, 42
installing, 19
running Kafka natively on Java environment for Windows, 422
TLS features, use to revoke certificates, 274
Java Authentication and Authorization Service (JAAS), 275
Java Management Extensions (see JMX interface)
Java Virtual Machine (see JVM)
JDBC
cloning connector source, 216
configuring JDBC source connector, 218
creating and configuring JDBC connector, 219
driver for MySQL, installing, 217

JDBC connector using JDBC and SQL to scan for new records, 220
JDK (Java Development Kit), 20
JMX interface, 339
finding the JMX port, 340
using JMX agent running directly on Kafka process to access metrics, 340
joins and/or aggregation in stream processing, 187
JscsSchemaConverter, 212
JSON
serializer and deserializer, generated with Gson, 405
support by Kafka, 211
JSON Web Tokens (JWTs), 283
JSONConverter, 212
JVM (Java Virtual Machine)
heap memory for, 34
monitoring, 366-367
garbage collection, 366
Java OS monitoring, 367

K

Kafka, 4-9
benefits of, 10-12
disk-based retention, 11
handling multiple producers and consumers, 10
high performance, 11
platform features, 11
scalability, 11
brokers and clusters, 8
contributing to, 115
multiple clusters, 9
origins of, 14-17
producers and consumers, 6
topics and partitions, 5
value provided to data pipelines, 203 and ZooKeeper, 20
Kafka Connect, 11, 203, 211-229
agnostic to data formats, 206
alternatives to, 229-230
connector example, file source and file sink, 214-216
connector example, MySQL to Elastic-search, 216-222
deeper look at, 225-229
connectors and tasks, 226

- converters and Connect's data model, 227
- offset management, 228
- worker processes, 227
- error handling and dead letter queues, 225
- MirrorMaker based on, deployment modes, 253
- producer and consumer metrics, 256
- versus producer and consumer, when to use, 210
- running, 211-214
 - configurations for Connect workers, 211
 - standalone mode, 214
- single message transformations (SMTs), 207, 223
- use of topic to store configuration, 118
- Kafka Console consumer, 215
- Kafka internals, 135-159
 - cluster membership, 135
 - controller, 136-139
 - physical storage, 149-159
 - compaction, 156-158
 - deleted events, 158
 - file format, 153
 - file management, 152
 - indexes, 155
 - partition allocation, 151
 - tiered storage, 149
 - timing of topic compaction, 159
 - replication, 139-142
 - request processing, 142-149
- Kafka Streams, 176, 381
 - APIs for manipulating timestamps of records, 387
 - architecture overview, 410-416
 - building a topology, 410
 - optimizing a topology, 411
 - scaling a topology, 412
 - surviving failures, 415
 - testing a topology, 411
 - assignment of time to each event, 387
 - enabling exactly-once semantics using transactions, 193
 - examples, 402-410
 - clickstream enrichment, 408-410
 - stock market statistics, 405-408
 - word count, 402-405
 - support for event time independent of the processing time, 400
- Kafka tools, additional, 427-432
 - client libraries, 431
 - cluster deployment and management, 429
 - comprehensive platforms, 427
 - monitoring and data exploration, 430
 - stream processing, 431
- Kafka, Franz, 17
- kafka-acls.sh tool, 335
- kafka-assigner tool, 351
- kafka-broker-api-versions.sh tool, 335
- kafka-config-topic, 215
- kafka-config.sh tool, 306
- kafka-configs.sh tool, 315-320
 - add-config retention.ms, 316
 - add-config-file argument, 315
 - alter --entity-type topics --entity-name, 316
 - alter command and --delete-config parameter, 319
 - describe command, 319
 - command to change controller mutation for user and client, 318
 - overriding broker configuration defaults, 318
- kafka-console-consumer.sh tool, 320, 322
 - bootstrap-server, --topic, and --whitelist parameters, 323
 - formatter kafka.tools.ChecksumMessageFormatter, 324
 - using consumer configuration options, 323
- kafka-console-producer.sh tool, 73, 320
 - command line arguments to use with --producer-property, 321
 - producing four messages to a topic, 320
 - using producer configuration options, 321
- kafka-consumer-groups.sh tool, 312-315
 - bootstrap-server and --list parameters, 312
 - delete --group arguments, 313
 - describe and --group parameters, 312
 - export --group and --topic options, 314
 - reset-offsets --group --from-file options, 315
 - reset-offsets and --dry-run options, 314
 - resetting offsets based on range of options, 244
 - using for lag monitoring in MirrorMaker, 255
- kafka-dump-log.sh tool, 332-333

--index-sanity-check option, 333
--print-data-log option, 333
--value-decoder-class option, 333
--verify-index-only option, 333
dumping logs from a sample topic, 332
`kafka-leader-election.sh` tool, 326
 --election-type and --all-topic-partitions options, 326
`kafka-mirror-maker.sh` tool, 335
`kafka-performance-producer` tool, 257
`kafka-preferred-replica-election.sh` tool, 326
`kafka-reassign-partitions.sh` tool, 327, 351
 --cancel option, 332
 --verify option, 330
 changing replication factor, 331
 moving topics onto brokers, 328
 other useful options, 329
 partition reassignment from a file, 329
`kafka-replica-verification.sh` tool, 334
`kafka-topics.sh` command, 305
`kafka-topics.sh` tool, 142
 --alter command
 increasing topic partitions, 310
 --bootstrap-server option, 306
 --delete --topic, 311
 --describe --topic, 308
 --describe commad options to filter output, 308
 --list option, 308
 --exclude-internal, 308
 commands to find topic partitions that may have probems, 309
 creating topics, 306
 --if-not-exists argument, 307
 listing under-replicated partitions, 350
`kafka.authorizer.logger`, 298
`kafka.request.logger`, 298
`KafkaAdminClient`, 114
 (see also `AdminClient`)
 admin operations implemented directly, 115
`KafkaAvroDeserializer`, 110
`KafkaAvroSerializer`, 66
`KafkaConsumer` object, 84
`KafkaFuture` object, 120
`KafkaPrincipal`, 269, 271
`KafkaStreams` object, 404, 407, 410
`KafkaStreams.start` method, 411
`kafka-console-producer.sh` tool, 320
Kerberos authentication, 275, 276
credentials with limited lifetime, 286
for ZooKeeper server, 299
secure DNS service required by, 276
 using SASL/GSSAPI for in ZooKeeper, 299
`kerberos.removeHostFromPrincipal`, 300
`kerberos.removeRealmFromPrincipal`, 300
key management system (KMS), 289
key stores and trust stores, 271
 creating for client and server with self-signed CA, 271
 for SSL on ZooKeeper, 300
key store for brokers using self-signed CA certificate, 272
 updates to avoid failed TLS handshake, 274
key-value pairs
 keys for `ProducerRecords`, 68
 mapping of keys to partitions, 69
 record headers, 71
`key.converter` configuration (Connect workers), 212
`key.deserializer` property, 84
`key.serializer` property, 50
keys, 4
 difficulty of adjusting keyed topics, 310
 for clients, 317
 valid keys for topics, 316
KRaft controller, 137-139
Kreps, Jay, 15
KStream object, 411
KTable object, 409, 411

L

lag
 monitoring, 377
 monitoring consumer lag, 374
 monitoring in MirrorMaker, 255
latency
 end-to-end, 165
 higher, in cross-datacenter communication, 235
 producer and end-to-end latency, 56
 transaction timeout and, 190
leader, 8
 consumer group, 83
leader count metric, 359
leader election, 128
 preferred leader replica, 326
 preferred replica election, 326
 starting on specific partitions or topics, 326

testing with VerifiableProducer, 177
unclean leader election, 166
with new KRaft controller, 139
leader replica, 140
 becoming unavailable, 166
LeaderAndIsr request, 148
LeaderSelector, 140
LEADER_NOT_AVAILABLE error, 171
linger.ms parameter, 59, 153, 258
LinkedIn Brooklin, 260
LinkedIn, Kafka's origin at, 14
Linux
 recommended OS for Kafka, 19
 tuning for Kafka clusters, 38
 disk, 41
 networking, 41
 virtual memory, 39
 tuning the Linux network, 258
listeners
 SSL and SASL_SSL, 288
listeners configuration, 25
ListOffsetsRequest, 148
ListTopicOptions object, 114
load-kafka-config, 215
local state, 393
local storage, 150
log append time, 386
log compacted, 9
log.cleaner.enabled configuration, 157
log.dirs configuration, 26, 149
log.retention.bytes parameter, 30
log.retention.ms parameter, 30
log.roll.ms parameter, 32
log.segment.bytes parameter, 31
log4j logs, 298
logging, 369
 brokers generating comprehensive log4j
 logs for auditing, 298
 dumping log segments, 332-333
 information about log compaction threads, 369
 using Kafka to collect system logs, 13
LoggingMessageFormatter, 324
Logstash, 229

M

macOS, installing Kafka on, 424
 installing manually, 425
 using Homebrew, 424

managing Kafka programmatically, 113-133
 adding partitions to a topic, 127
 AdminClient lifecycle, 115-117
 AdminClient overview, 113-115
 cluster metadata, 127
 configuration management, 121-123
 consumer group management, 123-126
 deleting records from a topic, 128
 leader election, 128
 reassigning replicas, 129
 testing, 131-133
 topic management with AdminClient, 118-121

map/reduce processing, 14

materializing the stream, 389

max.block.ms parameter, 57

max.compaction.lag.ms parameter, 159

max.connections parameter, 319

max.in.flight.requests.per.connection, 60, 61, 185, 258

max.inflight.requests parameter, 182

max.partition.fetch.bytes parameter, 89

max.poll.interval.ms parameter, 87, 90

max.poll.records parameter, 89

max.request.size parameter, 60

MaxFileDescriptorCount, 367

memory, 34
 configuring for compaction threads, 157
 Linux virtual memory, tuning for Kafka, 39

message serialization, 16

message.max.bytes parameter, 32, 60

MessageConversionsTimeMs, 155

MessageReader class, 322

messages, 4
 batch headers in Kafka, 153
 compression, 59
 coordinating size configurations, 33
 format down conversion, 155
 format in stored files, 153
 formatter, specifying for console consumer, 324
 schema for, 5

messages in rate, 359

messaging
 publish/subscribe, 1
 use of Kafka for, 13

metadata
 consumer group, added to transaction metadata, 189

loss of in data pipelines, causing coupling, 209
Metadata request and response, 148
ownership of, 138
problems with, leading to replacement of controller, 138
metadata requests, 144
MetadataFetch API, 139
metrics, 339-343
application health checks, 343
deciding which metrics you need, 341-342
alerting or debugging metrics, 341
consumer of metrics, automation or human, 342
historical metrics, 342
Kafka broker, 346-370
sources of, 339
application metrics from JMX interface, 339
nonapplication metrics, 340
using Kafka to collect, 13

microservices
Kafka transactions and, 192
min.compaction.lag.ms parameter, 159
min.insync.replicas parameter, 32, 168, 246, 318
mirroring, 233
(see also cross-cluster data mirroring)
after the failover, 245
challenge of active-active mirroring, 239

MirrorMaker, 9, 233, 247-259
configuring, 249
deploying in production, 253-256
exactly-once capabilities in version 2.0, 192
lightweight, 335
multicloud replication topology, 251
securing, 252
tuning, 257-259
version 2.0, 247

MockAdminClient class, 131-133
Mockito framework, 132
spy injection and doReturn methods, 132
monitoring, 14, 339-379
client monitoring, 370-376
end-to-end, 378
Kafka as monitoring system, 347
Kafka broker metrics, 346-370
Kafka's features for, 15
lag, 377

metric basics, 339-343
deciding which metrics you need, 341
sources of metrics, 339
of MirrorMaker in production, 254
of reliability in production, 178-179
service-level objectives, 343-346
tools for, 430

Multi-Region Clusters (MRC), 261
features supported, 261

multicloud architectures, 235-247
active-active, 238-240
active-standby, 240-245
hub-and-spoke, 236-238
multicloud replication topology, 251
realities of cross-datacenter communication, 235
stretch clusters, 246

multiphase processing/repartitioning, 395
MySQL data pipeline to Snowflake, 206
MySQL to Elasticsearch connector example, 216-222

N

Narkhede, Neha, 15
network threads, 355
networking, 35
improving network utilization when reassigning replicas, 330
introducing network faults, 178
Linux, tuning for Kafka clusters, 41
monitoring network utilization on brokers, 369
problems with, hardware and configuration issues, 353

NoOpMessageFormatter, 324
NotEnoughReplicasException, 168
notifications, 13
num.partitions parameter, 28
num.recovery.threads.per.data.dir configuration, 26

O

OAuth 2.0, 283
(see also SASL/OAUTHBEARER)
observers, 262
offline partitions count, 360
OffsetCommitRequest, 148
OffsetFetchRequest, 148
offsets, 7

auto.offset.reset parameter, 91
committing for records in a transaction, 199
committing offsets and records in a transaction, 191, 192
consumer offset migration in MirrorMaker, 250
consumers committing, 94-101
 asynchronous commit, 97
 automatic commit, 95
 combining synchronous and asynchronous commits, 99
 committing a specified offset, 100
 committing current offset, 96
consumers using ZooKeeper to commit, 45
consuming records with specific offsets, 104
enable.auto.commit parameter, 91
explicitly committing in consumers, 174-176
index mapping timestamps to, 156
index mapping to segment files and positions in the file, 155
listOffsets method, 128
management by workers for source and sink connectors, 228
managing for consumer group, 314
 exporting offsets, 314
 importing offsets, 314
modifying for consumer groups, 125-126
request types for, 148
retrieving by timestamp, 31
source task storing for source records, 226
start offset for applications after failover, 242
translation, 244
workers committing for source and sink connectors, 227
offsets.retention.minutes parameter, 93
online transaction processing (OLTP), 384
open source project (Kafka), 16
OpenFileDescriptorCount, 367
operating systems
 choosing OS for Kafka, 19
 Java OS monitoring, 367
 network configuration problems as OS issues, 353
 OS metrics, 347, 351
 OS monitoring, 367
 other than Linux, installing Kafka on, 421
operational-level agreement (OLA), 344

Options object argument, AdminClient methods, 114
Oracle JDK, 19
order of messages within a partition, 60, 162, 163
org.apache.kafka.tools package, configuration validation tools, 176
out-of-sequence events, 399
out-of-sync replicas, 141, 163
 becoming leader replica, 167
outbox pattern, 192
ownership of partitions, 7

P

partition count metric, 359
partition.assignment.strategy parameter, 91
PartitionAssignor, 83
Partitioner interface
 partition method, 70
partitioners, 49
partitions, 5, 68-71, 163
 adding for a topic, 127, 310
 allocation of, 151
 assigning to consumers, how it works, 83
 assignment strategies, 91
 assignment to standalone consumer, 110
 checking for a topic, 118
 choosing number of, 28
 clean and dirty portions, 156
 compacting, 157
 segment before and after compaction, 158
 finding topic partitions that may have problems, 309
 implementing custom partitioning strategy, 70
 index mapping offsets to segment files and positions in the file, 155
 Kafka cluster with large number of, filtering of topics for subscriptions, 85
 managing, 326-334
 changing a partition's replicas, 327-332
 dumping log segments, 332-333
 preferred replica election, 326
 replica verification, 334
 metrics
 per-partition metrics, 365
 under-replicated partitions, 366
offline partitions count, 360

order of message within, 60
ownership of, 7
preferred leader replica, 128
rebalance, consumer groups and, 80-83
reducing for a topic, 311
repartitioning in stream processing, 395
replication of, 8
size of, 150
specifying number for new topic, 306
splitting into segments for file management, 152
stream processing step requiring input from multiple, 413
triggering leader election for multiple partitions, 129
unbalanced partitions or leadership, 350
unclean leader election, 129
under-replicated, 348
using default partitioner, 68
partitionsFor method, 57
passwords
 clear-text, avoiding, 281
 in SASL/PLAIN, 278
 protection for, 301
percentiles, 363
performance
 commit frequency and, 174
 hardware and, 33
 impact of SSL, 270
 implications of tiered storage, 150
 transactions, 200
physical storage (see storage)
PID (producer ID), 182
PLAIN (see SASL/PLAIN)
PLAINTEXT transport layer, 268
platform features (Kafka), 11
platforms for working with Kafka, 427
plugin.path configuration (Connect workers), 212
poll loop (consumers), 86
 exiting, 105
 poll (Duration) method, 87
 thread safety, 87
preferences (user)
 applying to receipt of messages, 13
preferred leader election, 128, 326
preferred leaders, 141
 finding, 142
preferred replica election, 348
principal.builder.class parameter, 270
principals
 KafkaPrincipal representing client identity, 270
processing guarantees, 392
processing time, 387
processing with external lookup, stream-table joins, 396
processing with local state, 393
processing.guarantee configuration, 193
Processor API, 402
processor threads (network threads), 143
processors, 410
produce quotas, 73
produce requests, 144, 364
producer ID (PID), 182, 185
ProducerRecord object, 49, 322
 keys for records, 68
producers, 6, 47-75
 configuration validation with VerifiableProducer, 176
 configuring, 54-61
 message delivery time, 56-58
 configuring to be reliable, 169-172
 additional error handling, 171
 retries, 171
 send acknowledgments, 170
console producer, 320-322
constructing, 50-51
 mandatory properties, 50
 methods of sending messages, 51
deciding when to use Kafka Connect instead of, 210
defined, 6
idempotent, 182-186
 how they work, 182-184
 how to use, 185
 improvements in version 2.5, 186
 limitations of, 184
interceptors, 71
metrics, 370-373
 overall producer metrics, 371
 per-broker and per-topic, 372
monitoring for MirrorMaker in production, 256
multiple, Kafka handling of, 10
overview, 48-50
quotas and throttling for messages sent by, 73-75

record headers, 71
sending a message to Kafka, 52-54
 asynchronous send, 53
serializers, 61-68
 custom, 62
 serializing using Avro, 64
time-sensitivity requirements between consumers and, Kafka decoupling, 204
transactional, 189, 195
 tuning configuration in MirrorMaker, 258
production deployment systems, 253
production environment
 configuration, 42-46
 colocating applications on ZooKeeper, 44
 datacenter layout, 43
 garbage collector options, 42
Properties object, 84
Protobuf, 62
ProtobufConverter, 212
publish/subscribe messaging, 1
 beginning of, 2
 individual queue systems, 3
publish/subscribe pattern
 transactions and, 193
publishers, 6
 (see also producers)
purgatory buffer, 145

Q

quotas, 265, 376
 produce, consume, and request, 73
 throttling replication using, 129

R

rack awareness
 --disable-rack-aware argument, 307, 330
 disabling for replica assignment, 307
 partitions and replicas assigned to brokers
 on different racks, 151
rack-level misfortune, guarding against, 166
RackAwareReplicaSelector, 93, 140
Raft-based controller (KRaft), 137
RAID (redundant array of independent disks), 34
RangeAssignor, 91
Rao, Jun, 15
rate metrics, 357
RateUnit, 357

readers, 7
 (see also consumers)
reading data from Kafka, 77
 (see also consumers)
read_committed isolation level, 190
read_uncommitted isolation level, 190
reauthentication, 286-287
rebalance listeners, 101-103
rebalanceListener.onPartitionAssignment method, 87
rebalances (partition), 81
 consumer rebalances, 175
 cooperative, 81
 eager, 81
 consumer reliability and, 174
 triggering, 94
receive.buffer.bytes parameter, 61, 93, 257
record-error-rate metric, 182, 371
record-retry-rate metric, 371
RecordMetadata object, 49, 53
records
 committing records and offsets in a transaction, 191, 192
 deleting from a topic, 128
 headers, 71
records-lag-max metric, 374, 378
recovery point objective (RPO), 241
recovery time objective (RTO), 241
redundant array of independent disks (RAID), 34
regional and central clusters
 in hub-and-spoke architecture, 237
 use of cross-cluster data mirroring, 234
regular expressions
 consumer using to subscribe to multiple topics, 85
 matching all topics to consume from (--) whitelist option, 323
regulatory compliance
 use of cross-cluster data mirroring, 234
reliability, 161-180
 broker configuration, 164-169
 for data pipelines, 205
 guarantees, 162
 producer acks parameter and, 55
 replication, 163
 using consumers in a reliable system, 172-176

using producers in a reliable system, 169-172
validating for a system, 176-179
monitoring reliability, 178-179
monitoring reliability in production, 178
validating applications, 177
validating configuration, 176
remote storage, 150
RemoteLogManager, 150
repartitioning, 395
replica.lag.time.max.ms parameter, 141, 168
replicas
 changing for a partition, 327-332
 canceling replica reassigments, 331
 checking for a topic, 118
 determining right number for topics, 165
 fetching messages from closest replica, 93
 follower, 140
 in sync, 163
 keeping in sync, 168
 leader, 140
 out of sync, 163
 partition, allocating among brokers, 151
 placement of, 166
 reassigning, 129
 specifying number for new topic, 306
 verification of, 334
ReplicaSelector interface, 140
replication, 139-142, 163, 233
 between clusters, 236
 flow for active-standby architecture in MirrorMaker, 249
 improvements in, 164
 in multicluster, hub-and-spoke architecture, 237
 MirrorMaker support for, 248
 multicluster replication topology, 251
 preventing replication cycles in active-active mirroring, 239
 replica capacity per broker, 37
 replication factor for brokers, 165
 synchronous, in stretch clusters, 246
 throttling using quotas, 129
 under-replicated partitions, 348
replication factor, 164
 for brokers, 165
 changing for a partition, 331
replication of partitions, 8
replication.factor parameter, 165
reprocessing events, 400
reprocessing messages
 using an application, 123
request handler threads, 355
request metrics, 361
 Fetch request metrics, 361
 time metrics, 362
request processing, 142-149
 Fetch requests, 145
 other types of requests, 147
 produce requests, 144
request queue, 143
request quotas, 73
request-latency-avg metric, 371
request-response, 384
request.timeout.ms parameter, 58, 90, 117
RequestMetrics type, 182
response queue, 143
REST APIs
 for monitoring Confluent Replicator, 261
rest.home.name configuration (Connect workers), 213
rest.port configuration (Connect workers), 213
Result objects, Future object wrapped in, 114, 119
retention of messages, 8
 (see also storage)
 disk-based retention, 11
 setting for a topic, 316
retryable errors (producers), 53
retries
 async commits, 99
 configuring for producers, 171
 consumer, 175
 producer, 184
retries parameter, 58, 60
retry.backoff.ms parameter, 58
role-based access control, 296
rolling restart, testing, 177
RoundRobinAssignor, 92
RoundRobinPartitioner, 69
RPO (recovery point objective), 241
RTO (recovery time objective), 241

S

Salted Challenge Response Authentication Mechanism (SCRAM), 281
(see also SASL/SCRAM)

SASL (Simple Authentication and Security Layer), 117, 208, 275-276
configuring for ZooKeeper, 299
delegation tokens, 285-286
use of SASL/SCRAM to support authentication with, 286
use for MirrorMaker authentication, 252
sasl.jaas.config parameter, 276
sasl.kerberos.service.name parameter, 277
sasl.login.callback.handler.class parameter, 283
sasl.mechanism parameter, 275
SASL/DIGEST-MD5, 299
SASL/GSSAPI, 276-278
configuring, 276
security considerations, 277
SASL/OAUTHBEARER, 283
configuring, 283
security considerations, 285
SASL/PLAIN, 278-281
configuring, 278
loading credentials from external files, 303
security considerations, 281
SASL/SCRAM, 281-282
configuring, 281
security considerations, 282
SASL_PLAINTEXT transport layer, 268
SASL_SSL
SCRAM used with, 282
use with SASL/GSSAPI, 277
SASL_SSL transport layer, 268
scalability
in Kafka Streams, 416
Kafka's flexible scalability, 11
scaling storage independent of memory and CPUs in cluster, 150
scaling a topology, 412
schemas, 5
Avro compatibility rules for, 65
data sources and sinks, 206
data, in Connect API, 227
key and value converters, 212
loss of schema metadata in data pipelines, 209
schema.registry.url parameter, 110
storing Avro schema in Schema Registry, 65
support by Confluent Replicator, 261
SCRAM (Salted Challenge Response Authentication Mechanism), 281
(see also SASL/SCRAM)

SCRAM-SHA-256, 288
secret management systems, 208
Secure Sockets Layer (see SSL)
security
concerns for data pipelines, 208
Kafka, 265-304
auditing, 298
authentication, 269-289
authorization, 291-298
encryption, 289-291
locking down Kafka, 265-268
security protocols, 268-269
MirrorMaker, 252
securing the platform, 301-303
password protection, 301
ZooKeeper, 299-301
authorization, 301
SASL, 299
SSL, 300
security.inter.broker.protocol parameter, 269
seekToBeginning method, 104
seekToEnd method, 104
self-signed CA
key and trust stores for client and server
authentication, 271
send acknowledgments, 170
(see also acks parameter)
send method
synchronous and asynchronous, 51
synchronous send, 52
time to succeed or fail, 56
using async Producer.send, sending messages faster than they're accepted, 75
send.buffer.bytes parameter, 61, 93, 257
sendOffsetsToTransaction method, 199
sequence IDs, 184, 185
Serde classes, 405, 409
Serdess.String class, 405
serial attached storage (serial ATA), 34
serialization/deserialization
defining serializers for consumers, 84
deserializer for console consumer messages, 325
deserializers for consumers, 106-110
deserializing message by passing in decoder, 333
JSON serializer and deserializer, 405
producer serializers, 61-68
custom serializers, 62

serializing using Avro, 64
using Avro records with Kafka, 65-68
serializer for console consumer and producer, 320
Serializer interface, 50
serializers and deserializers, 289
servers
establishing authenticity of, 267
hostname verification, 271
typical servers and problems, 352
ZooKeeper, configuring in ensemble, 22
ZooKeeper, standalone, 21
service-level agreements (see SLAs)
service-level indicators (see SLIs)
service-level objectives (see SLOs)
session window, 390
session.timeout.ms configuration, 84, 89
shuffling, 414
ShutdownHook, 105
SimpleAuthorizer, 291
single message transformations (SMTs), 207, 223
single-event processing, 392
sink connectors, 206
error.tolerance configuration, 225
sinks, 206
SLAs (service-level agreements), 344
SLIs (service-level indicators)
defined, 343
metrics that make good SLIs, 344
SLOs (service-level objectives), 343
defined, 343
metrics that make good SLIs, 344
service-level definitions, 343
using in alerting, 345
SMART (Self-Monitoring, Analysis, and Reporting Technology) tools, 353
snappy compression, 59
Snowflake, data pipeline between MySQL and, 206
socket.receive.buffer.bytes parameter, 257
socket.send.buffer.bytes parameter, 257
solid-state disks (SSDs), 33
SSL (Secure Sockets Layer), 268, 270
client authentication, 271
configuring for ZooKeeper, 300
configuring TLS, 270
encrypting cross-datacenter traffic, 252
performance impacts of, 270
renegotiation not supported by Kafka, 287
TLS/SSL, 269
ssl.client.auth parameter, 271
ssl.principal.mapping.rules parameter, 271
stale metrics, 343
standalone consumer, 110
standalone mode, Kafka Connect, 214
standalone server (ZooKeeper), 20
standby replica, 416
state, 388
consumer application managing, 125
consumers maintaining, 176
controllers tracking latest state, 139
external, 389
local or internal, 388
processing with local state, 393
stream processing application state, 401
static group membership (consumers), 83
StickyAssignor, 92
stock market statistics example (Kafka Streams), 405-408
storage
disk capacity, 34
disk throughput, 33
durable, of messages, 8
physical, 149-159
compaction, 156-158
deleted events, 158
encryption of data, 289
file format, 153
file management, 152
partition allocation, 151
tiered storage, 149
when topic compaction occurs, 159
stream processing, 6, 381-419
about, 382-385
additional tools for, 431
books about, 382
concepts, 385-392
processing guarantees, 392
state, 388
stream-table duality, 389
time, 386
time windows, 390
topology, 385
defined, 385
design patterns, 392-402
interactive queries, 401

multiphase processing/repartitioning, 395
out-of-sequence events, 399
processing with external lookup, stream-table joins, 396
reprocessing, 400
single-event processing, 392
streaming join, 398
table-table joins, 398
Kafka Streams examples, 402-410
side effects while, 191
use cases, 416-417
using Kafka, 14

stream processing applications
transactions allowing accurate results from, 186

stream processing frameworks, 230
choosing a framework, 417

stream, defined, 6

stream-table joins, 396

stream-to-stream joins, 409

StreamBuilder object, 403

streaming joins, 398

streaming platform features (Kafka), 11

Streams AP (see Kafka Streams)

StreamsBuilder.build method, 411

StreamsConfig.TOPOLOGY_OPTIMIZATION, 411

stretch clusters, 261
2.5 DC (datacenter) architecture, 247

StringDeserializer, 107

subscribers, 7
(see also consumers)

super.users configuration (AclAuthorizer), 295

swap space (Linux), 39

system load, 368

T

table-table joins, 398

tables, 389
converting to/from streams, 389, 407
stream-table joins, 396

tasks, 211
connectors and, 226
for connector associated with MirrorMaker, 250
in Kafka Streams, 412
responsibilities of, 226

stream processing application requiring repartitioning, 414
stream processing, running on multiple threads and servers, 413

tasks.max parameter, 257

TCP buffer size, tuning, 257

TestContainers framework, 412

testing
in Kafka Streams applications, 411
test suite in Apache Kafka source repository, 177
tools for, 335
using MockAdminClient, 131-133
validating applications, 177

third-party clients, 48

thread pools for handling client requests, 355

thread safety (consumer poll loop), 87

Thrift, 62

throttling
of client requests, 74, 376
partition reassessments and, 330
of replication with quotas, 129
uneven behavior in poorly balanced clusters, 317
using max.connections parameters, 319

throughput
considerations in determining number of replicas per topic, 165
high and varying, in data pipelines, 205

tiered storage, 149
documentation in KIP-405, 150
performance improvements from, 150

time in stream processing, 386
being mindful of time zones, 388
event time, 386
log append time, 386
processing time, 387

time metrics for requests, 362

time windows, 390

timeliness in data pipelines, 204

TimeoutException, 75

timeoutMs setting, AdminClient methods, 114

TimestampExtractor interface, 387

timestamps
index mapping to Kafka message offsets, 156

TLS (Transport Layer Security), 268, 289
use with SASL/OAUTHBEARER, 285

ZooKeeper support for, 299

TLS/SSL, 269, 270
 configuring TLS, 270
 security considerations for TLS, 274

tombstones, 158

TopicExistsException, 120

topics, 5, 305-312
 adding partitions to, 127
 compaction of, timing, 159
 configuration and ACL migration, mirroring, 250
 configuration defaults, overriding, 315
 configuration, checking and correcting, 122
 consumers subscribing to, 85
 controlling reliability trade-offs at topic level, 164
 creating with kafka-topics.sh, 306
 creation of, 148
 default configurations, 27-33
 default.replication.factor, 29
 log.retention.bytes, 30
 log.retention.ms, 30
 log.roll.ms, 32
 log.segment.bytes, 31
 message.max.bytes, 32
 min.insync.replicas, 32
 num.partitions, 28
 per-topic overrides, no longer supported, 28
 deleting, 311
 deleting manually, 336
 deleting records from, 128
 describing topic details, 308
 finding potentially problematic topic partitions, 309
 finding topics at the minimum ISR settings, 309
 determining number of replicas for, 165
 good naming practices, 307
 increasing number of partitions for, 310
 topics with keyed messages, 310
 listing all in a cluster, 308
 managing with AdminClient, 113, 118-121
 checking if topic exists and creating it if not, 118
 listing topics in a cluster, 118
 metrics, 364
 per-topic metrics, 364
 mirrored with MirrorMaker, 249
 moving onto new brokers, 327

partitions, 6
per-topic consumer metrics, 375
per-topic producer metrics, 372
reducing partitions for, 311
removing configuration override for retention.ms, 319
removing topics to be deleted, 336
retention period for, 152
retention policies, 156
specifying for console consumer, 323
valid keys for, 316

topology in stream processing, 385
 building a topology, 410
 event processing with local state, 393
 optimizing a topology, 411
 processing topology, DAG, 402
 scaling a topology, 412
 testing a topology, 411

TopologyTestDriver, 412

transaction coordinator, 198

transaction log, 198

transaction.timeout.ms parameter, 190, 199

transactional producers, 189

transactional.id, 189, 195

transactional.id.expiration.ms, 200

transactions, 186-200
 how they guarantee exactly once, 188-191
 how they work, 198-200
 how to use, 193-196
 exactly-once guarantees without using Kafka Streams, 194

mirroring solutions not supporting, 241

performance, 200

problems not solved by, 191-193
 copying data from one Kafka cluster to another, 192
 publish/subscribe pattern, 193
 reading from database, writing to Kafka, then to another database, 192
 reading from Kafka topic and writing to database, 191
 side effects while stream processing, 191

problems solved by, 187
 reprocessing caused by application crashes, 187
 reprocessing caused by zombie applications, 188

transactional IDs and fencing, 196-198

- use by Kafka Streams for exactly-once guarantees, 392
use cases, 187
transformations, 207
available from outside main Kafka code base, 224
single message transformations in Kafka Connect, 223
- Transport Layer Security (see TLS)
- Trogdor test framework, 178
- trgdor.sh testing framework, 335
- trust stores, 271, 273
for brokers, 273
for SSL on ZooKeeper, 300
generating for client using broker's self-signed CA, 272
- tumbling window, 391
- two-phase commit (transactions), 198
- U**
- Uber uReplicator, 259
- unclean leader election, 129, 166
testing, 177
- unclean.leader.election.enable parameter, 166, 319
- under-replicated partitions, 348, 366
alerting trap, 349
- UniformStickyPartitioner, 69
- UNKNOWN_PRODUCER_ID errors, 186, 200
- unsafe operations, 335-337
deleting topics manually, 336
moving cluster controller, 335
removing topics to be deleted, 336
- user activity tracking, 12, 14
- User:ANONYMOUS principal, 270
- users
configuration defaults, overriding, 317
- V**
- value.converter configuration (Connect workers), 212
- value.deserializer property, 84
- value.serializer property (producers), 50
- VerifiableConsumer class, 176
- VerifiableProducer class, 176
- vm.dirty_background_ratio, 40
- vm.max_map_count, 40
- vm.overcommit_memory, 40
- vm.swappiness parameter, 39
- W**
- wakeup method (consumer), 105
- WakeUpException, 105
- windowed joins, 398
- WindowedSerde object, 407
- Windows OS, installing Kafka on, 421-424
using native Java, 422-424
using Windows Subsystem for Linux, 421
- word count example (Kafka Streams), 402-405
topology for, 410
- worker processes, 211, 227
starting and configuring, 211
- writers, 6
(see also producers)
- writing messages to Kafka, 47
(see also producers)
- X**
- XFS (Extents File System), 41
- Xinfra Monitor, 378
- Z**
- zero-copy method, 145
- zombie applications
reprocessing caused by, 188
zombie fencing, 189, 196
- ZooKeeper
broker registration with, 135
 colocating applications on, 44
consumer groups based in, 312
installing, 20-23
standalone server, 20
version 3.5.9, 20
ZooKeeper ensemble, 21
- not recommended for direct use in admin operations, 115
- replacement of ZooKeeper-based controller, 137
- restricted access for security of authorization, 297
- securing, 299-301
authorization, 301
SSL, 300
- zookeeper.connect parameter, 25, 38
- zookeeper.session.timeout.ms parameter, 168

About the Authors

Gwen Shapira is an engineering leader at Confluent, leading the cloud native Kafka team, which focuses on making Kafka more elastic, scalable, and multitenant for Confluent Cloud. She has 15 years of experience working with code and customers to build scalable data architectures. Gwen is a frequent presenter at industry conferences and a PMC member on the Apache Kafka project.

Todd Palino is a principal staff engineer in site reliability at LinkedIn, tackling the challenges of managing capacity and efficiency for the entire platform. Previously, he was responsible for architecture, day-to-day operations, and tools development for Kafka and ZooKeeper at LinkedIn, including the creation of an advanced monitoring and notification system. Todd is the developer of the open source project Burrow, a Kafka consumer monitoring tool, and can be found sharing his experience on SRE at industry conferences and tech talks. Todd has spent over 20 years in the technology industry running infrastructure services, including as a Systems Engineer at Verisign, developing service management automation for DNS, networking, and hardware management, as well as managing hardware and software standards across the company.

Rajini Sivaram is a principal engineer at Confluent, designing and developing cross-cluster replication features for Kafka and security features for Confluent Platform and Confluent Cloud. She is an Apache Kafka Committer and member of the Apache Kafka Program Management Committee. Prior to joining Confluent, she was at Pivotal, building a high-performance reactive API for Kafka based on Project Reactor. Earlier, Rajini was at IBM working on Kafka-as-a-Service for the IBM Bluemix platform. Her experience ranges from parallel and distributed systems to Java virtual machines and messaging systems.

Krit Petty is the site reliability engineering manager for Kafka at LinkedIn. Before becoming manager, he worked as an SRE on the team, expanding and increasing Kafka to overcome the hurdles associated with scaling Kafka to never-before-seen heights, including taking the first steps to moving LinkedIn's large-scale Kafka deployments into Microsoft's Azure cloud. Krit has a master's degree in computer science and previously worked managing Linux systems, and as a software engineer developing software for high-performance computing projects in the oil and gas industry.

Colophon

The animal on the cover of *Kafka: The Definitive Guide* is a blue-winged kookaburra (*Dacelo leachii*). It is part of the Alcedinidae family and can be found in southern New Guinea and the less dry area of northern Australia. They are considered to be river kingfisher birds.

The male kookaburra has a colorful look. The lower wing and tail feathers are blue, hence its name, but tails of females are reddish-brown with black bars. Both sexes have cream-colored undersides with streaks of brown, and white irises in their eyes. Adult kookaburras are smaller than other kingfishers at just 15 to 17 inches in length and, on average, weigh about 260 to 330 grams.

The diet of the blue-winged kookaburra is heavily carnivorous, with prey varying slightly given changing seasons. For example, in the summer months there is a larger abundance of lizards, insects, and frogs that this bird feeds on, but drier months introduce more crayfish, fish, rodents, and even smaller birds into their diet. They're not alone in eating other birds, however, as red goshawks and rufous owls have the blue-winged kookaburra on their menu when in season.

Breeding for the blue-winged kookaburra occurs in the months of September through December. Nests are built in hollows in the high parts of trees. Raising young is a community effort; there is at least one helper bird assisting Mom and Dad. Three to four eggs are laid and incubated for about 26 days. Chicks will fledge around 36 days after hatching—if they survive. Older siblings have been known to kill the younger ones in their aggressive and competitive first week of life. Those who aren't victims of fratricide or other causes of death will be trained by their parents to hunt for 6 to 10 weeks before heading off on their own.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

Color illustration by Karen Montgomery, based on a black and white engraving from *English Cyclopaedia*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.