O'REILLY®

Second Edition

Kafka

The Definitive Guide

Real-Time Data and Stream Processing at Scale

Early Release
RAW & UNEDITED

Compliments of

CONFLUENT

Gwen Shapira, Todd Palino,
Rajini Sivaram & Neha Narkhede

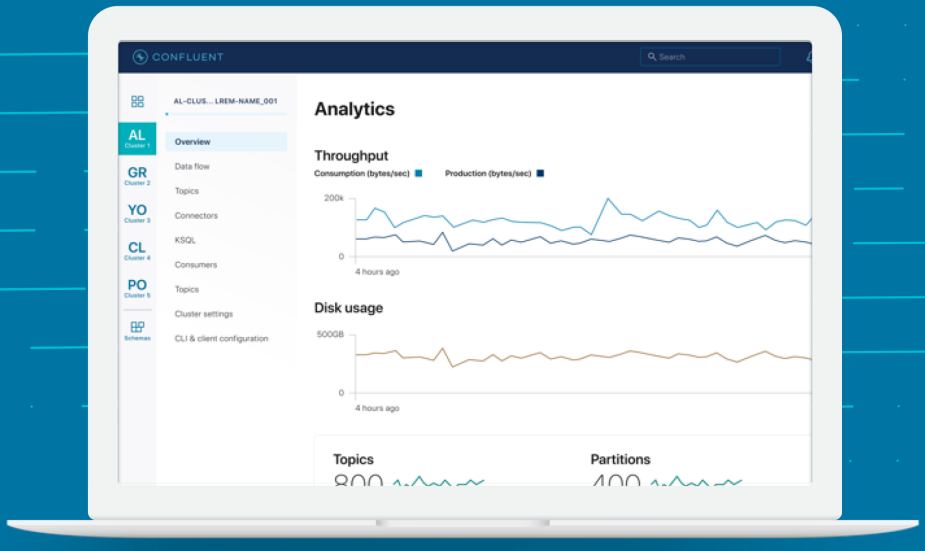
Confluent Cloud

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

aws Google Cloud Microsoft Azure



Try Confluent Cloud Free for 90 Days

New signups get $200 per month for your first 3 months, plus use promo code **KTDG2020** for an additional $200 credit!

TRY FREE

Claim your promo code in-product

Kafka: The Definitive Guide

Real-Time Data and Stream Processing at Scale

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Gwen Shapira, Todd Palino, Rajini Sivaram, and Neha Narkhede

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

Kafka: The Definitive Guide
by Gwen Shapira, Todd Palino, Rajini Sivaram, and Neha Narkhede

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: Jonathan Hassell
Development Editor: Gary O'Brien
Production Editor: Kate Galloway

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2017: First Edition
October 2021: Second Edition

Revision History for the Early Release

2020-05-22: First Release
2020-06-22: Second Release
2020-07-22: Third Release
2020-09-01: Fourth Release

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

978-1-492-04301-0

[LSI]

Table of Contents

CHAPTER 1
Meet Kafka

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *cshapi+ktdg@gmail.com*.

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 important. *Publish/subscribe 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.

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 write in 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.

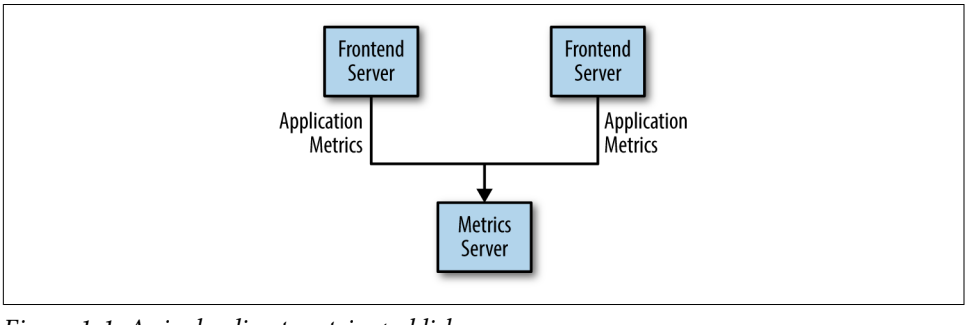


Figure 1-1. A single, direct metrics publisher

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.

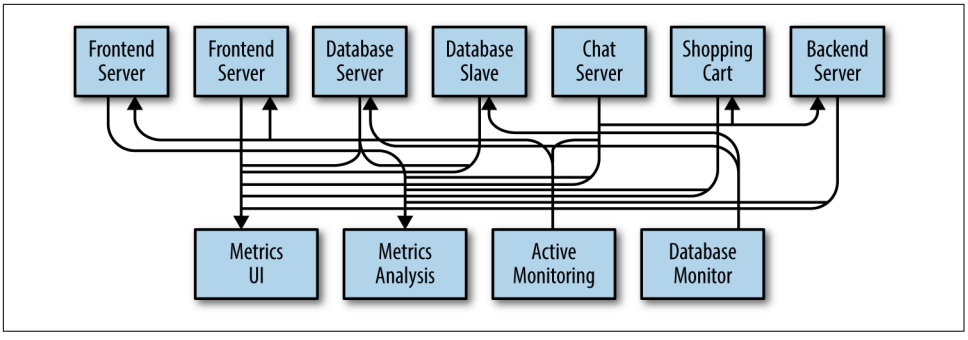


Figure 1-2. Many metrics publishers, using direct connections

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!

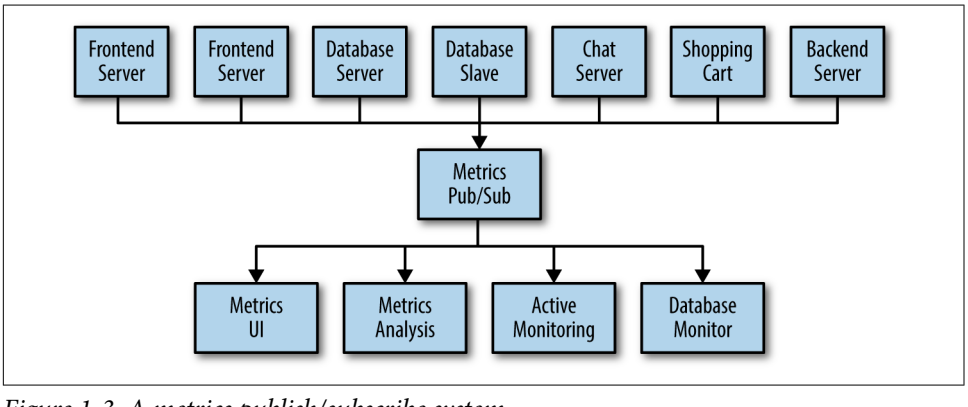


Figure 1-3. A metrics publish/subscribe system

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.

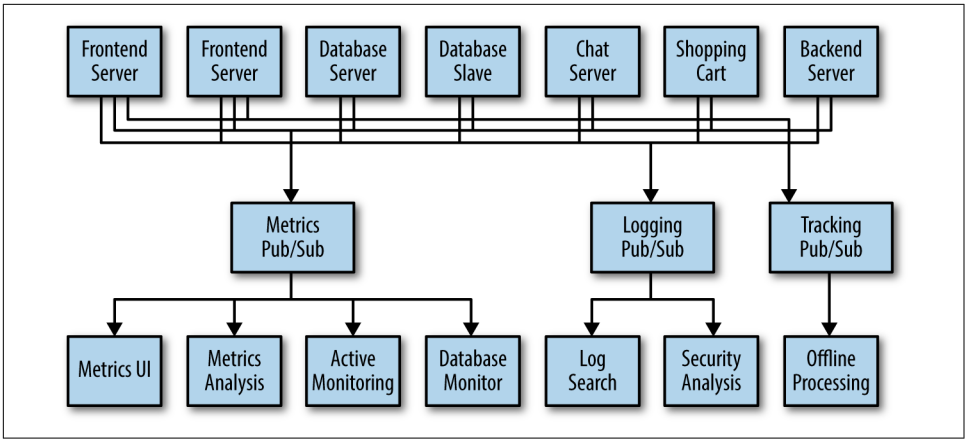


Figure 1-4. Multiple publish/subscribe systems

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 is 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 bit 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 assures that messages with the same key are always written to the same partition.

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 roundtrip 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 tradeoff 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 2.

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 2.

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 time-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.

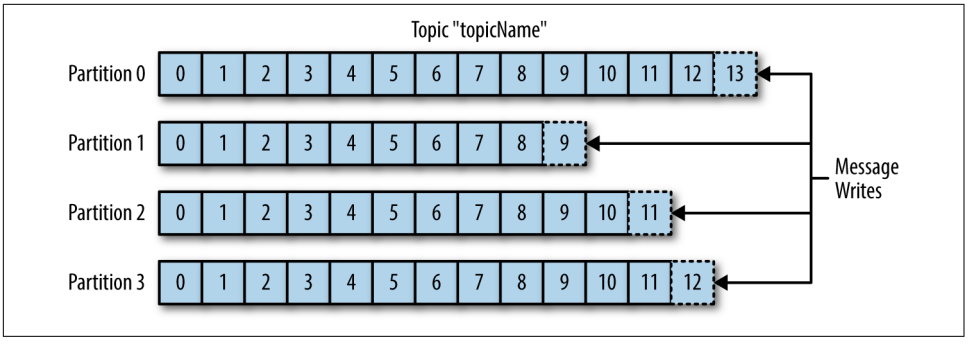


Figure 1-5. Representation of a topic with multiple partitions

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 13.

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*. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and 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 assures 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 2.

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. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The *offset* is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or 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 assures 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 rebalance the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 4.

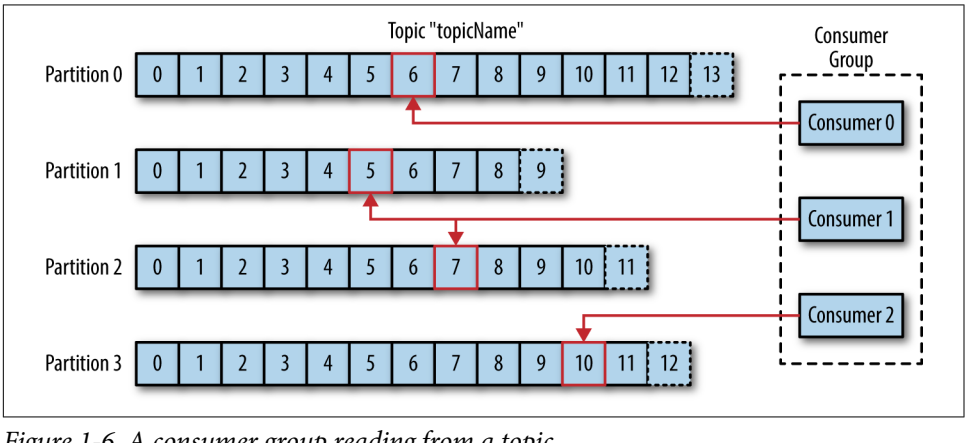


Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the mes-

sages that have been committed to disk. 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 partition may be assigned to multiple brokers, which will result in the partition being replicated (as seen in Figure 1-7). This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. However, all consumers and producers operating on that partition must connect to the leader. Cluster operations, including partition replication, are covered in detail in Chapter 7.

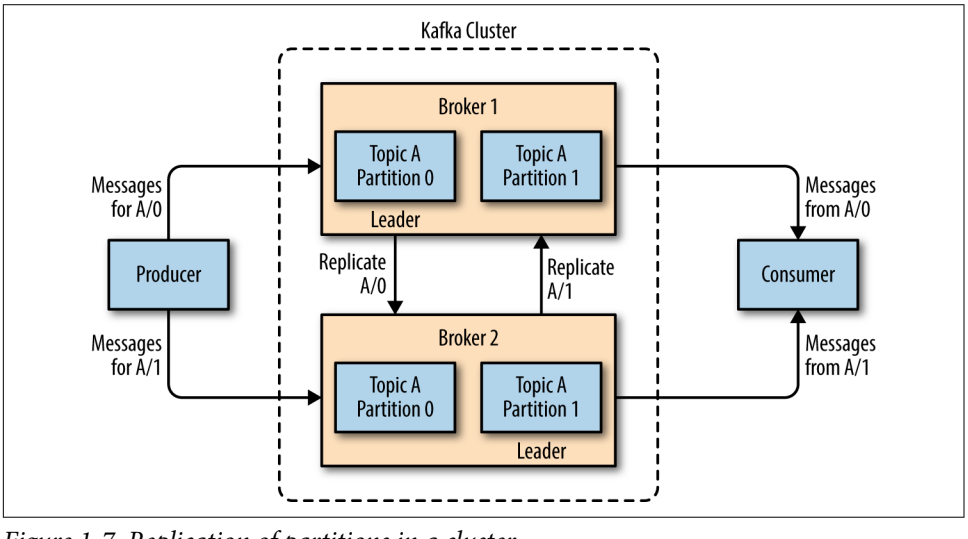


Figure 1-7. Replication of partitions in a cluster

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 topic reaches a certain size in bytes (e.g., 1 GB). Once these limits are reached, messages are expired and deleted so that the retention configuration is 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 mes-

sage 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 this purpose. 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 for 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 8.

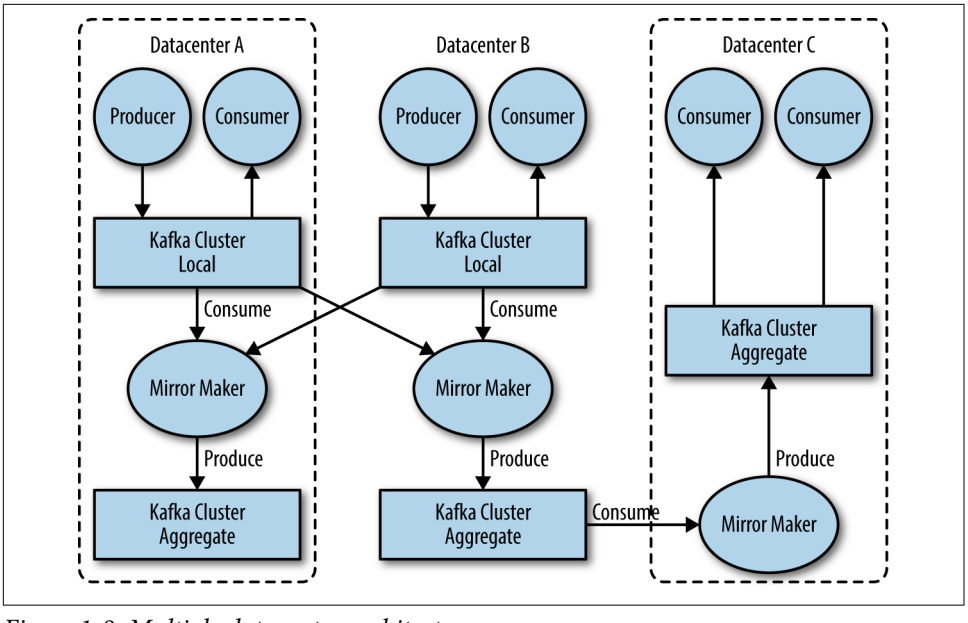


Figure 1-8. Multiple datacenter architecture

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. 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 committed 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.

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.

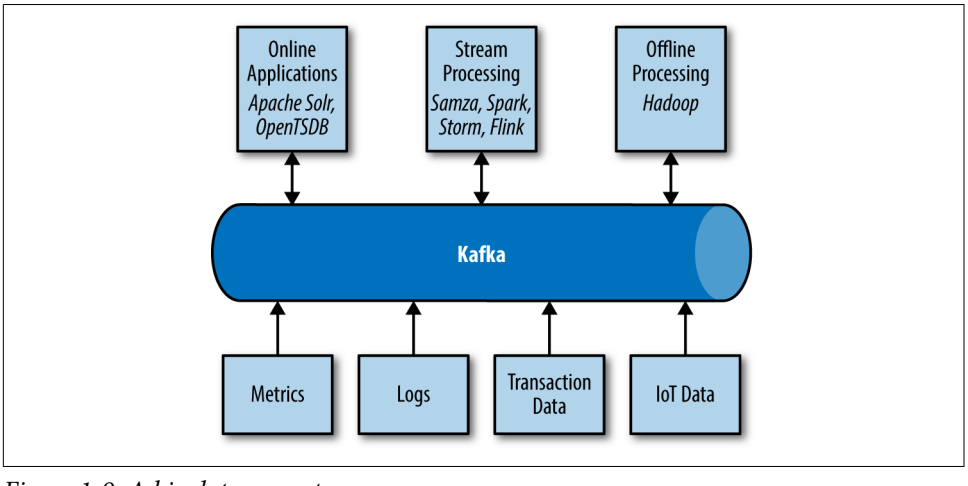


Figure 1-9. A big data ecosystem

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 which 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 trans-

forming messages using data from multiple sources. Stream processing is covered in Chapter 13.

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, *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, 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. This 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 out-

side 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. The largest concentrations of these are from Confluent (including KSQL, as well as their own schema registy and REST projects), and LinkedIn (including Cruise Control, Kafka Monitor, and Burrow).

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 providing 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 in London, Kafka Summit provides a place for the community to come together on a global scale and share knowlege about Apache Kafka and related projects.

The Name

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

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

Getting Started with Kafka

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

CHAPTER 2

Kafka Producers: Writing Messages to Kafka

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *cshapi+ktdg@gmail.com*.

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing 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 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 nor duplicate any messages. Latency should be low but latencies up to 500ms 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 clicked 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 2-1 shows the main steps involved in sending data to Kafka.

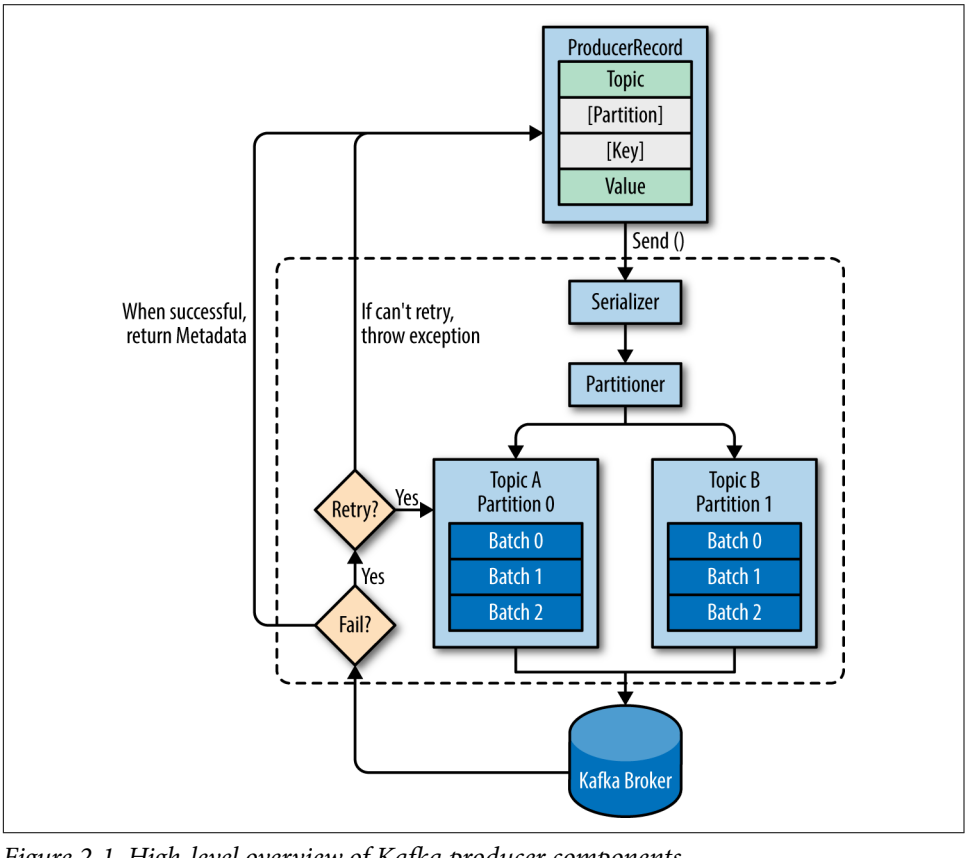


Figure 2-1. High-level overview of Kafka producer components

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 and/or a partition. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to ByteArrays so they can be sent over the network.

Next, the data is sent to a partitioner. If we specified a partition in the `ProducerRecord`, the partitioner doesn't do anything and simply returns the partition we specified. If we didn't, 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`, and `IntegerSerializer`, 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.

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

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

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

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

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

❶ We start with a `Properties` object.

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

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

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

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

Fire-and-forget
: We send a message to the server and don't really care if it arrives succesfully 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 non-retriable errors or timeout, messages will get lost and the application will not get any information or exceptions about this.

Synchronous send
: We send a message, the `send()` method returns a Future object, and we use `get()` to wait on the future and see if the `send()` was successful or not.

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 2-1, 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 a `SerializationException` when it fails to serialize the message, a `BufferExhaustedException` or `TimeoutException` if the buffer is full, or an `InterruptException` if the sending thread was interrupted.

Sending a Message Synchronously

Sending a message synchronously is simple but still allows the producer to catch exceptions when Kafka responds to the produce request with an error, or when send retries were exhausted. The main tradeoff involved is performance. Depending on how busy the Kafka cluster is, brokers can take anywhere from 2ms to 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 not used in production applications (but are very common in code examples).

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

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

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

❷ If there were any errors before sending data to Kafka, while sending, if the Kafka brokers returned a nonretriable error or if we exhausted the available retries, 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 roundtrip time between our application and the Kafka cluster is 10ms. 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.

In order 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 non-null 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. If you want to perform a blocking operation in the callback, it is recommended to use another thread and perform the 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; most 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

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 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. 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. Lets review in detail all three allowed values for the `acks` parameter:

- If `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 went wrong and the broker did 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.
- If `acks=1`, the producer will receive a success response from the broker the moment the leader replica received 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 a replica without this message gets elected as the new leader (via unclean leader election). In this case, throughput depends on whether we send messages synchronously or asynchronously. If our client code waits for a reply from the server (by calling the `get()` method of the `Future` object returned when sending a message) it will obviously increase latency significantly (at least by a network roundtrip). If the client uses callbacks, latency will be hidden, but throughput will be limited by the number of in-flight messages (i.e., how many messages the producer will send before receiving replies from the server).
- If `acks=all`, the producer will receive a success response from the broker once all in-sync replicas received 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 the case of 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 were 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 since the current KafkaProducer was introduced. We will describe here the latest implementation, which was 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 also from the point a `ProduceRecord` was placed in a batch for sending, until Kafka responds with success, non-retriable failure, or we run out of time allocated for sending.

If you use `send()` synchronously, the sending thread will block for both time intervals continously, 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 a diagram:

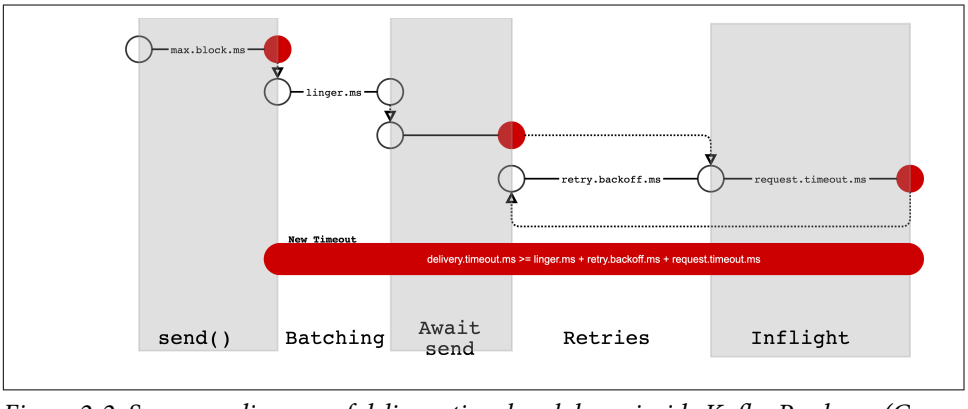


Figure 2-2. Sequence diagram of delivery time breakdown inside Kafka Producer (Contributed to the Apache Kafka project by Sumant Tambe under the ASLv2 license terms)

Below, 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 will block when calling `send()` and when explicitly requesting metadata via `partitionsFor()`. Those methods 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.timout.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 we give up, including time spent on retries. As you can see in Figure 2-2, this time should be greater than `linger.ms` and `request.timeout`. If you try to create a producer with inconsistent timeout configuration, you will get an exception. Messages can be successfully sent much faster than `delivery.timeout.ms` and typically will. This configuration is an upper bound.

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 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 lets keep retrying for 120s just to be on the safe side." Instead of converting this mental dialog to number of retries and time between retries, you just configure `deliver.timeout.ms` to 120s.

request.timeout.ms

This parameter control 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 produce request before giving up - it does not include retries, time spent before sending, etc. 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 100ms between retries, but you can control this using the `retry.backoff.ms` parameter.

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

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

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 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 exception, 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 messages the producer will send to the server without receiving responses. Setting this high can increase memory usage while improving throughput, but setting it too high can reduce throughput as batching becomes less efficient. Setting this to 1 will guarantee that messages will be written to the broker in the order in which they were sent, even when retries occur.

Ordering Guarantees

Apache Kafka preserves the order of messages within a partition. This means that if messages were 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 one means that it is possible that the broker will fail to write the first batch of messages, succeed to write the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.

Usually, setting the number of retries to zero is not an option in a reliable system, so if guaranteeing order is critical, we recommend setting `in.flight.requests.per.session=1` to make sure that while a batch of messages is retrying, additional messages will not be sent (because this has the potential to reverse the correct order). This will severely limit the throughput of the producer, so only use this when order is important.

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 those 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 that you configure your producer to maximize reliability - `acks=all` and decently large `delivery.timeout.ms` to allow sufficient retries. All to 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 first broker crashed before sending a response back 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.

If you wish to avoid this, you can set `enable.idempotence=true`. When 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 within a 5 message window, 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 (either directly or via `delivery.timeout.ms`) 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 and `ByteArrays`, but this does not cover most use cases. Eventually, you will want to be able to serialize more generic records.

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

Custom Serializers

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

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

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

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

    public int getID() {
        return customerID;
    }

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

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

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

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

public class CustomerSerializer implements Serializer<Customer> {

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

    @Override
    /**
    We are serializing Customer as:
```

```
        4 byte int representing customerId
        4 byte int representing length of customerName in UTF-8 bytes (0 if
            name is Null)
        N bytes representing customerName in UTF-8
        */
        public byte[] serialize(String topic, Customer data) {
            try {
                byte[] serializedName;
                int stringSize;
                if (data == null)
                    return null;
                else {
                    if (data.getName() != null) {
                        serializedName = data.getName().getBytes("UTF-8");
                        stringSize = serializedName.length;
                    } else {
                        serializedName = new byte[0];
                        stringSize = 0;
                    }
                }

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

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

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

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

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

Serializing Using Apache Avro

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

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

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka, is that when the application that is writing messages switches to a new 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 fax number 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 that we decide that in the new version, we will upgrade to the twenty-first 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 preupgrade applications that still use the fax numbers and postupgrade 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 than 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, then we recommend checking the documentation.

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 2-3 demonstrates this process.

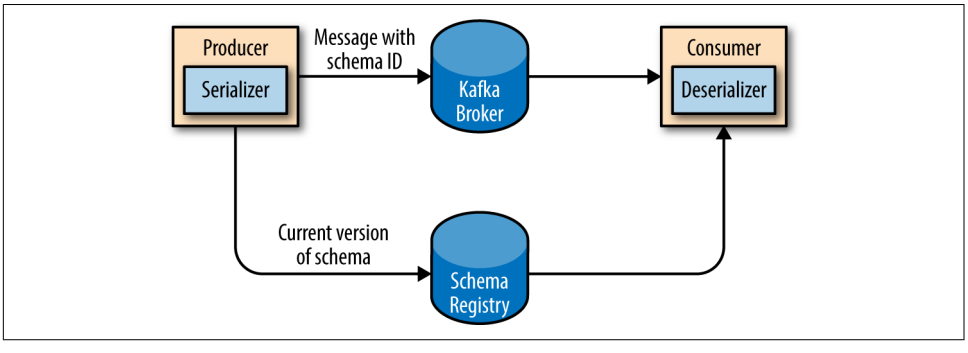


Figure 2-3. Flow diagram of serialization and deserialization of Avro records

Here is an example of how to produce generated Avro objects to Kafka (see the Avro Documentation for how to use code generation with Avro):

```
Properties props = new Properties();

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

String topic = "customerContacts";

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

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

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

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas.

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

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

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

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

What if you prefer to use generic Avro objects rather than the generated Avro objects? No worries. In this case, you just need to provide the schema:

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

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

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

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

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

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

❶ We still use the same `KafkaAvroSerializer`.

❷ And we provide the URI of the same schema registry.

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

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

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

Partitions

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

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

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

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

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

When the key is `null` and the default partitioner is used, the record will be sent to one of the available partitions of the topic at random. A round-robin algorithm will be used to balance the messages among the partitions. Starting in 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 a different random 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 primary key when loading data from Kafka to a relational database), but the workload may be skewed, so a single key may have disproportional 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 will get written to a different partition. When partitioning keys is important, the easiest solution is to create topics with sufficient partitions (Chapter

2 includes suggestions for how to determine a good 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 partitions.

Here is an example of a custom partitioner:

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

public class BananaPartitioner implements Partitioner {

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

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

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

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

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

    public void close() {}
}
```

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

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

Interceptors

There are times, where you want to modify the behavior of 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 acknowledgement for a send. The method does not allow modifying the response from Kafka, but you can capture information about the response.

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

It is tempting to use a producer interceptor to encrypt messages before they are sent to Kafka. However, if you configured compression (highly recommended!), messages will be compressed after they are intercepted. If the interceptor encrypts the messages, the compression step will attempt to compress encrypted messages. Encrypted messages do not compress well, if at all, which makes the compression futile. For a better client-side encryption method, refer to *End-to-End Encryption with Confluent Cloud* by Jason Gustafson. Despite the name, the technique described is generally applicable.

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

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

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

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

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

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

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

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

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

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

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

As we mentioned earlier, producer interceptors can be applied without any changes to the client code. To use the interceptor above with `kafka-console-producer` - an example application that ships with Apache Kafka, follow 3 simple steps:

- Add jar to classpath: `export CLASSPATH=$CLASSPATH:~./target/CountProducerInterceptor-1.0-SNAPSHOT.jar`
- Create a config file that includes:

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

- 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 in which messages are produced and consumed. This is done via the quota mechanism. Kafka has three quota types: produce, consume and request quotas. Produce and Consume quotas limit the rate at which clients can send and receive data, measured in bytes per second. Request quota limit the percentage of time client requests can spend on the request handler and network handler threads.

Quotas can be applied to all clients by setting default quotas, specific client-ids, specific users (as identified by their KafkaPrincipal) 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 megabytes per second 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 producer 4 megabytes a second and clientB 10 megabytes a second, 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 speific clients is through dynamic configuration that can be set using `kafka-config.sh` or the AdminClient API.

Lets look at few examples:

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

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

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

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

❷ Limiting user1 (identified by authenticated principal) to produce only 104 bytes per second and consume only 2048 bytes per second.

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

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

The throttling behavior is exposed to clients via `produce-throttle-time-avg`, `produce-throttle-time-max`, `fetch-throttle-time-avg` and `fetch-throttle-time-max` - the average and the maximum amount of time a produce request and fetch request was delayed due to throttling. Note that this time can represent throttling due to produce and consume throughput quotas, request time quota or both. Other types of client requests can only be throttled due to request time quota, 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 rate of accepting messages, the client will eventually run out of buffer space for storing the excess messages and will block the next `Producer.send()` call. If the timeout delay is insufficient to let the broker catch up to the producer and clear some space in the buffer - eventually `Producer.send()` will throw `TimeoutException`. Alternatively, some of the records that were already placed in batches will wait for longer than `delivery.timeout.ms` and expire, resulting in calling the `send()` callback with a `TimeoutException`. It is therefore important to plan and monitor to make sure that the broker capacity over time will match the rate at which producers are sending data.

Summary

We began this chapter with a simple example of a producer—just 10 lines of code that send events to Kafka. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behavior of the producers. We discussed serializers, which let us control the format of

the events we write to Kafka. We looked in-depth at Avro, one of many ways to serialize events, but one that is very commonly used with Kafka. We concluded the chapter with a discussion of partitioning in Kafka and an example of an advanced custom partitioning technique.

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

CHAPTER 3

Managing Apache Kafka Programmatically

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *cshapi+ktdg@gmail.com*.

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: 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 alternatively, the application can dynamically create a new topic if it receives events with unrecognized device type. The second alternative has downsides but avoiding the dependency on 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 admin client, there were few options, and none of them particularly user-friendly: You could capture `UNKNOWN_TOPIC_OR_PARTITION` exception from the `producer.send()` method and let your user know that they need to create the topic, or you could hope that the Kafka cluster you are writing to enabled automatic topic creation, or you can 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 results of asynchronous operations and they have methods for checking the status of the asynchronous operation, cancelling 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 `CreateTopicsResult` object which lets you wait until all topics are created, lets you check each topic status individually and also lets you 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. It is possible that at that point not every broker is 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, `listTopics` method takes `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 can be things like 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 the admin operations that are 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's 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, non-controversial bug fixes and improvements, before tackling a more significant change to the architecture or the protocol. Non-code 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 a user of the API, 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. The AdminClient API on the other hand, will remain exactly the same, just with a different implementation inside the Kafka cluster.

AdminClient Lifecycle: Creating, Configuring and Closing

In order to use Kafka's AdminClient, the first thing you have to do is construct an instance of the AdminClient class. This is quite straight forward:

```
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 3 brokers, just in case one is currently unavailable. We'll discuss how to configure a secure and authenticated connection separately in the Kafka Security chapter.

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 `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 on-going operations with timeout exception and release all resources. Calling `close` without a timeout implies that you'll wait as long as it takes for all on-going 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 Configurations Kafka documentation. In our opinion, the important configuration parameters are:

client.dns.lookup

This configuration was introduced in Apache Kafka 2.1.0 release.

By default, Kafka validates, resolves and creates connections based on the hostname provided in bootstrap server configuration (and later in the names returned by the

brokers as specified in `advertised.listeners` configuration). This simple model works most of the time, but fails to cover two important use-cases - use of DNS aliases, especially in bootstrap configuration, and use of a single DNS that maps to multiple IP addresses. These sound similar, but are slightly different. Lets look at each of these mutually-exclusive scenarios in a bit more detail.

Use of 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 bootstrap servers configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them. You'll use `all-brokers.hostname.com` for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate `all-brokers.hostname.com`, but the server principal will be `broker2.hostname.com`, if the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

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

DNS name with multiple IP addresses

With modern network architectures, it is common to put all the brokers behind a proxy or a load balancer. This is especially common if you use Kubernetes, where load-balancers are necessary to allow connections from outside the Kubernetes cluster. In these cases, you don't want the load balancers to become a single point of failure. It is therefore very common to make `broker1.hostname.com` point at a list of IPs, all of which resolve to load balancers, and all of them route traffic to the same broker. These IPs are also likely to change over time. By default, KafkaClient 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 the Overview section, 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 lack of timely response from Kafka in a different way. A common example is that services try to validate existence of specific topics when they first start, but if Kafka takes longer than 30s 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 is 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.

Lets start by listing all topics in the cluster:

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

Note that `admin.listTopics()` returns `ListTopicsResult` object which is a thin wrapper over a collection of `Futures`. `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 lets 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. But 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 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 are optional. If there are
        // not specified, the defaults configured on the Kafka brokers will be used
        CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(
                new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❺

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

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

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

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

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

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

❻ Finally, you want to wait for topic creation to return, and perhaps validate the result. In this example, we are checking the number of partitions. Since we specified the number of partitions when we created the topic, we are fairly certain it is correct. Checking the result is more common if you relied on broker defaults when creating the topic. Note that since we are again calling `get()` to check the results of `CreateTopic`, this method could throw an exception. `TopicExistsException` is common in this scenario and you'll want to handle it (perhaps by describing the topic to check for correct configuration).

Now that we have a topic, lets 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 "recyclebin" or "trashcan" 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 un-recoverable 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 usually waiting until the operation succeeds or times out is acceptable. There is one exception - if you are writing a server that is expected to process 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, sending them to Kafka and when Kafka responds, send the response to the client. In these scenarios, the versatility of `KafkaFuture` becomes quite useful. Here's a simple example.

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

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

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

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

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

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

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

❺ If the future completes with an exception, we send the error to the HTTP client

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

The key here is that we are not waiting for 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 long timeout value and one with 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 `Config Resource`. Config resources can be brokers, broker loggers and topics. Checking and modifying broker and broker logging configuration is typically done via 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 their 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 this is not the case.

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 non-default 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 `isDefault()` method that lets us know which configs were modified. A topic configuration is considered non-default if a user configured the topic to have a non-default value, or if a

broker level configuration was modified and the topic that was created inherited this non-default value from the broker.

❸ In order to modify a configuration, you specify a map of the `ConfigResource` you want to modify and a collection of operations. Each configuration modifying operation consists of configuration entry (which is 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. There are four types of operations that modify configuration in Kafka: `SET`, which sets the configuration value, `DELETE` which removes the value and resets to default, `APPEND` and `SUBSTRACT` - those apply only to configurations with `List` type and allows adding and removing values from the list without having to send the entire list to Kafka every time.

Describing configuration can be surprisingly handy in an emergency. I 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 fails to start. The team did not have a way to recover the original, and we prepared for significant trial and error as we attempt 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 their configuration using the AdminClient.

Consumer group management

We've mentioned before that unlike most message queues, Kafka allows you to re-process 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 re-read older messages from a topic. But using these APIs means that you programmed the ability to re-process data in advance into your application. Your application itself must expose the "re-process" functionality.

There are several scenarios in which you'll want to cause an application to re-process 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 would be to list them:

```
admin.listConsumerGroups().valid().get().forEach(System.out::println);
```

Note that by using `valid()` method, the collection that `get()` will return will only contain the consumer groups that the cluster returned without errors, if any. Any errors will be completely ignored, rather than thrown as exceptions. The `errors()` method can be used to get all the exceptions. If you use `all()` as we did in other examples, only the first error the cluster returned will be thrown as an exception. Likely causes of such errors are authorization, where you don't have permission to view the group, or cases when the coordinator for some of the consumer groups is not available.

If we want more information about some of the groups, we can describe them:

```
ConsumerGroupDescription groupDescription = admin
        .describeConsumerGroups(CONSUMER_GRP_LIST)
        .describedGroups().get(CONSUMER_GROUP).get();
        System.out.println("Description of group " + CONSUMER_GROUP
                + ":" + groupDescription);
```

The description contains a wealth of information about the group. This includes the group members, their identifiers and hosts, the partitions assigned to them, the algorithm used for the assignment and the host of the group coordinator. This description is very useful when troubleshooting consumer groups. One of the most important pieces of information about a consumer group is missing from this description - inevitably, we'll want to know what was the last offset committed by the group for each partition that it is consuming, and how much it is lagging behind the latest messages in the log.

In the past, the only way to get this information was to parse the commit messages that the consumer groups wrote to an internal Kafka topic. While this method accomplished its intent, Kafka does not guarantee compatibility of the internal message formats and therefore the old method is not recommended. We'll take a look at how Kafka's AdminClient allows us to retrieve this information.

```
Map<TopicPartition, OffsetAndMetadata> offsets =
        admin.listConsumerGroupOffsets(CONSUMER_GROUP)
                .partitionsToOffsetAndMetadata().get(); ❶

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

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

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

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ❸
    String topic = e.getKey().topic();
    int partition =  e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offset();

    System.out.println("Consumer group " + CONSUMER_GROUP
            + " has committed offset " + committedOffset
            + " to topic " + topic + " partition " + partition
            + ". The latest offset in the partition is "
            +  latestOffset + " so consumer group is "
            + (latestOffset - committedOffset) + " records behind");
}
```

❶ We retrieve a map of all topics and partitions that the consumer group handles, and the latest committed offset for each. Note that unlike `describeConsumerGroups`, `listConsumerGroupOffsets` only accepts a single consumer group and not a collection.

❷ For each one of the topics and partitions 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()`, those 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 who use them 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 code for the consumer, 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".

This is very useful for stateless consumers, but 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 strange impact on the stored state. For example, suppose that you have a streams application that is continuously counting shoes sold in your store, and suppose that at 8:00 am you discover that there was an error in inputs and you want to completely re-calculate the count since 3:00 am. If you reset the offsets to 3:00 am without appropriately modifying the stored aggregate, you will count twice every shoe that was sold today (you will also process all the data between 3:00 am and 8:00 am, but lets assume that this is necessary to correct the error). You need to take care to update the stored state accordingly. In development environment we usually delete the state store completely before resetting the offsets to the start of the input topic.

Also 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.

With all these warnings in mind, lets look at an example:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> earliestOffsets =
admin.listOffsets(requestEarliestOffsets).all().get(); ❶

Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:
        earliestOffsets.entrySet()) {
  resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ❷

try {
  admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ❸
} catch (ExecutionException e) {
  System.out.println("Failed to update the offsets committed by group "
            + CONSUMER_GROUP + " with error " + e.getMessage());
  if (e.getCause() instanceof UnknownMemberIdException)
      System.out.println("Check if consumer group is still active."); ❹
}
```

❶ In order to reset the consumer group so it will start processing from the earliest offset, we need to get the earliest offsets first.

❷ `alterConsumerGroupOffsets` takes as an argument a map with `OffsetAndMetadata` values. But `listOffsets` returns `ListOffsetsResultInfo`, we need to massage the results of the first method a bit, so we can use them as an argument.

❸ We are waiting on the future to complete so we can see if it completed successfully.

❹ One of the most common reasons that `alterConsumerGroupOffsets` will fail is when we didn't stop the consumer group first. 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 in the group is committing an offset for that group. In this case, we'll get `UnknownMemberIdException`.

Cluster Metadata

It is rare that an application has to explicitly discover anything at all about the cluster to which it connected. You can produce and consume messages without ever learning how many brokers exist and which one is the controller. Kafka clients abstract away this information - clients only need to be concerned with topics and partitions.

But just in case you are curious, this little snippet will satisfy your curiosity:

```
DescribeClusterResult cluster = admin.describeCluster();

System.out.println("Connected to cluster " + cluster.clusterId().get()); ❶
System.out.println("The brokers in the cluster are:");
cluster.nodes().get().forEach(node -> System.out.println("    * " + node));
System.out.println("The controller is: " + cluster.controller().get());
```

❶ Cluster identifier is a GUID and therefore is not human readable. It is still useful to check whether your client connected to the correct cluster.

Advanced Admin Operations

In this subsection, we'll discuss 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 each other, 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 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 `createPartitions` method. Note that if you try to expand multiple topics at once, it is possible that some of the topics will be successfully expanded while others will fail.

```
Map<String, NewPartitions> newPartitions = new HashMap<>();
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_PARTITIONS+2)); ❶
admin.createPartitions(newPartitions).all().get();
```

❶ When expanding topics, you need to specify the total number of partitions the topic will have after the partitions are added and not the number of new partitions.

Since `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 it 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 retention policy of 30 days can have older data if all the data fits into a single segment in each partition.

`deleteRecords` method will delete all the records with offsets older than those specified when calling the method. Remember that `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.get-
Value().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 leader, the number of leaders on each broker should be balanced. By default, Kafka will check every 5 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 this option is turned off, or if you want this to happen faster, `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 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. `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 will 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 have 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 away. Maybe you want to add more replicas. Maybe you want to move all replicas away from a broker so you can remove the machine. Or maybe few topics are so noisy that you need to isolate them away 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 when you reassign replicas from one broker to another, it 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 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. So we are going to assign each partition in the topic in a slightly different way:

```
Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new Hash-
Map<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
        Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
        Optional.of(new NewPartitionReassignment(Arrays.asList(0)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹
try {
    admin.alterPartitionReassignments(reassignment).all().get();
} catch (ExecutionException e) {
    if (e.getCause() instanceof NoReassignmentInProgressException) {
        System.out.println("Cancelling a reassignment that didn't exist.");
    }
}
System.out.println("currently reassigning: " +
        admin.listPartitionReassignments().reassignments().get()); ❺
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Description of demo topic:" + topicDescription); ❻
```

❶ We've added another replica to partition 0, placed the new replica on the new broker, but left the leader on the existing broker

❷ We didn't add any replicas to partition 1, simply moved the one existing replica to the new broker. Since I have only one replica, it is also the leader.

❸ We've added another replica to partition 2 and made it the preferred leader. The next preferred leader election will switch leadership to the new replica on the new broker. The existing replica will then become a follower.

❹ There is no on-going reassignment for partition 3, but if there was, this would have cancelled it and returned the state to what it was before the reassignment operation started.

❺ We can list the on-going reassignments

❻ We can also try to print the new state, but remember that it can take a while 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. Some of the 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.

In order to demonstrate how to test using `MockAdminClient`, lets 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 start 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 Hash-
Map<>();
        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'll start testing by instantiating our mock client:

We are using the Mockito testing framework to verify that the `MockAdminClient` methods are called as expected and to fill in for the unimplemented methods. Mockito is a fairly simple mocking framework with nice APIs, which makes it a good fit for a small example of a unit test.

```
@Before
public void setUp() {
    Node broker = new Node(0,"localhost",9092);
    this.admin = spy(new MockAdminClient(Collections.singletonList(broker),
broker)); ❶

    // without this, the tests will throw
    // `java.lang.UnsupportedOperationException: Not implemented yet`
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}
```

❶ `MockAdminClient` is instantiated with a list of brokers (here I'm using just one), and one broker that will be our controller. The brokers are just the broker id, hostname and port - all fake, of course. No brokers will run while executing these tests. We'll use Mockito's `spy` injection, so we can later check that `TopicCreator` executed correctly.

❷ Here we use Mockito's `doReturn` methods to make sure the mock admin client doesn't throw exceptions. Since the method we are testing expects `AlterConfigResult` that returns a `KafkaFuture` when calling its `all()` method, 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 `maybeCreateTopic()` method works properly:

```
@Test
public void testCreateTestTopic()
        throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}

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

❶ The topic name starts with "test", so we expect `maybeCreateTopic()` to create a topic. We are checking that `createTopics()` was called once.

❷ When the topic name doesn't start with "test", we're verifying 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 few other useful methods that are good to have in your back pocket - you never know when you'll need them.

CHAPTER 4
Monitoring Kafka

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *cshapi+ktdg@gmail.com*.

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 whomever is responsible for managing your monitoring system.

This section 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 hardcore 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 plugin 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.

Non-Application Metrics

Not all metrics will come from Kafka itself. There are five general groupings of where you can get your metrics from. When the thing that we are monitoring is the Kafka brokers, the categories are described in Table 4-1.

Table 4-1. Metric Sources

Category	Description
Application Metrics	These metrics are the ones 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 synthic 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 it is 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 to you 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 as to 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 overwhel 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 that includes 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. In order to properly adjust the ratio of air to fuel while the car is running, the computer needs a number of measurements of air densisty, the fuel, the exhaust, and other minutae 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 the level of service they can expect from the infrastructure service is. The clients want to be able to treat services like Kafka as a 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 describe one aspect of a service's reliability. They 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 them as a ratio between the number of good events and the total number of events. For example: the proportion of requests to a webserver 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 timeframe that it is measured over, frequently on the scale of days. For example, 99% of requests to the webserver 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 above 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 a SLA. The goal is to assure that the multiple activities that are necessary to fulfil 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 4-2.

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 4-2. 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	Is 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, bucket aggregations 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, service level objectives 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 chosen to use a long time scale, 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 10ms. 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.

(need a graph here for the burn rate)

On Tuesday at 10 AM, 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 PM, 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 published by O'Reilly Media, 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 assure that you have run a preferred replica election (see Chapter 11) recently. Kafka brokers do not automatically take partition leadership back (unless auto leader rebalance is enabled, but this configuration is not recommended) 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 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 4-3 for more details on under-replicated partitions.

Table 4-3. 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 author has spoken at length 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 non-zero 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 offline. 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 could also be an OS or Java issue that has caused the problem.

If the number of underreplicated 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 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, 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 11), 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 4-4.

Table 4-4. Utilization Metrics

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

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 11.

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. In order 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 underreplicated 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 controlled shutdown of a broker and you will need to force stop the broker instead. See Table 4-5 for more details on active controller count.

Table 4-5. Acive controller count metric

Metric name	Active controller count
JMX MBean	`kafka.controller:type=KafkaController,name=ActiveControllerCount`
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 4-6 for more details on controller queue size.

Table 4-6. Controller queue size metric

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 IO 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 4-7 for more details on the request handler idle ratio.

Table 4-7. Request handler idle ratio

Metric name	Request handler average idle percentage
JMX MBean	`kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent`

Metric name	Request handler average idle percentage
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 requests that will take a long time to process to purgatory. 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 4-8 for more details.

Table 4-8. Details on all topics bytes in metric

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 4-9 for more details.

Table 4-9. Details on all topics bytes out metric

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 maintenance work that is needed. See Table 4-10 for more details.

Table 4-10. Details on all topics messages in metric

Metric name	Messages in per second
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec`

Metric name	Messages in per second
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 4-11 for more details.

Table 4-11. Details on partition count metric

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 4-12 for more details.

Table 4-12. Details on leader count metric

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 4-13). 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 4-13. Offline partitions count metric

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 4-14. Request metrics names

AddOffsetsToTxn	AddPartitionsToTxn	AlterConfigs
AlterPartitionReassignments	AlterReplicaLogDirs	ApiVersions

AddOffsetsToTxn	AddPartitionsToTxn	AlterConfigs
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 8 metrics provided, providing insight into each of the phases of the request processing. For example, for the `Fetch` request, the metrics shown in Table 4-15 are available.

Table 4-15. 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`
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 time, 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
: Measures the total amount of time the broker spends processing the request, from receiving it to sending the response back to the requestor.

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 4-16, 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 4-16. Metrics for Each Topic

Name	JMX MBean
Bytes in rate	`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=`*`TOPIC NAME`*
Bytes out rate	`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=`*`TOPICNAME`*
Failed fetch rate	`kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=`*`TOPICNAME`*
Failed produce rate	`kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=`*`TOPICNAME`*
Messages in rate	`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=`*`TOPICNAME`*
Fetch request rate	`kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=`*`TOPICNAME`*
Produce request rate	`kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=`*`TOPICNAME`*

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 4-17). 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 4-17. Metrics for Each Topic

Name	JMX MBean
Partition size	`kafka.log:type=Log,name=Size,topic=`*`TOPICNAME`*`,partition=0`
Log segment count	`kafka.log:type=Log,name=NumLogSegments,topic=`*`TOPICNAME`*`,partition=0`
Log end offset	`kafka.log:type=Log,name=LogEndOffset,topic=`*`TOPICNAME`*`,partition=0`
Log start offset	`kafka.log:type=Log,name=LogStartOffset,topic=`*`TOPICNAME`*`,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 underreplicated. 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 11) 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 4-18.

Table 4-18. G1 Garbage Collection Metrics

Name	JMX MBean
Full GC cycles	`java.lang:type=GarbageCollector,name=G1 Old Generation`
Young GC cycles	`java.lang:type=GarbageCollector,name=G1 Young Generation`

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 IO, 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 uninterruptable 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 IO 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 `DEBUG` or `TRACE` levels. This logs information about every request sent to the broker. At `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 4-19 for an example.

Table 4-19. Kafka Producer Metric MBeans

Name	JMX MBean
Overall Producer	`kafka.producer:type=producer-metrics,client-id=`*`CLIENTID`*
Per-Broker	`kafka.producer:type=producer-node-metrics,client-id=`*`CLIENTID`*`,node-id=node-`*`BROKERID`*
Per-Topic	`kafka.producer:type=producer-topic-metrics,client-id=`*`CLIENTID`*`,topic=`*`TOPICNAME`*

Each of the metric beans in Table 4-19 have multiple attributes available to describe the state of the producer. The particular attributes that are of the most use are described in "Overall producer metrics" on page 102. Before proceeding, be sure you understand the semantics of how the producer works, as described in Chapter 2.

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 both 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 Mirror Maker, 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 that is 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 useable 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 new producer client, the new 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, 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 4-20.

Table 4-20. Kafka Consumer Metric MBeans

Name	JMX MBean
Overall Consumer	`kafka.consumer:type=consumer-metrics,client-id=`*`CLIENTID`*
Fetch Manager	`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=`*`CLIENTID`*
Per-Topic	`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=`*`CLIENTID`*`,topic=`*`TOPICNAME`*
Per-Broker	`kafka.consumer:type=consumer-node-metrics,client-id=`*`CLIENTID`*`,node-id=node-`*`BROKERID`*
Coordinator	`kafka.consumer:type=consumer-coordinator-metrics,client-id=`*`CLIENTID`*

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 (number of messages behind the broker) 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 108.

In order 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 between 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 4-21.

Table 4-21. Metrics to monitor

Client	Bean name
Consumer	bean `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID`, attribute `fetch-throttle-time-avg`
Producer	bean `kafka.producer:type=producer-metrics,client-id=CLIENTID`, attribute `produce-throttle-time-avg`

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 11 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 in order to reduce this complexity is to use Burrow. This is an open source application, originally developed by LinkedIn, which 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 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, 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 has reviewed the most critical ones to keep an eye on.

About the Authors

Neha Narkhede is cofounder and head of engineering at Confluent, a company backing the popular Apache Kafka messaging system. Prior to founding Confluent, Neha led streams infrastructure at LinkedIn, where she was responsible for LinkedIn's petabyte scale-streaming infrastructure built on top of Apache Kafka and Apache Samza. Neha specializes in building and scaling large distributed systems and is one of the initial authors of Apache Kafka. In the past she has worked on searching within the database at Oracle and holds a masters in computer science from Georgia Tech.

Gwen Shapira is a product manager at Confluent. She is a PMC member of the Apache Kafka project, has contributed Kafka integration to Apache Flume, and is a committer on Apache Sqoop. Gwen has 15 years of experience working with customers to design scalable data architectures. Formerly a software engineer at Cloudera, senior consultant at Pythian, Oracle ACE Director, and board member at NoCOUG. Gwen is a frequent speaker at industry conferences and contributes to multiple industry blogs including O'Reilly Radar.

Todd Palino is a senior staff site reliability engineer at LinkedIn, tasked with keeping the largest deployment of Apache Kafka, Zookeeper, and Samza fed and watered. He is responsible for architecture, day-to-day operations, and tools development, 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 with Apache Kafka at industry conferences and tech talks. Todd has spent more than 20 years in the technology industry running infrastructure services, most recently 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.

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 hollows in the high parts of trees. Raising young is a community effort, as there is at least one helper bird to help 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*.

The cover image is from *English Cyclopedia*. The cover fonts are URW Typewriter 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.