O'REILLY®

Understanding Message Brokers

Learn the Mechanics of Messaging through ActiveMQ and Kafka

Jakub Korab

Understanding Message Brokers

Learn the Mechanics of Messaging though ActiveMQ and Kafka

Jakub Korab

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

Understanding Message Brokers
by Jakub Korab

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/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Colleen Cole
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2017: First Edition

Revision History for the First Edition
2017-05-24: First Release

978-1-491-98153-5

[LSI]

Table of Contents

CHAPTER 1
Introduction

Intersystem messaging is one of the more poorly understood areas of IT. As a developer or architect you may be intimately familiar with various application frameworks, and database options. It is likely, however, that you have only a passing familiarity with how broker-based messaging technologies work. If you feel this way, don't worry—you're in good company.

People typically come into contact with messaging infrastructure in a very limited way. It is not uncommon to be pointed at a system that was set up a long time ago, or to download a distribution from the internet, drop it into a production-like environment, and start writing code against it. Once the infrastructure is pushed to production, the results can be mixed: message loss on failure, distribution not working the way you had expected, or brokers "hanging" your producers or not distributing messages to your consumers.

Does this sound in any way familiar?

A common scenario is that your messaging code will work fine—for a while. Until it does not. This period lulls many into a false sense of security, which leads to more code being written while holding on to misconceptions about fundamental behavior of the technology. When things start to go wrong you are left facing an uncomfortable truth: that you did not really understand the underlying behavior of the product or the trade-offs its authors chose to make, such as performance versus reliability, or transactionality versus horizontal scalability.

Without a high-level understanding of how brokers work, people make seemingly sensible assertions about their messaging systems such as:

- The system will never lose messages
- Messages will be processed in order
- Adding consumers will make the system go faster
- Messages will be delivered exactly once

Unfortunately, some of the above statements are based on assumptions that are applicable only in certain circumstances, while others are just incorrect.

This book will teach you how to reason about broker-based messaging systems by comparing and contrasting two popular broker technologies: Apache ActiveMQ and Apache Kafka. It will outline the use cases and design drivers that led to their developers taking very different approaches to the same domain—the exchange of messages between systems with a broker intermediary. We will go into these technologies from the ground up, and highlight the impacts of various design choices along the way. You will come away with a high-level understanding of both products, an understanding of how they should and should not be used, and an appreciation of what to look out for when considering other messaging technologies in the future.

Before we begin, let's go all the way back to basics.

What Is a Messaging System, and Why Do We Need One?

In order for two applications to communicate with each other, they must first define an interface. Defining this interface involves picking a transport or protocol, such as HTTP, MQTT, or SMTP, and agreeing on the shape of the messages to be exchanged between the two systems. This may be through a strict process, such as by defining an XML schema for an expense claim message payload, or it may be much less formal, for example, an agreement between two developers that some part of an HTTP request will contain a customer ID.

As long as the two systems agree on the shape of those messages and the way in which they will send the messages to each other, it is then possible for them to communicate with each other without concern for how the other system is implemented. The internals of those systems, such as the programming language or the application frameworks used, can vary over time. As long as the contract itself is maintained, then communication can continue with no change from the other side. The two systems are effectively decoupled by that interface.

Messaging systems typically involve the introduction of an intermediary between the two systems that are communicating in order to further decouple the sender from the receiver or receivers. In doing so, the messaging system allows a sender to send a message without knowing where the receiver is, whether it is active, or indeed how many instances of them there are.

Let's consider a couple of analogies of the types of problems that a messaging system addresses and introduce some basic terms.

Point-to-Point

> Alexandra walks into the post office to send a parcel to Adam. She walks up to the counter and hands the teller the parcel. The teller places the parcel behind the counter and gives Alexandra a receipt. Adam does not need to be at home at the moment that the parcel is sent. Alexandra trusts that the parcel will be delivered to Adam at some point in the future, and is free to carry on with the rest of her day. At some point later, Adam receives the parcel.

This is an example of the *point-to-point* messaging domain. The post office here acts as a distribution mechanism for parcels, guaranteeing that each parcel will be delivered once. Using the post office separates the act of sending a parcel from the delivery of the parcel.

In classical messaging systems, the point-to-point domain is implemented through *queues*. A queue acts as a first in, first out (FIFO) buffer to which one or more consumers can subscribe. Each message is delivered to only *one of the subscribed consumers*. Queues will typically attempt to distribute the messages fairly among the consumers. Only one consumer will receive a given message.

Queues are termed as being durable. *Durability* is a quality of service that guarantees that the messaging system will retain messages in

the absence of any active subscribers until a consumer next subscribes to the queue to take delivery of them.

Durability is often confused with *persistence*, and while the two terms come across as interchangeable, they serve different functions. Persistence determines whether a messaging system writes the message to some form of storage between receiving and dispatching it to a consumer. Messages sent to a queue may or may not be persistent.

Point-to-point messaging is used when the use case calls for a message to be acted upon once only. Examples of this include depositing funds into an account or fulfilling a shipping order. We will discuss later on why the messaging system in itself is incapable of providing once-only delivery and why queues can at best provide an *at-least-once* delivery guarantee.

Publish-Subscribe

> Gabriella dials in to a conference call. While she is connected, she hears everything that the speaker is saying, along with the rest of the call participants. When she disconnects, she misses out on what is said. On reconnecting, she continues to hear what is being said.

This is an example of the *publish-subscribe* messaging domain. The conference call acts as a broadcast mechanism. The person speaking does not care how many people are currently dialed into the call—the system guarantees that anyone who is currently dialed in will hear what is being said.

In classical messaging systems, the publish-subscribe messaging domain is implemented through *topics*. A topic provides the same sort of broadcast facility as the conference call mechanism. When a message is sent into a topic, it is distributed to *all subscribed consumers*.

Topics are typically *nondurable*. Much like the listener who does not hear what is said on the conference call when she disconnects, topic subscribers miss any messages that are sent while they are offline. For this reason, it can be said that topics provide an *at-most-once* delivery guarantee for each consumer.

Publish-subscribe messaging is typically used when messages are informational in nature and the loss of a single message is not particularly significant. For example, a topic might transmit temperature readings from a group of sensors once every second. A system

that subscribes to the topic that is interested in the current temperature will not be concerned if it misses a message—another will arrive shortly.

Hybrid Models

> A store's website places order messages onto a message "queue." A fulfilment system is the primary consumer of those messages. In addition, an auditing system needs to have copies of these order messages for tracking later on. Both systems cannot miss messages, even if the systems themselves are unavailable for some time. The website should not be aware of the other systems.

Use cases often call for a hybrid of publish-subscribe and point-to-point messaging, such as when multiple systems each want a copy of a message and require both durability and persistence to prevent message loss.

These cases call for a destination (the general term for queues and topics) that distributes messages much like a topic, such that each message is sent to a distinct system interested in those messages, but where each system can define multiple consumers that consume the inbound messages, much like a queue. The consumption type in this case is *once-per-interested-party*. These hybrid destinations frequently require durability, such that if a consumer disconnects, the messages that are sent in the meantime are received once the consumer reconnects.

Hybrid models are not new and can be addressed in most messaging systems, including both ActiveMQ (via virtual or composite destinations, which compose topics and queues) and Kafka (implicitly, as a fundamental design feature of its destination).

Now that we have some basic terminology and an understanding of why we might want to use a messaging system, let's jump into the details.

CHAPTER 2
ActiveMQ

ActiveMQ is best described as a classical messaging system. It was written in 2004, filling a need for an open source message broker. At the time if you wanted to use messaging within your applications, the only choices were expensive commercial products.

ActiveMQ was designed to implement the Java Message Service (JMS) specification. This decision was made in order to fill the requirement for a JMS-compliant messaging implementation in the Apache Geronimo project—an open source J2EE application server.

A messaging system (or message-oriented middleware, as it is sometimes called) that implements the JMS specification is composed of the following constructs:

Broker
: The centralized piece of middleware that distributes messages.

Client
: A piece of software that exchanges messages using a broker. This in turn is made up of the following artifacts:

  - Your code, which uses the JMS API.
  - The JMS API—a set of interfaces for interacting with the broker according to guarantees laid out in the JMS specification.
  - The system's client library, which provides the implementation of the API and communicates with the broker.

The client and broker communicate with each other through an application layer protocol, also known as a *wire protocol* (Figure 2-1). The JMS specification left the details of this protocol up to individual implementations.

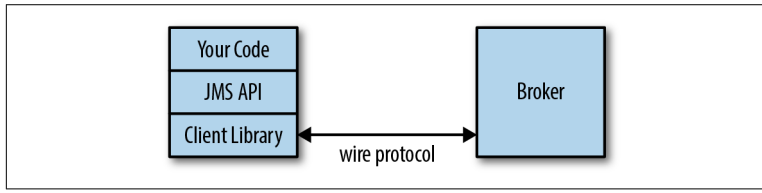

Figure 2-1. JMS overview

JMS uses the term *provider* to describe the vendor's implementation of the messaging system underlying the JMS API, which comprises the broker as well as its client libraries.

The choice to implement JMS had far-reaching consequences on the implementation decisions taken by the authors of ActiveMQ. The specification itself set out clear guidelines about the responsibilities of a messaging client and the broker that it communicates with, favoring to place obligation for the distribution and delivery of messages on the broker. The client's primary obligation is to interact with the destination (queue or topic) of the messages it sends. The specification itself focused on making the API interaction with the broker relatively simple.

This direction impacted heavily on the performance of ActiveMQ, as we will see later on. Adding to the complexities of the broker, the compatibility suite for the specification provided by Sun Microsystems had numerous corner cases, with their own performance impacts, that all had to be fulfilled in order for ActiveMQ to be considered JMS-compliant.

Connectivity

While the API and expected behavior were well defined by JMS, the actual protocol for communication between the client and the broker was deliberately left out of the JMS specification, so that existing brokers could be made JMS-compatible. As such, ActiveMQ was free to define its own wire protocol—OpenWire. OpenWire is used by the ActiveMQ JMS client library implementation, as well as

its .Net and C++ counterparts—NMS and CMS—which are subprojects of ActiveMQ, hosted at the Apache Software Foundation.

Over time, support for other wire protocols was added into ActiveMQ, which increased its interoperability options from other languages and environments:

AMQP 1.0
: The Advanced Message Queuing Protocol (ISO/IEC 19464:2014) should not be confused with its 0.X predecessors, which are implemented in other messaging systems, in particular within RabbitMQ, which uses 0.9.1. AMQP 1.0 is a general purpose binary protocol for the exchange of messages between two peers. It does not have the notion of clients or brokers, and includes features such as flow control, transactions, and various qualities of service (at-most-once, at-least-once, and exactly-once).

STOMP
: Simple/Streaming Text Oriented Messaging Protocol, an easy-to-implement protocol that has dozens of client implementations across various languages.

XMPP
: Extensible Messaging and Presence Protocol. Originally called Jabber, this XML-based protocol was originally designed for chat systems, but has been extended beyond its initial use cases to include publish-subscribe messaging.

MQTT
: A lightweight, publish-subscribe protocol (ISO/IEC 20922:2016) used for Machine-to-Machine (M2M) and Internet of Things (IoT) applications.

ActiveMQ also supports the layering of the above protocols over WebSockets, which enables full duplex communication between applications in a web browser and destinations in the broker.

With this in mind, these days when we talk about ActiveMQ, we no longer refer exclusively to a communications stack based on the JMS/NMS/CMS libraries and the OpenWire protocol. It is becoming quite common to mix and match languages, platforms, and external libraries that are best suited to the application at hand. It is possible, for example, to have a JavaScript application running in a browser using the Eclipse Paho MQTT library to send messages to

ActiveMQ over Websockets, and have those messages consumed by a C++ server process that uses AMQP via the Apache Qpid Proton library. From this perspective, the messaging landscape is becoming much more diverse.

Looking to the future, AMQP in particular is going to feature much more heavily than it has to date as components that are neither clients nor brokers become a more familiar part of the messaging landscape. The Apache Qpid Dispatch Router, for example, acts as a message router that clients connect to directly, allowing different destinations to be handled by distinct brokers, as well as providing a sharding facility.

When dealing with third-party libraries and external components, you need to be aware that they are of variable quality and may not be compatible with the features provided within ActiveMQ. As a very simple example, it is not possible to send messages to a queue via MQTT (without a bit of routing configured within the broker). As such, you will need to spend some time working through the options to determine the messaging stack most appropriate for your application requirements.

The Performance-Reliability Trade-off

Before we dive into the details of how point-to-point messaging in ActiveMQ works, we need to talk a bit about something that all data-intensive systems need to deal with: the trade-off between performance and reliability.

Any system that accepts data, be it a message broker or a database, needs to be instructed about how that data should be handled if the system fails. Failure can take many forms, but for the sake of simplicity, we will narrow it down to a situation where the system loses power and immediately shuts down. In a situation such as this, we need to reason about what happened to the data that the system had. If the data (in this case, messages) was in memory or a volatile piece of hardware, such as a cache, then that data will be lost. However, if the data had been sent to nonvolatile storage, such as a disk, then it will once again be accessible when the system is brought back online.

From that perspective, it makes sense that if we do not want to lose messages if a broker goes down, then we need to write them to per-

sistent storage. The cost of this particular decision is unfortunately quite high.

Consider that the difference between writing a megabyte of data to disk is between 100 to 1000 times slower than writing it to memory. As such, it is up to the application developer to make a decision as to whether the price of message reliability is worth the associated performance cost. Decisions such as these need to be made on a use case basis.

The performance-reliability trade-off is based on a spectrum of choices. The higher the reliability, the lower the performance. If you decide to make the system less reliable, say by keeping messages in memory only, your performance will increase significantly. The JMS defaults that ActiveMQ comes tuned with out of the box favor reliability. There are numerous mechanisms that allow you to tune the broker, and your interaction with it, to the position on this spectrum that best addresses your particular messaging use cases.

This trade-off applies at the level of individual brokers. However an individual broker is tuned, it is possible to scale messaging beyond this point through careful consideration of message flows and separation of traffic out over multiple brokers. This can be achieved by giving certain destinations their own brokers, or by partitioning the overall stream of messages either at the application level or through the use of an intermediary component. We will look more closely at how to consider broker topologies later on.

Message Persistence

ActiveMQ comes with a number of pluggable strategies for persisting messages. These take the form of persistence adapters, which can be thought of as engines for the storage of messages. These include disk-based options such as KahaDB and LevelDB, as well as the possibility of using a database via JDBC. As the former are most commonly used, we will focus our discussion on those.

When persistent messages are received by a broker, they are first written to disk into a *journal*. A journal is an append-only disk-based data structure made up of multiple files. Incoming messages are serialized into a protocol-independent object representation by the broker and are then marshaled into a binary form, which is then written to the end of the journal. The journal contains a log of all

incoming messages, as well as details of those messages that have been acknowledged as consumed by the client.

Disk-based persistence adapters maintain index files which keep track of where the next messages to be dispatched are positioned within the journal. When all of the messages from a journal file have been consumed, it will either be deleted or archived by a background worker thread within ActiveMQ. If this journal is corrupted during a broker failure, then ActiveMQ will rebuild it based on the information within the journal files.

Messages from all queues are written in the same journal files, which means that if a single message is unconsumed, the entire file (usually either 32 MB or 100 MB in size by default, depending on the persistence adapter) cannot be cleaned up. This can cause problems with running out of disk space over time.

Classical message brokers are not intended as a long-term storage mechanism—consume your messages!

Journals are an extremely efficient mechanism for the storage and subsequent retrieval of messages, as the disk access for both operations is sequential. On conventional hard drives this minimizes the amount of disk seek activity across cylinders, as the heads on a disk simply keep reading or writing over the sectors on the spinning disk platter. Likewise on SSDs, sequential access is much faster than random access, as the former makes better use of the drive's memory pages.

Disk Performance Factors

There are a number of factors that will determine the speed at which a disk can operate. To understand these, let us consider the way that we write to disk through a simplified mental model of a pipe (Figure 2-2).

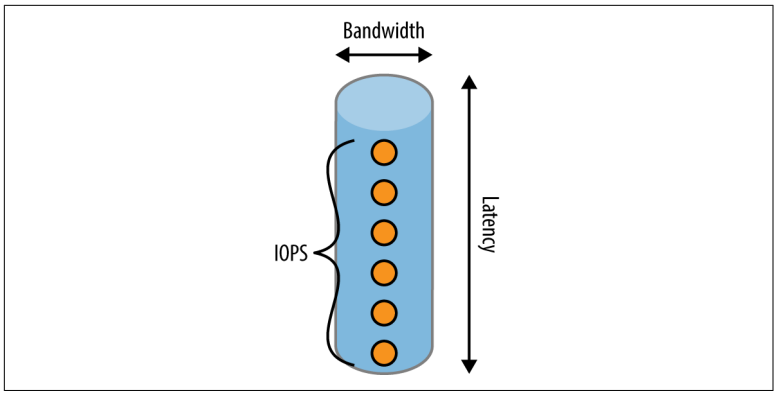

Figure 2-2. Pipe model of disk performance

A pipe has three dimensions:

Length
: This corresponds with the *latency* that is expected for a single operation to complete. On most local disks this is pretty good, but can become a major limiting factor in cloud environments where the local disk is actually across a network. For example, at the time of writing (April 2017) Amazon guarantees that writes to their EBS storage devices will be performed in "under 2 ms." If we are performing writes sequentially, then this gives a maximum throughput limit of 500 writes per second.

Width
: This will dictate the carrying capacity or *bandwidth* of a single operation. Filesystem caches exploit this property by combining many small writes into a smaller set of larger write operations performed against the disk.

The carrying capacity over a period of time
: This idea, visualized as the number of things that can be in the pipe at the same time, is expressed by a metric called *IOPS* (input/output operations per second). IOPS is commonly used by storage manufacturers and cloud providers as a performance measurement. A hard disk will have different IOPS values in different contexts: whether the workload is mostly made up of reads, writes, or a combination of the two; and whether those operations are sequential, random-access, or mixed. The IOPS measurements that are most interesting from a broker perspec-

tive are sequential reads and writes, as these correspond to reading and writing journal logs.

The maximum throughput of a message broker will be defined by *the first of these limits to be hit*, and the tuning of a broker largely depends on how the interaction with the disks is performed. This is not just a factor of how the broker is configured, for instance, but also depends on how the producers are interacting with the broker. As with anything performance-related, it is necessary to test the broker under a representative workload (i.e., as close to real messages as possible) and on the actual storage setup that will be used in production. This is done in order to get an understanding of how the system will behave in reality.

The JMS API

Before we go into the details of how ActiveMQ exchanges messages with clients, we first need to examine the JMS API. The API defines a set of programming interfaces used by client code:

`ConnectionFactory`
: This is the top-level interface used for establishing connections with a broker. In a typical messaging application, there exists a single instance of this interface. In ActiveMQ this is the `ActiveMQConnectionFactory`. At a high level, this construct is instructed with the location of the message broker, as well as the low-level details of how it should communicate with it. As implied by the name, a `ConnectionFactory` is the mechanism by which `Connection` objects are created.

`Connection`
: This is a long lived object that is roughly analogous to a TCP connection—once established, it typically lives for the lifetime of the application until it is shut down. A connection is thread-safe and can be worked with by multiple threads at the same time. `Connection` objects allow you to create `Session` objects.

`Session`
: This is a thread's handle on communication with a broker. Sessions are not thread-safe, which means that they cannot be accessed by multiple threads at the same time. A `Session` is the main transactional handle through which the programmer may commit and roll back messaging operations, if it is running in

transacted mode. Using this object, you create `Message`, `MessageConsumer`, and `MessageProducer` objects, as well as get handles on `Topic` and `Queue` objects.

`MessageProducer`
: This interface allows you to send a message to a destination.

`MessageConsumer`
: This interface allows the developer to receive messages. There are two mechanisms for retrieving a message:

  - Registering a `MessageListener`. This is a message handler interface implemented by you that will sequentially process any messages pushed by the broker using a single thread.
  - Polling for messages using the `receive()` method.

`Message`
: This is probably the most important construct as it is the one that carries your data. Messages in JMS are composed of two aspects:

  - Metadata about the message. A message contains headers and properties. Both of these can be thought of as entries in a map. Headers are well-known entries, specified by the JMS specification and accessible directly via the API, such as `JMSDestination` and `JMSTimestamp`. Properties are arbitrary pieces of information about the message that you set to simplify message processing or routing, without the need to read the message payload itself. You may, for instance, set an `AccountID` or `OrderType` header.
  - The body of the message. A number of different message types can be created from a `Session`, based on the type of content that will be sent in the body, the most common being `TextMessage` for strings and `BytesMessage` for binary data.

How Queues Work: A Tale of Two Brains

A useful, though imprecise, model of how ActiveMQ works is that of two halves of a brain. One part is responsible for accepting messages from producers, and the other dispatches those messages to

consumers. In reality, the relationship is much more complex for performance optimization purposes, but the model is adequate for a basic understanding.

Producing Messages into a Queue

Let's consider the interaction that occurs when a message is sent. Figure 2-3 shows us a simplified model of the process by which messages are accepted by the broker; it does not match perfectly to the behavior in every case, but is good enough to get a baseline understanding.

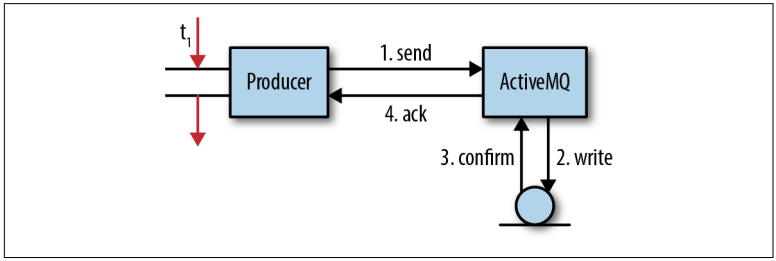

Figure 2-3. Producing messages to JMS

Within the client application, a thread has gotten a handle on a `MessageProducer`. It has created a `Message` with the intended message payload and invokes `MessageProducer.send("orders", message)`, with the target destination of the message being a queue. As the programmer did not want to lose the message if the broker went down, the `JMSDeliveryMode` header of the message was set to `PERSISTENT` (the default behavior).

At this point (1) the sending thread calls into the client library and marshals the message into OpenWire format. The message is then sent over to the broker.

Within the broker, a receiving thread accepts the message off the wire and unmarshals it into an internal object representation. The message object is then passed to the persistence adapter, which marshals the message using the Google Protocol Buffers format and writes it to storage (2).

Once the message has been written to storage, the persistence adapter needs to get a confirmation that the message has actually been written (3). This is typically the slowest part of the entire interaction; more on this later.

Once the broker is satisfied that the message has been stored, it responds with an acknowledgement back to the client (4). The client thread that originally invoked the `send()` operation is then free to continue performing its processing.

This waiting for acknowledgement of persistent messages is fundamental to the guarantee that the JMS API provides—if you want the message to be persisted, you presumably also care about whether the message was accepted by the broker in the first place. There are a number of reasons why this might not be possible, for instance, a memory or storage limit being reached. Instead of crashing, the broker will either pause the send operation, causing the producer to wait until there are enough system resources to process the message (a process called Producer Flow Control), or it will send a negative acknowledgement back to the producer, triggering an exception to be thrown. The exact behavior is configurable on a per-broker basis.

There is a substantial amount of I/O interaction happening in this simple operation, with two network operations between the producer and the broker, one storage operation, and a confirmation step. The storage operation could be a simple disk write or another network hop to a storage server.

This raises an important point about message brokers: they are extremely I/O intensive and very sensitive to the underlying infrastructure, in particular, disks.

Let's take a closer look at the confirmation step (3) in the above interaction. If the persistence adapter is file based, then storing a message involves a write to the filesystem. If this is the case, then why would we need a confirmation that a write has been completed? Surely the act of completing a write means that a write has occurred?

Not quite. As tends to be the case with these things, the closer you look at a something, the more complex it turns out to be. The culprit in this particular case is *caches*.

Caches, Caches Everywhere

When an operating system process, such as a broker, writes to disk, it interacts with the filesystem. The filesystem is a process that abstracts away the details of interacting with the underlying storage medium by providing an API for file operations, such as `OPEN`,

`CLOSE`, `READ`, and `WRITE`. One of those functions is to *minimize the amount of writes* by buffering data written to it by operating system processes into blocks that can be written out to disk at the same time. Filesystem writes, which seem to interact with disks, are actually written to this *buffer cache*.

Incidentally, this is why your computer complains when you remove a USB stick without safely ejecting it—those files you copied may not actually have been written!

Once data makes it beyond the buffer cache, it hits the next level of caching, this time at the hardware level—the *disk drive controller cache*. These are of particular note on RAID-based systems, and serve a similar function as caching at the operating system level: to minimize the amount of interactions that are needed with the disks themselves. These caches fall into two categories:

write-through
: Writes are passed to the disk on arrival.

write-back
: Writes are only performed against the disks once the buffer has reached a certain threshold.

Data held in these caches can easily be lost when a power failure occurs, as the memory used by them is typically *volatile*. More expensive cards have battery backup units (BBUs) which maintain power to the caches until the overall system can have power restored, at which point the data is written to disk.

The last level of caches is on the disks themselves. *Disk caches* exist on hard disks (both standard hard drives and SSDs) and can be write-through or write-back. Most commercial drives use caches that are write-back and volatile, again meaning that data can be lost in the event of a power failure.

Going back to the message broker, the confirmation step is needed to make sure that the data has actually made it all the way down to the disk. Unfortunately, it is up to the filesystem to interact with these hardware buffers, so all that a process such as ActiveMQ can do is to send the filesystem a signal that it wants all system buffers to synchronize with the underlying device. This is achieved by the broker calling `java.io.FileDescriptor.sync()`, which in turn triggers the `fsync()` POSIX operation.

This syncing behavior is a JMS requirement to ensure that all messages that are marked as persistent are actually saved to disk, and is therefore performed after the receipt of each message or set of related messages in a transaction. As such, the speed with which the disk can `sync()` is of critical importance to the performance of the broker.

Internal Contention

The use of a single journal for all queues adds an additional complication. At any given time, there may be multiple producers all sending messages. Within the broker, there are multiple threads that receive these messages from the inbound socket connections. Each thread needs to persist its message to the journal. As it is not possible for multiple threads to write to the same file at the same time because the writes would conflict with each other, the writes need to be queued up through the use of a mutual exclusion mechanism. We call this *thread contention*.

Each message must be fully written and synced before the next message can be processed. This limitation impacts all queues in the broker at the same. So the performance of how quickly a message can be accepted is the write time to disk, plus any time waiting on other threads to complete their writes.

ActiveMQ includes a write buffer into which receiving threads write their messages while they are waiting for the previous write to complete. The buffer is then written in one operation the next time the message is available. Once completed, the threads are then notified. In this way, the broker maximizes the use of the storage bandwidth.

To minimize the impact of thread contention, it is possible to assign sets of queues to their own journals through the use of the mKahaDB adapter. This approach reduces the wait times for writes, as at any one time threads will likely be writing to different journals and will not need to compete with each other for exclusive access to any one journal file.

Transactions

The advantage of using a single journal for all queues is that from the broker authors' perspective it is much simpler to implement transactions.

Let us consider an example where multiple messages are sent from a producer to multiple queues. Using a transaction means that the entire set of sends must be treated as a single atomic operation. In this interaction, the ActiveMQ client library is able to make some optimizations which greatly increase send performance.

In the operation shown in Figure 2-4, the producer sends three messages, all to different queues. Instead of the normal interaction with the broker, where each message is acknowledged, the client sends all three messages asynchronously, that is, without waiting for a response. These messages are held in memory by the broker. Once the operation is completed, the producer tells its session to commit, which in turn causes the broker to perform a single large write with a single sync operation.

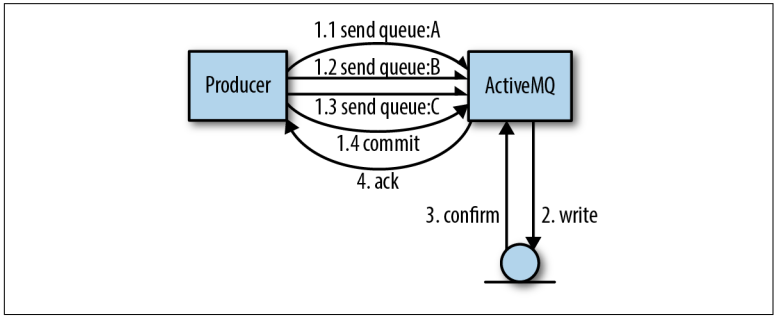

Figure 2-4. Producing messages within a transaction

This type of operation sees ActiveMQ using two optimizations for a performance increase:

- A removal of the wait time before the next send is possible in the producer
- Combining many small disk operations into one larger one—this makes use of the width dimension of our pipe model of disks

If you were to compare this with a situation where each queue was stored in its own journal, then the broker would need to ensure some form of transactional coordination between each of the writes.

Consuming Messages from a Queue

The process of message consumption begins when a consumer expresses a demand for messages, either by setting up a `MessageListener` to process messages as they arrive or by making calls to the `MessageConsumer.receive()` method (Figure 2-5).

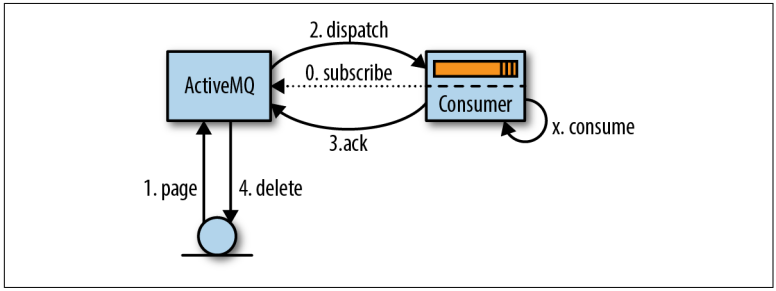

Figure 2-5. Consuming messages via JMS

When ActiveMQ is aware of a consumer, it pages messages from storage into memory for distribution (1). These messages are then *dispatched* to the consumer (2), often in multiples to minimize the amount of network communication. The broker keeps track of which messages have been dispatched and to which consumer.

The messages that are received by the consumer are not immediately processed by the application code, but are placed into an area of memory known as the *prefetch buffer*. The purpose of this buffer is to even out message flow so that the broker can feed the consumer messages as they become available for dispatch, while the consumer can consume them in an orderly fashion, one at a time.

At some point after arriving in the prefetch buffer, messages are consumed by the application logic (X), and an acknowledgement of consumption is sent back to the broker (3). The time ordering between the processing of the message and its acknowledgement is configurable through a setting on the JMS Session called the *acknowledgement mode*, which we will discuss in a little while.

Once a message acknowledgement is received by the broker, the message is removed from memory and deleted from the message

store (4). The term "deletion" is somewhat misleading, as in reality a record of the acknowledgement is written into the journal and a pointer within the index is incremented. The actual deletion of the journal file containing the message will be garbage collected by a background thread based on this information.

The behavior described above is a simplification to aid understanding. In reality, ActiveMQ does not simply page from disk, but instead uses a cursor mechanism between the receiving part of the broker and the dispatching part in order to minimize interaction with the broker's storage wherever possible. Paging, as described above, is one of the modes used in this mechanism. Cursors can be viewed as an application-level cache which needs to be kept synchronized with the broker's storage. The coherency protocol involved is a large part of what makes ActiveMQ's dispatching mechanism much more complex than that of Kafka, which is described in the next chapter.

Acknowledgement Modes and Transactions

The various acknowledgement modes that specify the order between consumption and acknowledgement have a substantial impact on what logic needs to be implemented in the client. They are as follows:

`AUTO_ACKNOWLEDGE`
: This is the most commonly used mode, probably because it has the word `AUTO` in it. This mode causes the client library to acknowledge the message *at the same time* as the message is consumed via a call to `receive()`. This means that if the business logic triggered by the message throws an exception, the message is lost, as it has already been deleted on the broker. If message consumption is via a listener, then the message will only be acknowledged when the listener is successfully completed.

`CLIENT_ACKNOWLEDGE`
: An acknowledgement will only be sent when the consumer code calls the `Message.acknowledge()` method explicitly.

`DUPS_OK_ACKNOWLEDGE`
: Here, acknowledgements will be buffered up in the consumer before all being sent at the same time to reduce the amount of

network traffic. However, should the client system shut down, then the acknowledgements will be lost and the messages will be re-dispatched and processed a second time. The code must therefore deal with the likelihood of duplicate messages.

Acknowledgement modes are supplemented by a transactional consumption facility. When a `Session` is created, it may be flagged as being transacted. This means that it is up to the programmer to explicitly call `Session.commit()` or `Session.rollback()`. On the consumption side, transactions expand the range of interactions that the code can perform as a single atomic operation. For example, it is possible to consume and process multiple messages as a single unit, or to consume a message from one queue and then send to another queue using the same `Session`.

Dispatch and Multiple Consumers

So far we have discussed the behavior of message consumption with a single consumer. Let's now consider how this model applies to multiple consumers.

When more than one consumer subscribes to a queue, the default behavior of the broker is to dispatch messages in a round-robin fashion to consumers that have space in their prefetch buffers. The messages will be dispatched in the order that they arrived on the queue—this is the only first in, first out (FIFO) guarantee provided.

When a consumer shuts down unexpectedly, any messages that had been dispatched to it but had not yet been acknowledged will be re-dispatched to another available consumer.

This raises an important point: even where consumer transactions are being used, there is no guarantee that a message will not be processed multiple times.

Consider the following processing logic within a consumer:

1. A message is consumed from a queue; a transaction starts.
2. A web service is invoked with the contents of the message.
3. The transaction is committed; an acknowledgement is sent to the broker.

If the client terminates between step 2 and step 3, then the consumption of the message has already affected some other system

through the web service call. Web service calls are HTTP requests, and as such are not transactional.

This behavior is true of all queueing systems—even when transactional, they cannot guarantee that the processing of their messages will not be free of side effects. Having looked at the processing of messages in details, we can safely say that:

There is no such thing as exactly-once message delivery.

Queues provide an *at-least-once* delivery guarantee, and sensitive pieces of code should always consider the possibility of receiving duplicate messages. Later on we will discuss how a messaging client can apply idempotent consumption to keep track of previously seen messages and discard duplicates.

Message Ordering

Given a set of messages that arrive in the order `[A, B, C, D]`, and two consumers `C1` and `C2`, the normal distribution of messages will be as follows:

```
C1: [A, C]
C2: [B, D]
```

Since the broker has no control over the performance of consumer processes, and since the order of processing is concurrent, it is non-deterministic. If `C1` is slower than `C2`, the original set of messages could be processed as `[B, D, A, C]`.

This behavior can be surprising to newcomers, who expect that messages will be processed in order, and who design their messaging application on this basis. The requirement for messages that were sent by the same sender to be processed in order relative to each other, also known as *causal ordering* is quite common.

Take as an example the following use case taken from online betting:

1. A user account is set up.
2. Money is deposited into the account.
3. A bet is placed that withdraws money from the account.

It makes sense, therefore, that the messages must be processed in the order that they were sent for the overall account state to make sense. Strange things could happen if the system tried to remove money

from an account that had no funds. There are, of course, ways to get around this.

The *exclusive consumer* model involves dispatching all messages from a queue to a single consumer. Using this approach, when multiple application instances or threads connect to a queue, they subscribe with a specific destination option: `my.queue?consumer.exclusive=true`. When an exclusive consumer is connected, it receives all of the messages. When a second consumer connects, it receives no messages until the first one disconnects. This second consumer is effectively a warm-standby, while the first consumer will now receive messages in the exact same order as they were written to the journal—in causal order.

The downside of this approach is that while the processing of messages is sequential, it is a performance bottleneck as all messages must be processed by a single consumer.

To address this type of use case in a more intelligent way, we need to re-examine the problem. Do *all* of the messages need to be processed in order? In the betting use case above, only the messages related to a single account need to be sequentially processed. ActiveMQ provides a mechanism for dealing with this situation, called *JMS message groups*.

Message groups are a type of partitioning mechanism that allows producers to categorize messages into groups that will be sequentially processed according to a business key. This business key is set into a message property named `JMSXGroupID`.

The natural key to use in the betting use case would be the account ID.

To illustrate how dispatch works, consider a set of messages that arrive in the following order:

```
[(A, Group1), (B, Group1), (C, Group2), (D, Group3), (E, Group2)]
```

When a message is processed by the dispatching mechanism in ActiveMQ, and it sees a `JMSXGroupID` that has not previously been seen, that key is assigned to a consumer on a round-robin basis. From that point on, all messages with that key will be sent to that consumer.

Here, the groups will be assigned between two consumers, `C1` and `C2`, as follows:

```
C1: [Group1, Group3]
C2: [Group2]
```

The messages will be dispatched and processed as follows:

```
C1: [(A, Group1), (B, Group1), (D, Group3)]
C2: [(C, Group2), (E, Group2)]
```

If a consumer fails, then any groups assigned to it will be reallocated between the remaining consumers, and any unacknowledged messages will be redispatched accordingly. So while we can guarantee that all related messages will be processed in order, we cannot say that they will be processed by the same consumer.

High Availability

ActiveMQ provides high availability through a master-slave scheme based on shared storage. In this arrangement, two or more (though usually two) brokers are set up on separate servers with their messages being persisted to a message store located at an external location. The message store cannot be used by multiple broker instances at the same time, so its secondary function is to act as a locking mechanism to determine which broker gets exclusive access (Figure 2-6).

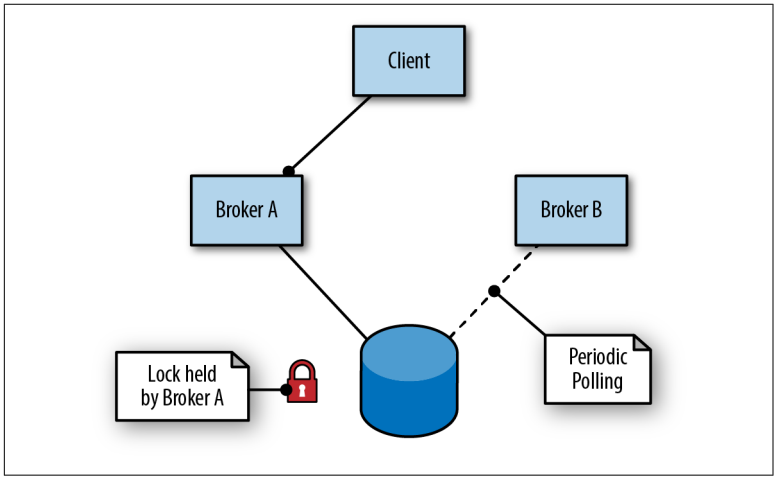

Figure 2-6. Broker A is master while Broker B is in standby as slave

The first broker to connect to the store (Broker A) takes on the role of the master and opens its ports to messaging traffic. When a second broker (Broker B) connects to the store, it attempts to acquire

the lock, and as it is unable to, pauses for a short period before attempting to acquire the lock again. This is known as holding back in a slave state.

It the meantime, the client alternates between the addresses of two brokers in an attempt to connect to an inbound port, known as the transport connector. Once a master broker is available, the client connects to its port and can produce and consume messages.

When Broker A, which has held the role of the master, fails due to a process outage (Figure 2-7), the following events occur:

1. The client is disconnected, and immediately attempts to reconnect by alternating between the addresses of the two brokers.
2. The lock within the message is released. The timing of this varies between store implementations.
3. Broker B, which has been in slave mode periodically attempting to acquire the lock, finally succeeds and takes over the role of the master, opening its ports.
4. The client connects to Broker B and continues its work.

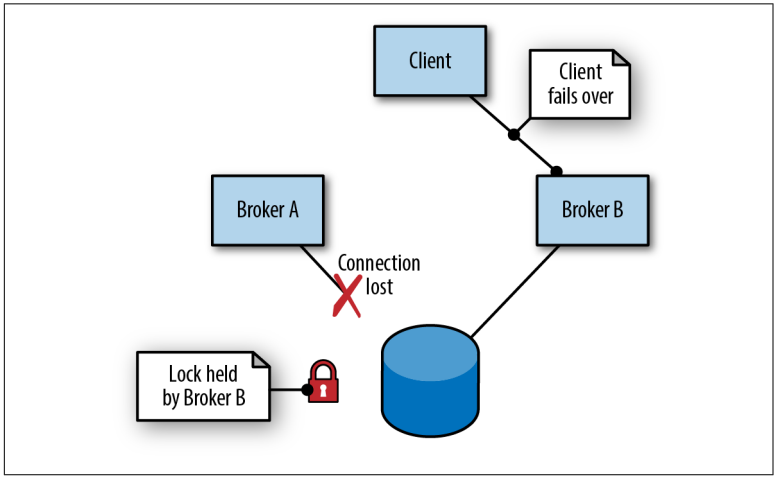

Figure 2-7. Broker A terminates, losing connection to store; Broker B takes over as master

NOTE

Logic to alternate between multiple broker addresses is not guaranteed to be built into the the client library, as it is with the JMS/NMS/CMS implementations. If a library provides only reconnection to a single address, then it may be necessary to place the broker pair behind a load balancer, which also needs to be made highly available.

The primary disadvantage of this approach is that it requires multiple physical servers to facilitate a single logical broker. In this scenario, one broker server out of the two is idle, waiting for its partner to fail before it can commence work.

The approach also has the additional complexity of requiring that the broker's underlying storage, whether it is a shared network file system or a database, is also highly available. This brings additional hardware and administration costs to a broker setup. It is tempting in this scenario to reuse existing highly available storage installations used by other parts of your infrastructure, such as a database, but this is a mistake.

It is important to remember that the disk is the main limiter on overall broker performance. If the disk itself is used concurrently by a process other than the message broker, then the disk interaction of that process is likely to slow down broker writes and therefore the rate at which messages can flow through the system. These sorts of slowdowns are difficult to diagnose, and the only way that they can be worked around is to separate the two processes onto different storage volumes.

In order to ensure consistent broker performance, dedicated and exclusive storage is required.

Scaling Up and Out

At some point in a project's lifetime, it may hit up against a performance limit of the message broker. These limits are typically down to resources, in particular the interaction of ActiveMQ with its underlying storage. These problems are usually due to the volume of messages or conflicts in messaging throughput between destinations, such as where one queue floods the broker at a peak time.

There are a number of ways to extract more performance out of a broker infrastructure:

- Do not use persistence unless you need to. Some use cases tolerate message loss on failure, especially ones when one system feeds full snapshot state to another over a queue, either periodically or on request.
- Run the broker on faster disks. In the field, significant differences in write throughput have been seen between standard HDDs and memory-based alternatives.
- Make better use of disk dimensions. As shown in the pipe model of disk interaction outlined earlier, it is possible to get better throughput by using transactions to send groups of messages, thereby combining multiple writes into a larger one.
- Use traffic partitioning. It is possible to get better throughput by splitting destinations over one of the following:
  - Multiple disks within the one broker, such as by using the mKahaDB persistance adapter over multiple directories with each mounted to a different disk.
  - Multiple brokers, with the partitioning of traffic performed manually by the client application. ActiveMQ does not provide any native features for this purpose.

One of the most common causes of broker performance issues is simply trying to do too much with a single instance. Typically this arises in situations where the broker is naively shared by multiple applications without consideration of the broker's existing load or understanding of capacity. Over time a single broker is loaded more and more until it no longer behaves adequately.

The problem is often caused at the design phase of a project where a systems architect might come up with a diagram such as Figure 2-8.

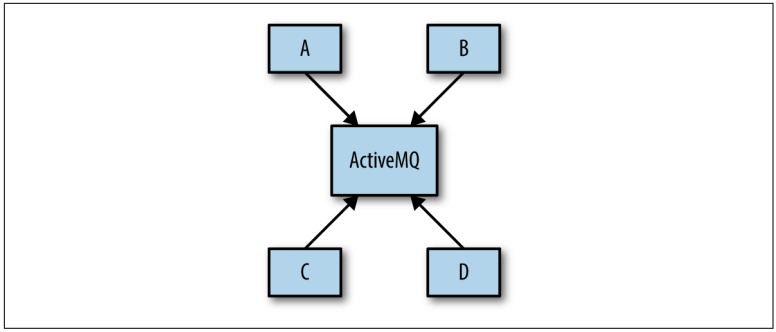

Figure 2-8. Conceptual view of a messaging infrastructure

The intent is for multiple applications to communicate with each other asynchronously via ActiveMQ. The intent is not refined further, and the diagram then forms the basis of a physical broker installation. This approach has a name—the Universal Data Pipeline.

This misses a fundamental analysis step between the conceptual design above and the physical implementation. Before going off to build a concrete setup, we need to perform an analysis that will be then used to inform a physical design. The first step of this process should be to identify which systems communicate with each other—a simple boxes and arrows diagram will suffice (Figure 2-9).

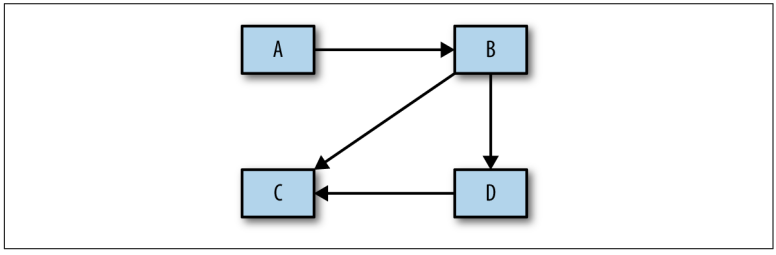

Figure 2-9. Sketch of message flows between systems

Once this is established, you can drill down into the details to answer questions such as:

- How many queues and topics are being used?
- What sorts of message volumes are expected over each?

- How big are the messages on each destination? Large messages can cause issues in the paging process, leading to memory limits being hit and blocking the broker.
- Are the message flows going to be uniform over the course of the day, or are there bursts due to batch jobs? Large bursts on one less-used queue might interfere with timely disk writes for high-throughput destinations.
- Are the systems in the same data center or different ones? Remote communication implies some form of broker networking.

The idea is to identify individual messaging use cases that can be combined or split out onto separate brokers (Figure 2-10). Once broken down in this manner, the use cases can be simulated in conjunction with each other using ActiveMQ's Performance Module to identify any issues.

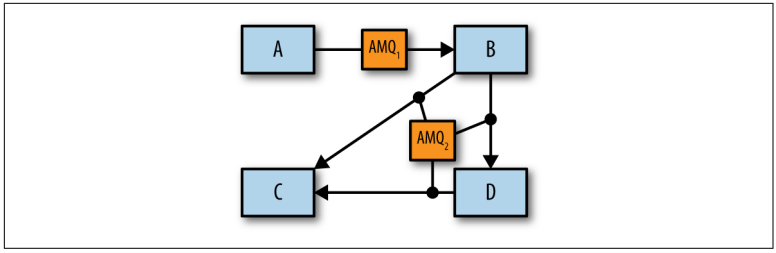

Figure 2-10. Identifying individual brokers

Once the appropriate number of logical brokers is identified, you can then work out how to implement these at the physical level through high availability setups and broker networks.

Summary

In this chapter we have examined the mechanics by which ActiveMQ receives and distributes messages. We discussed features that are enabled by this architecture, including sticky load-balancing of related messages and transactions. In doing so we introduced a set of concepts common to all messaging systems, including wire protocols and journals. We also looked in some detail at the complexities of writing to disk and how brokers can make use of techniques such as batching writes in order to increase performance. Finally, we examined how ActiveMQ can be made highly available,

and how it can be scaled beyond the capacity of an individual broker.

In the next chapter we will take a look at Apache Kafka and how its architecture reimagines the relationship between clients and brokers to provide an incredibly resilient messaging pipeline with many times greater throughput than a regular message broker. We will discuss the functionality that it trades off to achieve this, and examine in brief the application architectures that it enables.

CHAPTER 3
Kafka

Kafka was designed at LinkedIn to get around some of the limitations of traditional message brokers, and to avoid the need to set up different message brokers for different point-to-point setups, as described in "Scaling Up and Out" on page 28. LinkedIn's use cases were predominantly based around ingesting very large volumes of data such as page clicks and access logs in a unidirectional way, while allowing multiple systems to consume that data without affecting the performance of producers or other consumers. In effect, Kafka's reason for being is to enable the sort of messaging architecture that the Universal Data Pipeline describes.

Given this end goal, other requirements naturally emerged. Kafka had to:

- Be extremely fast
- Allow massive message throughput
- Support publish-subscribe as well as point-to-point
- Not slow down with the addition of consumers; both queue and topic performance degrades in ActiveMQ as the number of consumers rise on a destination
- Be horizontally scalable; if a single broker that persists messages can only do so at the maximum rate of the disk, it makes sense that to exceed this you need to go beyond a single broker instance
- Permit the retention and replay of messages

In order to achieve all of this, Kafka adopted an architecture that redefined the roles and responsibilities of messaging clients and brokers. The JMS model is very broker-centric, where the broker is responsible for the distribution of messages, and clients only have to worry about sending and receiving messages. Kafka, on the other hand, is client-centric, with the client taking over many of the functions of a traditional broker, such as fair distribution of related messages to consumers, in return for an extremely fast and scalable broker. To people coming from a traditional messaging background, working with Kafka requires a fundamental shift in perspective.

This engineering direction resulted in a messaging infrastructure that is capable of many orders of magnitude higher throughput than a regular broker. As we will see, this approach comes with trade-offs than meant that Kafka is not suitable for certain types of workloads and installations.

Unified Destination Model

To enable the requirements outlined above, Kafka unified both publish-subscribe and point-to-point messaging under a single destination type—the *topic*. This is confusing for people coming from a messaging background where the word topic refers to a broadcast mechanism from which consumption is nondurable, Kafka topics should be considered a hybrid destination type, as defined in the introduction to this book.

NOTE

For the remainder of this chapter, unless we explicitly state otherwise, the term "topic" will refer to a Kafka topic.

In order to fully understand how topics behave and the guarantees that they provide, we need to first consider how they are implemented within Kafka.

Each topic in Kafka has its own journal.

Producers that send messages to Kafka append to this journal, and consumers read from the journal through the use of pointers, which are continually moved forward. Periodically, Kafka removes the oldest parts of the journal, regardless of whether the messages con-

tained within have been read or not. It is a central part of Kafka's design that the broker is not concerned with whether its messages are consumed—that responsibility belongs to the client.

NOTE

The terms "journal" and "pointer" do not appear in the Kafka documentation. These well-known terms are used here in order to aid understanding.

This model is completely different from ActiveMQ's, where the messages from all queues are stored in the same journal, and the broker marks messages as deleted once they have been consumed.

Let's now drill down a bit and consider the topic's journal in greater depth.

A Kafka journal is composed of multiple partitions (Figure 3-1). Kafka provides strong ordering guarantees in each partition. This means that messages written into a partition in a particular order will be read in the same order. Each partition is implemented as a rolling log file that contains a *subset* of all of the messages sent into the topic by its producers. A topic created has a single partition by default. This idea of partitioning is central to Kafka's ability to scale horizontally.

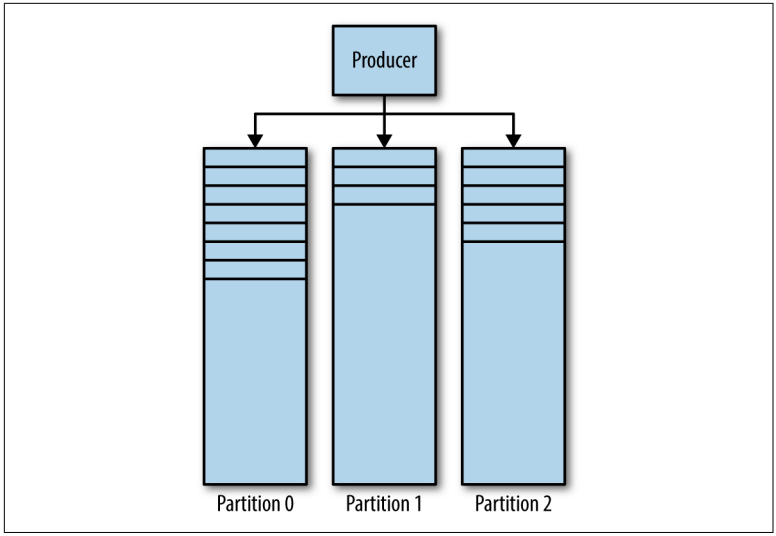

Figure 3-1. Kafka partitions

When a producer sends a message into a Kafka topic, it decides which partition the message is to be sent to. We will look at this in more detail later.

Consuming Messages

A client that wants to consume messages controls a named pointer, called a *consumer group*, that points to a message *offset* in a partition. The offset is an incrementally numbered position that starts at 0 at the beginning of the partition. This consumer group, referenced in the API through a user-defined `group_id`, corresponds to a *single logical consumer or system*.

Most systems that use messaging consume from a destination via multiple instances and threads in order to process messages in parallel. As such, there will typically be many consumer instances sharing the same consumer group.

The problem of consumption can be broken down as follows:

- A topic has multiple partitions
- Many consumer groups can consume from a topic at the same time
- A consumer group can have many separate instances

This is a nontrivial, many-to-many problem. To understand how Kafka addresses the relationship between consumer groups, consumer instances, and partitions, we will consider a series of progressively more complex consumption scenarios.

Consumers and Consumer Groups

Let us take as a starting point a topic with a single partition (Figure 3-2).

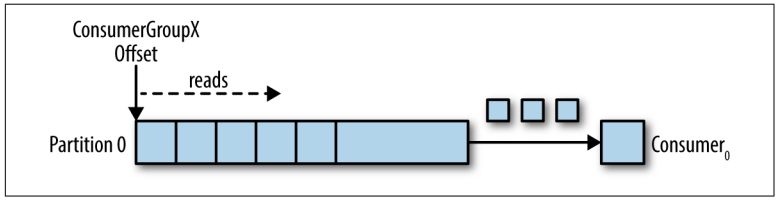

Figure 3-2. Consumer reading from a partition

When a consumer instance connects with its own `group_id` to this topic, it is assigned a partition to consume from and an offset in that partition. The position of this offset is configurable within the client as either pointing to the latest position (the newest message) or the earliest (the oldest message). The consumer polls for messages from the topic, which results in reading them sequentially from the log.

The offset position is regularly committed back to Kafka and stored as messages on an internal topic in `__consumer_offsets`. The consumed messages are not deleted in any way, unlike a regular broker, and the client is free to rewind the offset to reprocess messages that it has already seen.

When a second logical consumer connects with a different `group_id`, it controls a second pointer that is independent of the first (Figure 3-3). As such, a Kafka topic acts like a queue where a single consumer exists, and like a regular pub-sub topic where multiple consumers are subscribed, with the added advantage that all messages are persisted and can be processed multiple times.

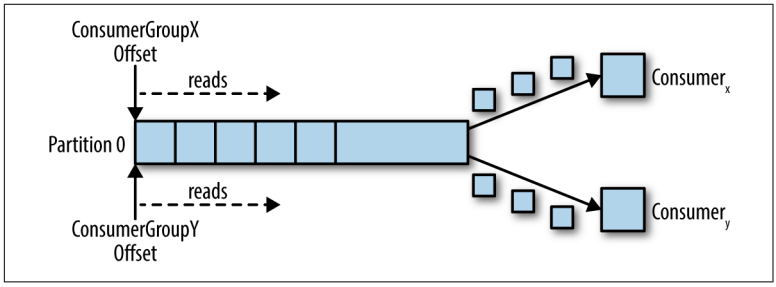

Figure 3-3. Two consumers in different consumer groups reading from the same partition

Consumers Within a Consumer Group

Where a single consumer instance reads from a partition, it has full control of the pointer and processes the messages, as described in the previous section.

If multiple consumer instances were to connect with the same `group_id` to a topic with one partition, the instance that connected *last* will be assigned control of the pointer, and it will receive all of the messages from that point onward (Figure 3-4).

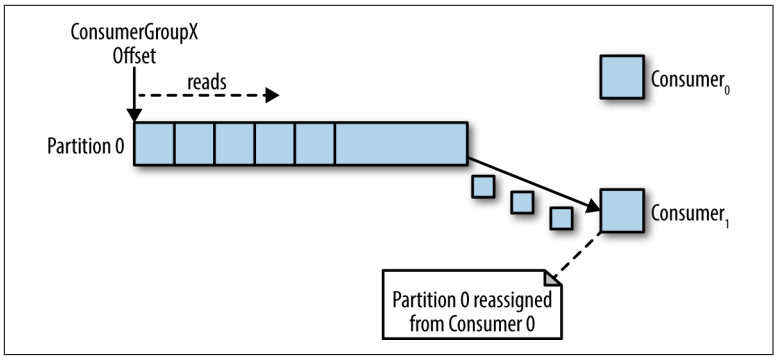

Figure 3-4. Two consumers in the same consumer group reading from the same partition

This mode of processing where consumer instances outnumber partitions can be thought of as a type of exclusive-consumer. It can be useful if you want active-passive (or hot-warm) clustering of your consumer instances, though having multiple consumers processing in parallel (active-active, or hot-hot) is much more typical that having consumers on standby.

NOTE

The distribution behavior outlined above can be quite surprising when compared with how a regular JMS queue behaves. In that model, messages sent on a queue would be shared evenly between the two consumers.

Most often, when we create multiple consumer instances we do so in order to process messages in parallel, either to increase the rate of consumption or the resiliency of consumer process. Since only one consumer instance can read at a time from a partition, how is this achieved in Kafka?

One way to do this is to use a single consumer instance to consume all of the messages and hand these to a pool of threads. While this approach increases processing throughput, it adds to the complexity of the consumer logic and does nothing to assist in the resiliency of the consuming system. If the single consumer instance goes offline, due to a power failure or similar event, then consumption stops.

The canonical way to address this problem in Kafka is to use more partitions.

Partitioning

Partitions are the primary mechanism for parallelizing consumption and scaling a topic beyond the throughput limits of a single broker instance. To get a better understanding of this, let's now consider the situation where there exists a topic with two partitions, and a single consumer subscribes to that topic (Figure 3-5).

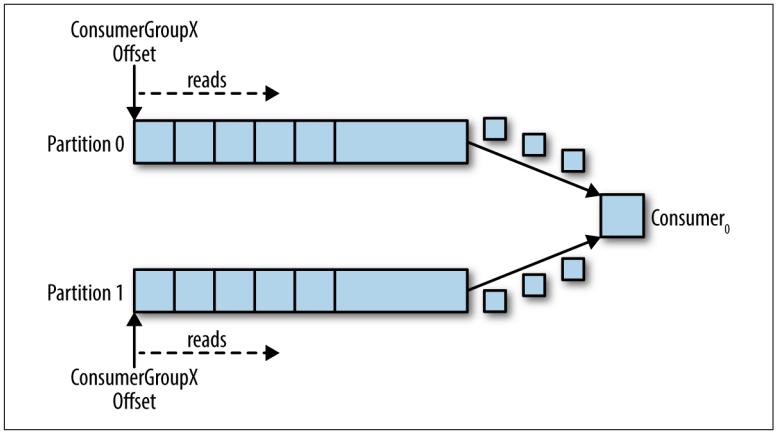

Figure 3-5. One consumer reading from multiple partitions

In this scenario, the consumer is assigned control of the pointers corresponding to its `group_id` in both partitions, and proceeds to consume messages from both.

When an additional consumer is added to this topic for the same `group_id`, Kafka will reallocate one of the partitions from the first to the second consumer. Each consumer instance will then consume from a single topic partition (Figure 3-6).

In order to enable parallel processing of messages in parallel from 20 threads, you would therefore need a *minimum* of 20 partitions. Any fewer partitions, and you would be left with consumers that are not allocated anything to work on, as described in our exclusive-consumer discussion earlier.

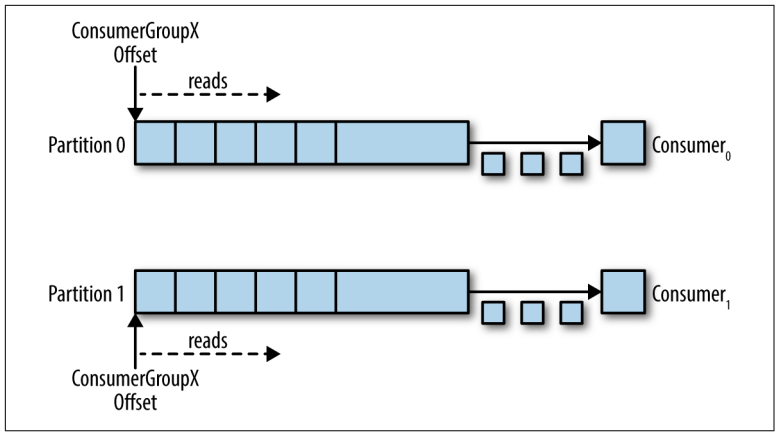

Figure 3-6. Two consumers in the same group reading from different partitions

This scheme greatly reduces the complexity of a Kafka broker's work as compared with the message distribution required to support a JMS queue. There is no bookkeeping of:

- Which consumer should get the next message based on round-robin distribution, the current capacity of prefetch buffers, or previous messages (as per-JMS message groups).
- Which messages have gone to which consumers and need to be redelivered in the event of failure.

All that a Kafka broker has to do is sequentially feed messages to a consumer as the latter asks for them.

However, the requirements for parallelizing consumption and replaying failed messages do not go away—the responsibility for them is simply transferred from the broker to the client. This means that they need to be considered within your code.

Sending Messages

The responsibility for deciding which partition to send a message to is assigned to the producer of that message. To understand the mechanism by which this is done, we first need to consider what it is that we are actually sending.

While in JMS, we make use of a message construct with metadata (headers and properties) and a body containing the payload; in Kafka the message is a *key-value pair*. The payload of the message is sent as the value. The key, on the other hand, is used primarily for partitioning purposes and should contain a *business-specific key* in order to place related messages on the same partition.

In Chapter 2 we discussed a use case from online betting where related events need to be processed in order by a single consumer:

1. A user account is set up.
2. Money is deposited into the account.
3. A bet is placed that withdraws money from the account.

If each event is a message sent to a topic, then the natural key in this case would be the account ID.

When a message is sent using the Kafka Producer API, it is handed to a partitioning function that, given the message and the current state of the Kafka cluster, returns a partition ID to which the message should be sent. This function is implemented through the `Partitioner` interface in Java.

This interface looks as follows:

```
interface Partitioner {
    int partition(String topic,
            Object key,   byte[] keyBytes,
            Object value, byte[] valueBytes,
            Cluster cluster);
}
```

The default implementation of the Partitioner uses a general-purpose hashing algorithm over the key, or round-robin if no key is provided, to determine the partition. This default works well in the majority of cases. There will, however, be times when you want to write your own.

Writing Your Own Partitioning Strategy

Let's consider an example where you want to send metadata along with a message payload. The payload is an instruction to perform a deposit into a betting account. A deposit instruction is something that we would like to ensure is not modified in transit, and want the confidence that only trusted upstream systems can initiate the

instruction. In such a case, the sending and receiving systems agree to use a signature to verify the authenticity of the message.

In regular JMS, we would simply define a `signature` message property and append it to the message. However, Kafka does not provide us with a mechanism to transmit metadata—only a key and a value.

Since the value is the bank transfer payload whose integrity we want to preserve, we are left with no alternative other than defining a data structure for use within the key. Assuming that we need an account ID for partitioning, as all messages relating to an account must be processed in order, we come up with the following JSON structure:

```
{
    "signature": "541661622185851c248b41bf0cea7ad0",
    "accountId": "10007865234"
}
```

As the `signature` value is going to vary on a per-payload basis, the default hashing `Partitioner` strategy is not going to reliably group related messages. We would therefore need to write our own strategy that will parse this key and partition on the `accountId` value.

NOTE

Kafka includes checksums to detect corruption of messages in storage and has a comprehensive set of security features. Even so, industry-specific requirements such as the one above do occasionally appear.

A custom partitioning strategy needs to ensure that all related messages end up in the same partition. While this seems straightforward, the requirement may be complicated by the importance of the ordering of related messages and how fixed the number of partitions in a topic is.

The number of partitions in a topic can change over time, as they can be added to if the traffic goes beyond what was initially expected. As such, message keys may need to be associated with the partition that they were initially sent to, implying a piece of state that needs to be shared across producer instances.

Another factor to consider is the evenness of the distribution of messages among partitions. Typically, keys are not evenly distributed across messages, and hash functions are not guaranteed to distribute messages fairly for a small set of keys.

It is important to note that however you decide to partition the messages, the partitioner itself may need to be reused.

Consider the requirement to replicate data between Kafka clusters in different geographical locations. Kafka comes with a standalone command-line tool called MirrorMaker for this purpose, used to consume messages from one cluster and produce them into another.

MirrorMaker needs to understand the keys of the topic being replicated in order to maintain relative ordering between messages as it replicates between clusters, as the number of partitions for that topic may not be the same in the two clusters.

Custom partitioning strategies are relatively rare, as the defaults of hashing or round-robin work for the majority of use cases. If, however, you require strong ordering guarantees or need to move metadata outside of payloads, then partitioning is something that you will need to consider in closer detail.

Kafka's scalability and performance benefits come from moving some of the responsibilities of a traditional broker onto the client. In this case, the decision around distribution of potentially related messages to multiple consumers running in parallel.

NOTE

JMS-based brokers also need to deal with these sorts of requirements. Interestingly, the mechanism of sending related messages to the same consumer—implemented via JMS Message Groups, a form of sticky load-balancing—also requires the sender to mark messages as related. In the JMS case, the broker is responsible for sending that group of related messages to a single consumer out of many, and transferring the ownership of the group if the consumer goes down.

Producer Considerations

Partitioning is not the only thing that needs to be considered when sending messages. Let us consider the `send()` methods on the `Producer` class in the Java API:

```
Future<RecordMetadata> send(ProducerRecord<K,V> record);
Future<RecordMetadata> send(ProducerRecord<K,V> record,
                            Callback callback);
```

The immediate thing to note is that both methods return a `Future`, which indicates that the send operation is not performed immediately. What happens is that the message (`ProducerRecord`) is written into a send buffer for each active partition and transmitted on to the broker by a background thread within the Kafka client library. While this makes the operation incredibly fast, it does mean that a naively written application could lose messages if its process goes down.

As always there is a way of making the sending operation more reliable at the cost of performance. The size of this buffer can be tuned to be 0, and the sending application thread can be forced to wait until the transmission of the message to the broker has been completed, as follows:

```
RecordMetadata metadata = producer.send(record).get();
```

Consumption Revisited

The consumption of messages has additional complexities that need to be reasoned about. Unlike the JMS API, which can trigger a message listener in reaction to the arrival of a message, Kafka's `Consumer` interface is polling only. Let's take a closer look at the `poll()` method used for this purpose:

```
ConsumerRecords<K,V> poll(long timeout);
```

The method's return value is a container structure containing multiple `ConsumerRecord` objects from potentially multiple partitions. The `ConsumerRecord` itself is a holder object for a key-value pair, with associated metadata such as which partition it came from.

As discussed in Chapter 2, we need to constantly keep in mind what happens to messages once they are either successfully processed or not, such as if the client is unable to process a message or if it terminates. In JMS, this was handled through the acknowledgement mode. The broker would either delete a successfully processed message or redeliver an unprocessed or failed one (assuming transactions were in play). Kafka works quite differently. Messages are not deleted on the broker once consumed, and the responsibility for working out what happens on failure lies with the consuming code itself.

As we have discussed, a consumer group is associated with an offset in the log. The position in the log associated with that offset corresponds to the next message to be handed out in response to a `poll()`. What is critical in consumption is the timing around when that offset is incremented.

Going back to the consumption model discussed earlier, the processing of a message has three phases:

1. Fetch a message for consumption.
2. Process the message.
3. Acknowledge the message.

The Kafka consumer comes with the configuration option `enable.auto.commit`. This is a commonly used default setting, as is usually the case with settings containing the word "auto."

Before Kafka 0.10, a client using this setting would send the offset of the last consumed message on the next `poll()` after processing. This meant that any messages that were fetched ran the possibility of being reprocessed if the client processed the messages, but terminated unexpectedly before calling `poll()`. As the broker does not keep any state around how many times a message is consumed, the next consumer to fetch that same message would have no idea that anything untoward had happened. This behavior was pseudo-transactional; an offset was committed only if the message processing was successful, but if the client terminated, the broker would send the same message again to another client. This behavior corresponded an *at-least-once* message delivery guarantee.

In Kafka 0.10, the client code was changed so that the commit became something that the client library would trigger periodically, as defined by the `auto.commit.interval.ms` setting. This behavior sits somewhere between the JMS `AUTO_ACKNOWLEDGE` and `DUPS_OK_ACKNOWLEDGE` modes. When using auto commit, messages could be acknowledged regardless of whether they were actually processed—this might occur in the instance of a slow consumer. If the consumer terminated, messages would be fetched by the next consumer from the committed position, leading to the possibility of missing messages. In this instance, Kafka didn't lose messages, the consuming code just didn't process them.

This mode also has the same potential as in 0.9: messages may have been processed, but in the event of failure, the offset may not have been committed, leading to potential duplicate delivery. The more messages you retrieve during a `poll()`, the greater this problem is.

As discussed in "Consuming Messages from a Queue" on page 21, there is no such thing as once-only message delivery in a messaging system once you take failure modes into account.

In Kafka, there are two ways to commit the offset: automatically, and manually. In both cases, you may process the messages multiple times if you have processed the message but failed before committing. You may also not process the message at all if the commit happened in the background and your code terminated before it got around to processing (a possibility in Kafka 0.9 and earlier).

Controlling the offset commit process manually is enabled in the Kafka consumer API by setting the `enable.auto.commit` to `false`, and calling one of the following methods explicitly:

```
void commitSync();
void commitAsync();
```

If you care about at-least-once processing, you would commit the offset manually via `commitSync()`, executed immediately after your processing of messages.

These methods prevent messages from being acknowledged before being processed, but do nothing to address the potential for duplicate processing, while at the same time giving the impression of transactionality. Kafka is nontransactional. There is no way for a client to:

- Roll back a failed message automatically. Consumers must deal with exceptions thrown due to bad payloads and backend outages themselves, as they cannot rely on the broker redelivering messages.
- Send messages to multiple topics within the same atomic operation. As we will soon see, control of different topics and partitions may lie with many different machines in a Kafka cluster that do not coordinate transactions on sending. There is some work going on at the time of writing to enable this via KIP-98.
- Tie the consumption of a message from one topic to the sending of another message on another topic. Again, Kafka's architec-

ture depends on many independent machines working as a single bus, and no attempt is made to hide this. For instance, there is no API construct that would enable the tying together of a `Consumer` and `Producer` in a transaction; in JMS this is mediated by the `Session` from which `MessageProducers` and `MessageConsumers` are created.

So if we cannot rely on transactions, how do we ensure semantics closer to those provided by a traditional messaging system?

If there is a possibility that the consumer's offset is incremented before the message has been processed, such as during a consumer crash, then there is no way for a consumer to know if its consumer group has missed messages when it is assigned a partition. As such, one strategy is to rewind the offset to a previous position. The Kafka Consumer API provides the following methods to enable this:

```
void seek(TopicPartition partition, long offset);
void seekToBeginning(Collection<TopicPartition> partitions);
```

The `seek()` method can be used with the `offsetsForTimes(Map<TopicPartition,Long> timestampsToSearch)` method to rewind to a state at some specific point in the past.

Implicity, using this approach means that it is highly likely that some messages that were previously processed will be consumed and processed all over again. To get around this we can use idempotent consumption, as described in Chapter 4, to keep track of previously seen messages and discard duplicates.

Alternatively, your consumer code can be simple where some message loss or duplication is acceptable. When we consider the sorts of use cases that Kafka is typically used for, namely movement of log events, metrics, tracking clicks, and so on, we find that the loss of individual messages is not likely to have a meaningful impact on the surrounding applications. In such cases, defaults may be perfectly acceptable. On the other hand, if your application needs to transmit payments, you may care deeply about each individual message. It all comes down to context.

As a personal observation, as the rate of messages rises, the less valuable any individual message is going to be. For high-volume messages, their value typically lies in considering them in an aggregate form.

High Availability

Kafka's approach to high availability is significantly different from that of ActiveMQ. Kafka is designed around horizontally scalable clusters in which all broker instances accept and distribute messages at the same time.

A Kafka cluster is made up of multiple broker instances running on separate servers. Kafka has been designed to run on commodity standalone hardware, with each node having its own dedicated storage. The use of storage area networks (SANs) is discouraged as many compute nodes may compete for time slices of the storage and create contention.

Kafka is an *always-on* system. A lot of large Kafka users never take their clusters down, and the software always provides an upgrade path via a rolling restart. This is managed by guaranteeing compatibility with the previous version for messages and inter-broker communication.

Brokers are connected to a cluster of ZooKeeper servers which acts as a registry of configuration information and is used to coordinate the roles of each broker. ZooKeeper is itself a distributed system, that provides high availability through replication of information in a *quorum* setup.

At its most basic, a topic is created on the Kafka cluster with the following properties:

- Number of partitions. As discussed earlier, the exact value used here depends upon the desired degree of parallel consumption.
- Replication factor. This defines how many broker instances in the cluster should contain the logs for this partition.

Using ZooKeepers for coordination, Kafka attempts to fairly distribute new partitions among the brokers in the cluster. This is performed by one instance that serves the role of a Controller.

At runtime, *for each topic partition*, the Controller assigns the roles of a *leader* (master) and *followers* (slaves) to the brokers. The broker acting as leader for a given partition is responsible for accepting all messages sent to it by producers, and distributing messages to consumers. As messages are sent into a topic partition, they are replicated to all broker nodes acting as followers for that partition. Each

node that contains the logs for the partition is refered to as a *replica*. A broker may act as a leader for some partitions and as a follower for others.

A follower that contains all of the messages held by the leader is referred to as being an *in-sync replica*. Should the broker acting as leader for a partition go offline, any broker that is up-to-date or in-sync for that partition may take over as leader. This is an incredibly resilient design.

Part of the producer's configuration is an `acks` setting, which will dictate how long many replicas must acknowledge receipt of the message before the application thread continues when it is sent: `0`, `1`, or `all`. If set to `all`, on receipt of a message, the leader will send confirmation back to the producer once it has received acknowledgements of the write from a number of replicas (including itself), defined by the topic's `min.insync.replicas` setting (`1` by default). If a message cannot be successfully replicated, then the producer will raise an exception to the application (`NotEnoughReplicas` or `NotEnoughReplicasAfterAppend`).

A typical configuration is to create a topic with a replication factor of 3 (1 leader, 2 followers for each partition) and set `min.insync.replicas` to 2. That way the cluster will tolerate one of the brokers managing a topic partition going offline with no impact on client applications.

This brings us back to the now-familiar performance versus reliability trade-off. Replication comes at the cost of additional time waiting for acknowledgments from followers; although as it is performed in parallel, replication to a minimum of three nodes has similar performance as that of two (ignoring the increased network bandwidth usage).

Using this replication scheme, Kafka cleverly avoids the need to ensure that every message is physically written to disk via a `sync()` operation. Each message sent by a producer will be written to the partition's log, but as discussed in Chapter 2, a write to a file is initially performed into an operating system buffer. If that message is replicated to another Kafka instance and resides in its memory, loss of the leader does not mean that the message itself was lost—the in-sync replica can take over.

Avoiding the need to `sync()` means that Kafka can accept messages at the rate at which it can write into memory. Conversely, the longer it can avoid flushing its memory to disk, the better. For this reason it is not unusual to see Kafka brokers assigned 64 GB of memory or more. This use of memory means that a single Kafka instance can easily operate at speeds many thousands of times faster than a traditional message broker.

Kafka can also be configured to `sync()` batches of messages. As everything in Kafka is geared around batching, this actually performs quite well for many use cases and is a useful tool for users that require very strong guarantees. Much of Kafka's raw performance comes from messages that are sent to the broker as batches, and from having those messages read from the broker in sequential blocks via zero-copy. The latter is a big win from a performance and resource perspective, and is only possible due to the use of the underlying journal data structure, which is laid out per partition.

Much higher performance is possible across a Kafka cluster than through the use of a single Kafka broker, as a topic's partitions may be horizontally scaled over many separate machines.

Summary

In this chapter we looked at how Kafka's architecture reimagines the relationship between clients and brokers to provide an incredibly resilient messaging pipeline with many times greater throughput than a regular message broker. We discussed the functionality that it trades off to achieve this, and examined in brief the application architectures that it enables. In the next chapter we will look at common concerns that messaging-based applications need to deal with and discuss strategies for addressing them. We will complete the chapter by outlining how to reason about messaging technologies in general so that you can evaluate their suitability to your use cases.

CHAPTER 4

Messaging Considerations and Patterns

In the previous chapters we explored two very different approaches to broker-based messaging and how they handle concerns such as throughput, high availability and transactionality. In this chapter we are going to focus on the other half of a messaging system—the client logic. We will discuss some of the mechanical complexities that message-based systems need to address and some patterns that can be applied to deal with them. To round out the discussion, we will discuss how to reason about messaging products in general and how they apply to your application logic.

Dealing with Failure

A system based on messaging must be able to deal with failures in a graceful way. Failures come in many forms at various parts of the application.

Failures When Sending

When producing messages, the primary failure that needs to be considered is a broker outage. Client libraries usually provide logic around reconnection and resending of unacknowledged messages in the event that the broker becomes unavailable.

Reconnection involves cycling through the set of known addresses for a broker, with delays in-between. The exact details vary between client libraries.

While a broker is unavailable, the application thread performing the send may be blocked from performing any additional work if the send operation is synchronous. This can be problematic if that thread is reacting to outside stimuli, such as responding to a web service request.

If all of the threads in a web server's thread pool are suspended while they are attempting to communicate with a broker, the server will begin rejecting requests back to upstream systems, typically with `HTTP 503 Service Unavailable`. This situation is referred to as *back-pressure*, and is nontrivial to address.

One possibility for ensuring that an unreachable broker does not exhaust an application's resource pool is to implement the circuit breaker pattern around messaging code (Figure 4-1). At a high level, a circuit breaker is a piece of logic around a method call that is used to prevent threads from accessing a remote resource, such as a broker, in response to application-defined exceptions.

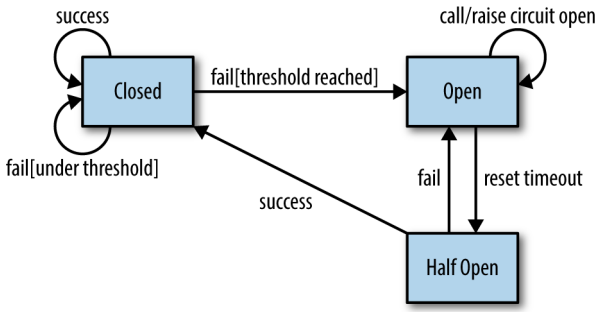

Figure 4-1. Circuit breaker

A circuit breaker has three states:

Closed
: Traffic calls are routed into the method as normal.

Open
: An error state around the resource has been detected; subsequent method calls are routed to alternative logic, such as the formatting of an immediate error message.

Half-Open
: Triggered after a period of time to allow threads to retest the protected reource. From here, the circuit breaker will either be closed, allowing application logic to continue as normal, or reopened.

Circuit breaker implementations vary in terms of functionality and can take into consideration concerns such as timeouts and error thresholds. In order for message sends to work correctly with circuit breakers, the client library must be configured to throw an exception at some point if it cannot send. It should not attempt to reconnect infinitely.

Asynchronous sends, such as those performed by Kafka, are not in themselves a workaround for this issue. An asynchronous send will involve messages being placed into a finite in-memory buffer that is periodically sent by a background thread to the broker. In the case of Kafka's client library, if this buffer becomes full before the client is able to reconnect, then any subsequent sends will be blocked (i.e., become synchronous). At this time the application thread will wait for some period until eventually the operation is abandoned and a `TimeoutException` is thrown. At best, asynchronous sends will delay the exhaustion of an application's thread pool.

Failures When Consuming

In consumption there are two different types of failures:

Permanent
: The message that you are consuming will never be able to be processed.

Temporary
: The message would normally be processed, but this is not possible at this time.

Permanent failures

Permanent failures are usually caused by an incorrectly formed payload or, less commonly, by an illegal state within the entity that a message refers to (e.g., the betting account that a withdrawal is coming from being suspended or canceled). In both cases, the failure is related to the application, and if at all possible, this is where it should be handled. Where this is not possible, the client will often fall back to broker redelivery.

JMS-based message brokers provide a redelivery mechanism that is used with transactions. Here, messages are redispatched for processing by the consumer when an exception is thrown. When messages are redelivered, the broker keeps track of this by updating two message headers:

- `JMSRedelivered` set to `true` to indicate redelivery
- `JMSXDeliveryCount` incremented with each delivery

Once the delivery count exceeds a preconfigured threshold, the message is sent to a *dead-letter queue* or DLQ. DLQs have a tendency to be used as a dumping ground in most message-based systems and are rarely given much thought. If left unconsumed, these queues can prevent the cleanup of journals and ultimately cause brokers to run out of disk space.

So what should you do with these queues? Messages from a DLQ can be quite valuable as they may indicate corner cases that your application had not considered or actions requiring human intervention to correct. As such they should be drained to either a log file or some form of database for periodic inspection.

As previously discussed, Kafka provides no mechanism for transactional consumption and therefore no built-in mechanism for message redelivery on error. It is the responsibility of your client code to provide redelivery logic and send messages to dead-letter topics if needed.

Temporary failures

Temporary failures in message consumption fall into one of two categories:

Global
: Affecting all messages. This includes situations such as a consumer's backend system being unavailable.

Local
: The current message cannot be processed, but other messages on the queue can. An example of this is a database record relating to a message being locked and therefore temporarily not being updateable.

Failures of this type are by their nature transient and will likely correct themselves over time. As such, the way they need to be handled in significantly different ways to handle permanent failures. A message that cannot be processed now is not necessarily illegitimate and should not end up on a DLQ. Going back to our deposit example, not being able to credit a payment to an account does not mean that the payment should just be ignored.

There are a couple of options that you may want to consider on a case by case basis, depending on the capabilities of your messaging system:

- If the problem is *local*, perform retries within the message consumer itself until the situation corrects itself. There will always be a point at which you give up. Consider escalating the message for human intervention within the application itself.
- If the situation is *global*, then relying on a redelivery mechanism that eventually pushes messages into a DLQ will result in a succession of perfectly-legitimate messages being drained from the source queue and effectively being discarded. In production systems, this type of situation is characterized by DLQs accumulating messages in bursts. One solution to this situation is to turn off consumption altogether until the situation is rectified through the use of the Kill Switch pattern (Figure 4-2).

A Kill Switch operates by catching exceptions related to transient issues and pausing consumption. The message currently being processed should either be rolled back if using a transaction, or held

onto by the consuming thread. In both cases, it should be possible to reprocess the message later.

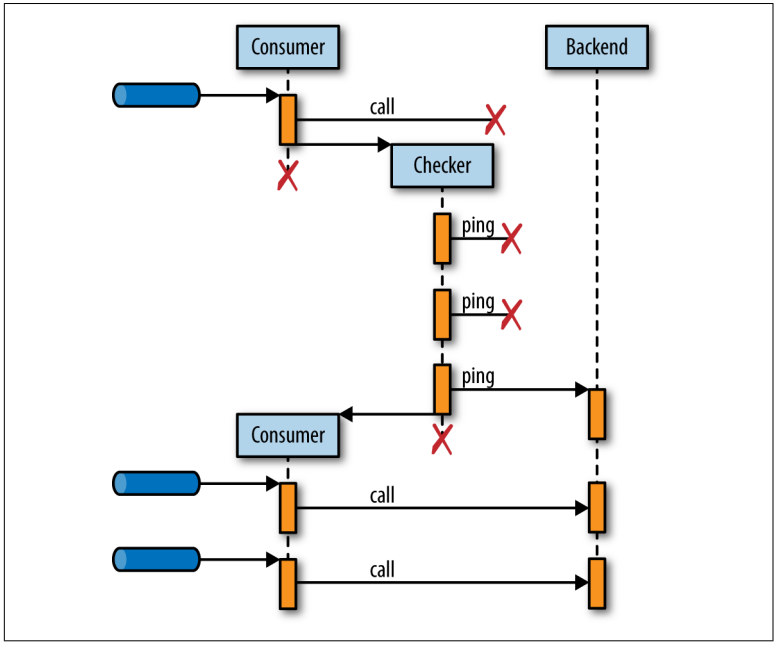

Figure 4-2. Kill Switch sequence diagram

The consumer should trigger a background checker task to periodically determine whether the issue has gone away. If the issue is a web service outage, the check might be a poll of a URL that simply acknowledges that the service is up. If the issue is a database outage, then the check might consist of a dummy SQL query being run (e.g., `SELECT 1 FROM DUAL` on Oracle). If the check operation succeeds, then the checker task reactivates the message consumer and terminates itself.

Preventing Duplicate Messages with Idempotent Consumption

Previously we discussed that systems based on queues must deal with the possibility of duplicate messages. In the event of a consumer system going offline unexpectedly, there may be a situation where messages were processed but had not yet been acknowledged. This applies regardless of whether you are using a transaction-capable broker and did not yet commit, or in the case of Kafka did not move the consumed offset forward. In both cases, when the client is restarted, these unacknowledged messages will be reprocessed.

Duplication may also occur when a system that is upstream of a broker reissues the same payloads. Consider the scenario where a system has its inputs reloaded into it after an outage involving data loss. The replay of the data into the system causes a side effect of sending messages into a queue. The messages are technically different from those that were sent in the past (they have different message IDs or offsets), but they trigger the same consumer logic multiple times.

To avoid processing the message multiple times, the consumption logic needs to be made *idempotent*. The Idempotent Consumer pattern acts like a stateful filter that allows logic wrapped by it to be executed only once. Two elements are needed to implement this:

- A way to uniquely identify each message by a business key.
- A place to store previously seen keys. This is referred to as an idempotent repository. Idempotent repositories are containers for a durable set of keys that will survive restarts of the consumer and can be implemented in database tables, journals, or similar.

Consider the following JSON message which credits an account:

```
{
    "timestamp" : "20170206150030",
    "accountId" : "100035765",
    "amount" : "100"
}
```

When a message arrives, the consumer needs to uniquely identify it. As discussed earlier, built-in surrogate keys such as a message ID or

offset are not adequate to protect from upstream replays. In the message above, a good candidate for this key is a combination of the `timestamp` and `account` fields of the message, as it is unlikely that two deposits for the same account happen at the same time.

The idempotent repository is checked to see whether it contains the key, and if it does not, the logic wrapped by it is executed, otherwise it is skipped. They key is stored in the idempotent repository according to one of two strategies:

Eagerly
: Before the wrapped logic is executed. In this case, the consumer needs to remove the key if the wrapped logic throws an error.

Lazily
: After the logic is executed. In this situation, you run the risk of duplicate processing if the key is not stored due to a system crash.

In addition to timings, when developing idempotent repositories you need to be aware that they may be accessed by multiple consumer instances at the same time.

NOTE

The Apache Camel project is a Java-based integration framework that includes an implementation of numerous integration patterns, including the Idempotent Consumer. The project's documentation provides a good starting point for implementing this pattern in other environments. It includes many idempotent repository implementations for storing keys in files, databases, in-memory data grids, and even Kafka topics.

What to Consider When Looking at Messaging Technologies

Message brokers are a tool, and you should aim to use the right one for the job. As with any technology, it is difficult to make objective decisions unless you know what questions to ask. Your choice of messaging technology must first and foremost be led by your use cases.

What sort of a system are you building? Is it message-driven, with clear relationships between producers and consumers, or event-

driven where consumers subscribe to streams of events? A basic queue-based system is enough for the former, while there are numerous options for the latter involving persistent and non-persistent messaging, the choice of which will depend on whether or not you care about missing messages.

If you need to persist messages, then you need to consider how that persistence is performed. What sorts of storage options does the product support? Do you need a shared filesystem, or a database? Your operating environment will feed back into your requirements. There is no point looking at a messaging system designed for independent commodity servers with dedicated disks if you are forced to use a storage area network (SAN).

Broker storage is closely related to the high availability mechanism. If you are targeting a cloud deployment, then highly available shared resources such as a network filesystem will likely not be available. Look at whether the messaging system supports native replication at the level of the broker or its storage engine, or whether it requires a third-party mechanism such as a replicated filesystem. High availability also needs to be considered over the entire software to hardware stack—a highly available broker is not really highly available if both master and slave can be taken offline by a filesystem or drive failure.

What sort of message ordering guarantees does your application require? If there is an ordering relationship between messages, then what sort of support does the system provide for sending these related messages to a single consumer?

Are certain consumers in your applications only interested in subsets of the overall message stream? Does the system support filtering messages for individual consumers, or do you need to build an external filter that drops unwanted messages from the stream?

Where is the system going to be deployed? Are you targeting a private data center, cloud, or a combination of the two? How many sites do you need to pass messages between? Is the flow unidirectional, in which case replication might be enough, or do you have more complex routing requirements? Does the messaging system support routing or store-and-forward networking? Is the replication handled by an external process? If so, how is that process made highly available?

For a long time, marketing in this field was driven by performance metrics, but what does it matter if a broker can push thousands of messages per second if your total load is going to be much lower than that? Get an understanding of what your real throughput is likely to be before being swayed by numbers. Large message volumes per day can translate to relatively small numbers per second. Consider your traffic profile—are the messages going to be a constant 24-hour stream, or will the bulk of traffic fall into a smaller timeframe? Average numbers are not particularly useful—you will get peaks and troughs. Consider how the system will behave on the largest volumes.

Perform load tests to get a good understanding of how the system will work with your use cases. A good load test should verify system performance with:

- Estimated message volumes and sizes—use sample payloads wherever possible.
- Expected number of producers, consumer, and destinations.
- The actual hardware that the system will run on. This is not always possible, but as we discussed, it will have a substantial impact on performance.

If you are intending to send large messages, check how the system deals with them. Do you need some form of additional external storage outside of the messaging system, such as when using the Claim Check pattern? Or is there some form of built-in support for streaming? If streaming very large content like video, do you need persistence at all?

Do you need low latency? If so, how low? Different business domains will have different views on this. Intermediary systems such as brokers add processing time between production and consumption—perhaps you should consider brokerless options such as ZeroMQ or an AMQP routing setup?

Consider the interaction between messaging counterparties. Are you going to be performing request-response over messaging? Does the messaging system support the sorts of constructs that are required, i.e., message headers, temporary destinations, and selectors?

Are you intending on spanning protocols? Do you have C++ servers that need to communcate with web clients? What support does the messaging system provide for this?

The list goes on…transactions, compression, encryption, duration of message storage (possibly impacted by local legislation), and commercial support options.

It might initially seem overwhelming, but if you start looking at the problem from the point of view of your use cases and deployment environment, the thought process behind this exercise will steer you naturally.

CHAPTER 5
Conclusion

In this book we examined two messaging technologies at a high level in order to better understand their general characteristics. This was by no means a comprehensive list of the pros and cons of each technology, but instead an exercise in understanding how the design choices of each one impact their feature sets, and an introduction into the high-level mechanics of message delivery.

ActiveMQ represents a classic broker-centric design that handles a lot of the complexity of message distribution on behalf of clients. It provides a relatively simple setup that works for a broad range of messaging use cases. In implementing the JMS API, it provides mechanisms such as transactions and message redelivery on failure. ActiveMQ is implemented through a set of Java libraries, and aside from providing a standalone broker distribution, can be embedded within any JVM process, such as an application server or IoT messaging gateway.

Kafka, on the other hand, is a distributed system. It provides a functionally simpler broker that can be horizontally scaled out, giving many orders of magnitude higher throughput. It provides massive performance at the same time as fault tolerance through the use of replication, avoiding the latency cost of synchronously writing each message to disk.

ActiveMQ is a technology that focuses on ephemeral movement of data—once consumed, messages are deleted. When used properly, the amount of storage used is low—queues consumed at the rate of message production will trend toward empty. This means much

smaller maximum disk requirements than Kafka—gigabytes versus terabytes.

Kafka's log-based design means that messages are not deleted when consumed, and as such can be processed many times. This enables a completely different category of applications to be built—ones which can consider the messaging layer as a source of historical data and can use it to build application state.

ActiveMQ's requirements lead to a design that is limited by the performance of its storage and relies on a high-availability mechanism requiring multiple servers, of which some are not in use while in slave mode. Where messages are physically located matters a lot more than it does in Kafka. To provide horizontal scalability, you need to wire brokers together into store-and-forward networks, then worry about which one is responsible for messages at any given point in time.

Kafka requires a much more involved system involving a ZooKeeper cluster and requires an understanding of how applications will make use of the system (e.g., how many consumers will exist on each topic) before it is configured. It relies upon the client code taking over the responsibility of guaranteeing ordering of related messages, and correct management of consumer group offsets while dealing with messages failures.

Do not believe the myth of a magical messaging fabric—a system that will solve all problems in all operating environments. As with any technology area there are trade-offs, even within messaging systems in the same general category. These trade-offs will quite often impact how your applications are designed and written.

Your choices in this area should be led first and foremost by a good understanding of your own use cases, desired design outcomes, and target operating environment. Spend some time looking into the details of a messaging product before jumping in. Ask questions:

- How does this system distribute messages?
- What are its connectivity options?
- How can it be made highly available?
- How do you monitor and maintain it?
- What logic needs to be implemented within my application?

I hope that this book has given you an appreciation of some of the mechanics and trade-offs of broker-based messaging systems, and will help you to consider these products in an informed way. Happy messaging!

About the Author

Jakub Korab is a UK-based specialist messaging and integration consultant who runs his own consultancy, Ameliant. Over the past six years he has worked with over 100 clients around the world to design, develop, and troubleshoot large, multisystem integrations and messaging installations using a set of open source tools from the Apache Software Foundation. His experience has spanned industries including finance, shipping, logistics, aviation, industrial IoT, and space exploration. He is coauthor of the *Apache Camel Developer's Cookbook* (Packt, 2013), and an international speaker who has presented at conferences across Europe, including Devoxx UK, JavaZone (Norway), Voxxed Days (Bristol, Belgrade, and Ticino), DevWeek, and O'Reilly SACon. Prior to going independent, he worked for the company behind ActiveMQ—FuseSource, later acquired by RedHat—and is currently partnering with Confluent, the company behind Kafka.