目錄

Apache Kafka 官方文档中文版

Apache Kafka是一个高吞吐量分布式消息系统。

Kafka在国内很多公司都有大规模的应用，但关于它的中文资料并不多，只找到了12年某版本的设计章节的翻译。

为了方便大家学习交流，尽最大努力翻译一下完整的官方的手册。

原文版本选择当前最新的Kafka 0.10.0的文档（2016-08）。

前辈们在OS China上翻译的设计章节非常优秀，如果之前没有阅读过推荐先参考一下。

翻译中

发现有小伙伴已经下载了，如果你发现后面还是英文不是我掺假是你着急了！

最近国内 Gitbook 比较不稳定，可以使用 Github Pages 地址阅读

Github Pages：Apache Kafka 官方文档中文版

源文档地址：Kafka 0.10.0 Documentation

Github: BeanMr/apache-kafka-documentation-cn

Gitbook下载: Apache Kafka Documentation CN

译者：@D2Feng @Ein Verne

翻译采用章节中英文对照的形式进行，未翻译的章节保持原文。

译文章节组织及内容尽量保持与原文一直，但有时某些句子直译会有些蹩脚，所以可能会进行一些语句上调整。

因为本人能力和精力有限，译文如有不妥欢迎提issue，更期望大家能共同参与进来。

参与翻译Pull Request流程

小伙伴@numbbbbb在《The Swift Programming Language》对协作流程中进行了详细的介绍，小伙伴@looly在他的ES翻译中总结了一下，我抄过来并再次感谢他们的分享。

1. 首先fork的项目apache-kafka-documentation-cn到你自己的Github

2. 把fork过去的项目也就是你的项目clone到你的本地
3. 运行 `git remote add ddfeng` 把我的库添加为远端库
4. 运行 `git pull ddfeng master` 拉取并合并到本地
5. 翻译内容或者更正之前的内容。
6. commit后push到自己的库（ `git push origin master` ）
7. 登录Github在你首页可以看到一个 `pull request` 按钮，点击它，填写一些说明信息，然后提交即可。

1~3是初始化操作，执行一次即可。

在提交前请先执行第4步同步库，这样可以及时发现和避免冲突，然后执行5~7既可。

如果嫌以上过程繁琐，你只是准备指出一些不当翻译，也可以点击段落后'+'直接评论。

小伙伴 @Ein Verne 建议统一中文排版并提供了参考规范，希望大家尽量采纳。

JustDoIT，您的任何建议和尝试都值得尊重！*

因为在我天朝Gitbook比Github还不稳定，所以本人做了个 Hook 来将 Gitbook 生成的内容发布到 Github Pages 上。

在发布的过程会保留 /docs 目录下的 CNAME 文件，如果你要发布到自己的 Github Pages 请注意修改。

术语表

排序暂时按照文章中出现的顺序

英文	翻译
logs	日志
topic	主题
partition	分区
segment	日志段
offset	偏移量
Geo-Replication	
producers	生产者
consumer	消费者
broker	消息代理？
Multi-tenancy	多租户
Metrics	
pipeline	管线、管道
Replicated Logs	副本日志
In-Sync Replicas	同步副本
LEO	日志文件最近偏移
HW	
rebalance	
registry	注册表
rack	

1.1 简介

Kafka 是一个实现了分布式、分区、提交后复制的日志服务。它通过一套独特的设计提供了消息系统中间件的功能。

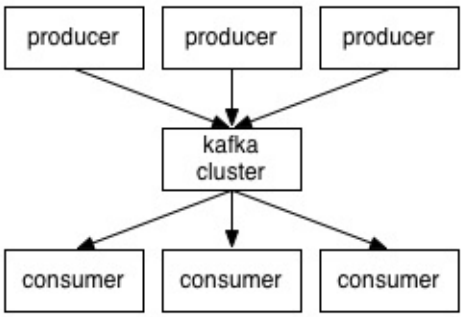
这是什么意思呢？

首先我们回顾几个基础的消息系统术语：

- Kafka 将消息源放在称为*topics*的归类组维护
- 我们将发布消息到 Kafka topic 上的处理程序称之为*producers*
- 我们将订阅 topic 并处理消息源发布的信息的程序称之为*consumers*
- Kafka 采用集群方式运行，集群由一台或者多台服务器构成，每个机器被称之为一个*broker*

（译者注：这些基本名词怎么翻译都觉着怪还是尽量理解下原文）

所以高度概括起来，producers（生产者）通过网络将 messages（消息）发送到 Kafka 机器，然后由集群将这些消息提供给 consumers（消费者），如下图所示：

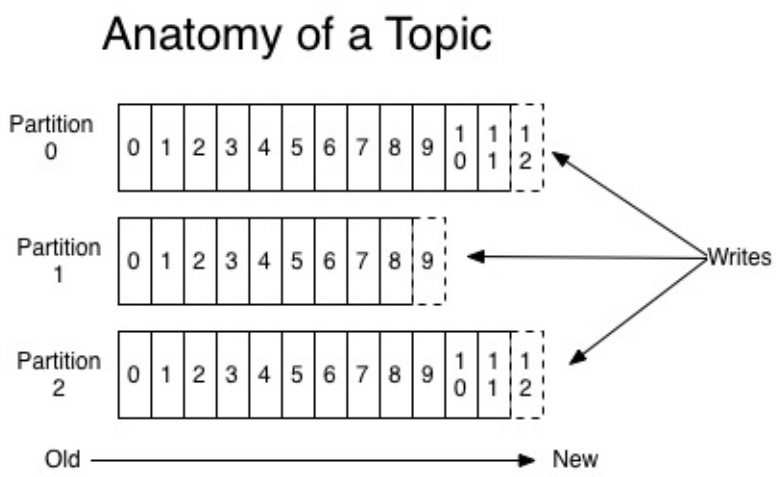


Clients（客户端）和 Servers（服务器）通过一个简单的、高效的基于 TCP 的协议 进行交互。官方为 Kafka 提供一个 Java 客户端，但更多其他语言的客户端 可以在这里找到。

Topics and Logs

Let's first dive into the high-level abstraction Kafka provides—the topic. 首先我们先来深入 Kafka 提供的关于 Topic 的高层抽象。

Topic 是一个消息投递目标的名称，这个目标可以理解为一个消息归类或者消息源。对于每个 Topic，Kafka 会为其维护一个如下图所示的分区的日志文件：



每个 partition（分区）是一个有序的、不可修改的、消息组成的队列；这些消息是被不断的 appended（追加）到这个 commit log（提交日志文件）上的。在这些 patitions 之中的每个消息都会被赋予一个叫做*offset*的顺序 id 编号，用来在 partition 之中唯一性的标示这个消息。

Kafka 集群会保存一个时间段内所有被发布出来的信息，无论这个消息是否已经被消费过，这个时间段是可以配置的。比如日志保存时间段被设置为 2 天，那么 2 天以内发布的消息都是可以消费的；而之前的消息为了释放空间将会抛弃掉。Kafka 的性能与数据量不相干，所以保存大量的消息数据不会造成性能问题。

实际上 Kafka 关注的关于每个消费者的元数据信息也基本上仅仅只有这个消费者的"offset"也就是它访问到了 log 的哪个位置。这个 offset 是由消费者控制的，通常情况下当消费者读取信息时这个数值是线性递增的，但实际上消费者可以自行随意的控制这个值从而随意控制其消费信息的顺序。例如，一个消费者可以将其重置到更早的时间来实现信息的重新处理。

这些特性组合起来就意味着 Kafka 消费者是非常低消耗，它们可以随意的被添加或者移除而不会对集群或者其他的消费者造成太多的干扰。例如，你可以通过我们的命令行工具"tail"（译者注：Linux 的 tail 命令的意思）任何消息队列的内容，这不会对任何已有的消费者产生任何影响。

对 log 进行分区主要是为了以下几个目的：第一、这可以让 log 的伸缩能力超过单台服务器上线，每个独立的 partition 的大小受限于单台服务器的容积，但是一个 topic 可以有很多 partition 从而使得它有能力处理任意大小的数据。第二、在并行处理方面这可以作为一个独立的单元。

分布式

log 的 partition 被分布到 Kafka 集群之中；每个服务器负责处理彼此共享的 partition 的一部分数据和请求。每个 partition 被复制成指定的份数散布到机器之中来提供故障转移能力。

对于每一个 partition 都会有一个服务器作为它的"leader"并且有零个或者多个服务器作为"followers"。leader 服务器负责处理关于这个 partition 所有的读写请求，followers 服务器则被动的复制 leader 服务器。如果有 leader 服务器失效，那么 followers 服务器将有一台被自动选举成为新的 leader。每个服务器作为某些 partition 的 leader 的同时也作为其它服务器的 follower，从而实现了集群的负载均衡。

Producers

生产者将数据发布到它们选定的 topics 上。生产者负责决定哪个消息发送到 topic 的哪个 partition 上。这可以通过简单的轮询策略来实现从而实现负载均衡，也可以通过某种语义分区功能实现（基于某个消息的某个键）。关于 partition 功能的应用将在后文进一步介绍。

Consumers

通常消息通信有两种模式：队列模式和订阅模式。在队列模式 中一组消费者可能是从一个服务器读取消息，每个消息被发送给其中一个消费者。在订阅模式，消息被广播给所有的消费者。Kafka 提供了一个抽象，把*consumer group*的所有消费者视为同一个抽象的消费者。

每个消费者都有一个自己的消费组名称标示，每一个发布到 topic 上的消息会被投递到每个订阅了此 topic 的消费者组的某一个消费者（译者注：每组都会投递，但每组都只会投递一份到某个消费者）。这个被选中的消费者实例可以在不同的处理程序中或者不同的机器之上。

如果所有的消费者实例都有相同的消费组标示 (consumer group), 那么整个结构就是一个传统的消息队列模式，消费者之间负载均衡。

如果所有的消费者实例都采用不同的消费组，那么真个结构就是订阅模式，每一个消息将被广播给每一个消费者。

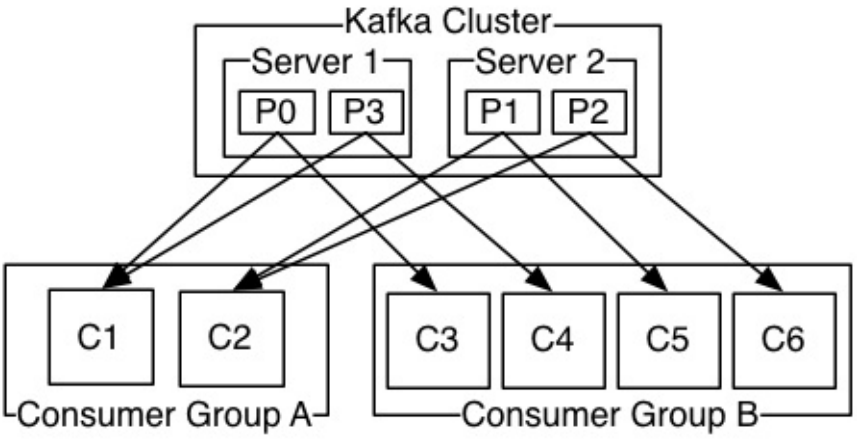
通常来说，我们发现在实际应用的场景，常常是一个 topic 有数量较少的几个消费组订阅，每个消费组都是一个逻辑上的订阅者。每个消费组由由很多消费者实例构成从而实现横向的扩展和故障转移。其实这也是一个消息订阅模式，无非是消费者不再是一个单独的处理程序而是一个消费者集群。

kafka 还提供了相比传统消息系统更加严格的消息顺序保证。

传统的消息队列在服务器上有序的保存消息，当有多个消费者的时候消息也是按序发送消息。但是因为消息投递到消费者的过程是异步的，所以消息到达消费者的顺序可能是乱序的。这就意味着在并行计算的场景下，消息的有序性已经丧失了。消息系统通常采用一个“排他消费者”的概念来规避这个问题，但这样就意味着失去了并行处理的能力。

Kafka 在这一点上做的更优秀。Kafka 有一个 Topic 中按照 partition 并行的概念，这使它即可以提供消息的有序性担保，又可以提供消费者之间的负载均衡。这是通过将 Topic 中的 partition 绑定到消费者组中的具体消费者实现的。通过这种方案我们可以保证消费者是某个 partition 唯一消

费者，从而完成消息的有序消费。因为 Topic 有多个 partition 所以在消费者实例之间还是负载均衡的。注意，虽然有以上方案，但是如果想担保消息的有序性那么我们就不能为一个 partition 注册多个消费者了。



一个两节点的 kafka 集群支持的 2 个消费组的四个分区 (P0-P3)。消费者 A 有两个消费者实例，消费者 B 有四个消费者实例。

Kafka 仅仅提供提供 partition 之内的消息的全局有序，在不同的 partition 之间不能担保。partition 的消息有序性加上可以按照指定的 key 划分消息的 partition，这基本上满足了大部分应用的需求。如果你必须要实现一个全局有序的消息队列，那么可以采用 Topic 只划分 1 个 partition 来实现。但是这就意味着你的每个消费组只有有唯一的一个消费者进程。

Guarantees

在上层 Kafka 提供一下可靠性保证：

- 生产者发送到 Topic 某个 partition 的消息都被有序的追加到之前发送的消息之后。意思就是如果一个消息 M1、M2 是同一个生产者发送的，先发送的 M1 那么 M1 的 offse 就比 M2 更小也就是更早的保存在 log 中。
- 对于特定的消费者，它观察到的消息的顺序与消息保存到 log 中的顺序一致。
- 对于一个复制 N 份的 Topic，系统能保证在 N-1 台服务器失效的情况下不丢失任何已提交到 log 中的消息。

更多关于可靠性保证的细节，将会在后续的本文档设计章节进行讨论。

1.1 Introduction

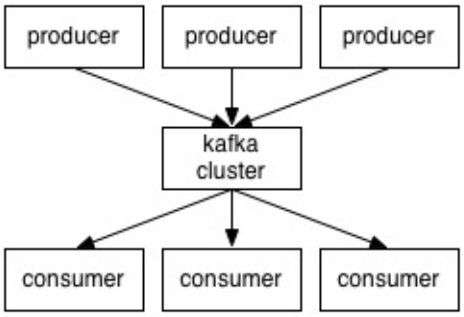
Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

What does all that mean?

First let's review some basic messaging terminology:

- Kafka maintains feeds of messages in categories called *topics*.
- We'll call processes that publish messages to a Kafka topic *producers*.
- We'll call processes that subscribe to topics and process the feed of published messages *consumers*.
- Kafka is run as a cluster comprised of one or more servers each of which is called a *broker*.

So, at a high level, producers send messages over the network to the Kafka cluster which in turn serves them up to consumers like this:

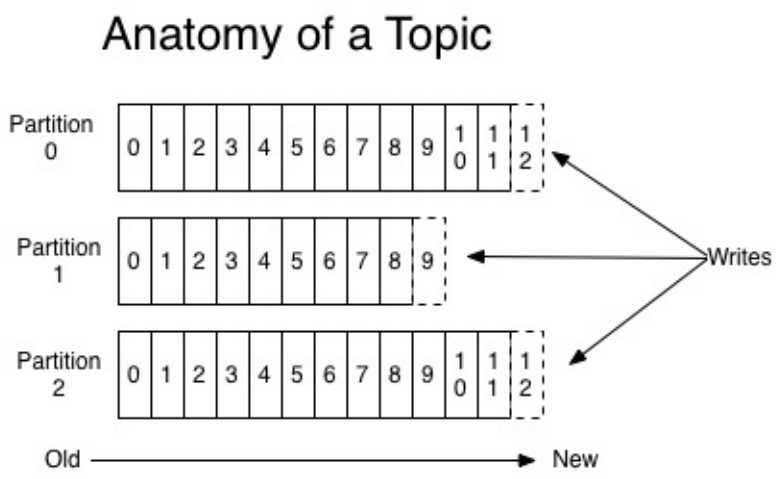


Communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. We provide a Java client for Kafka, but clients are available in many languages.

Topics and Logs

Let's first dive into the high-level abstraction Kafka provides—the topic.

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Each partition is an ordered, immutable sequence of messages that is continually appended to—a commit log. The messages in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each message within the partition.

The Kafka cluster retains all published messages—whether or not they have been consumed—for a configurable period of time. For example if the log retention is set to two days, then for the two days after a message is published it is available for consumption, after which it will be discarded to free up space. Kafka's performance is effectively constant with respect to data size so retaining lots of data is not a problem.

In fact the only metadata retained on a per-consumer basis is the position of the consumer in the log, called the "offset". This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads messages, but in fact the position is controlled by the consumer and it can consume messages in any order it likes. For example a consumer can reset to an older offset to reprocess.

This combination of features means that Kafka consumers are very cheap—they can come and go without much impact on the cluster or on other consumers. For example, you can use our command line tools to "tail" the contents of any topic without changing what is consumed by any existing consumers.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism—more on that in a bit.

Distribution

The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.

Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which message to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the message). More on the use of partitioning in a second.

Consumers

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers. Kafka offers a single consumer abstraction that generalizes both of these—the *consumer group*.

Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.

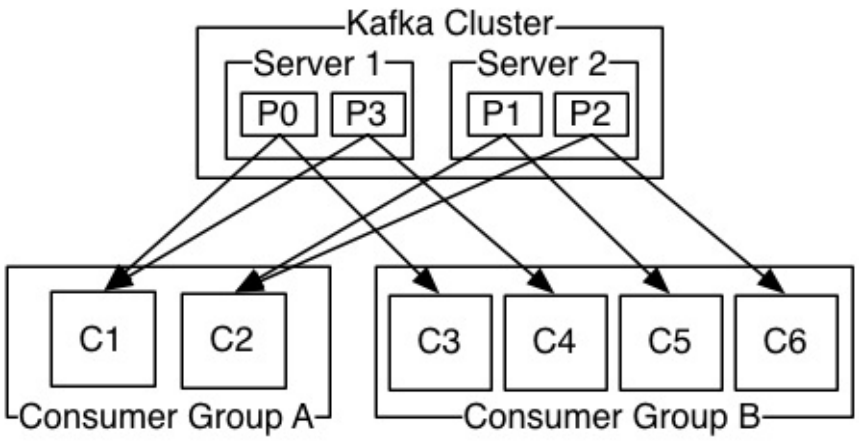
If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers.

If all the consumer instances have different consumer groups, then this works like publish-subscribe and all messages are broadcast to all consumers.

More commonly, however, we have found that topics have a small number of consumer groups, one for each "logical subscriber". Each group is composed of many consumer instances for scalability and fault tolerance. This is nothing more than publish-subscribe semantics where the subscriber is a cluster of consumers instead of a single process.

Kafka has stronger ordering guarantees than a traditional messaging system, too.

A traditional queue retains messages in-order on the server, and if multiple consumers consume from the queue then the server hands out messages in the order they are stored. However, although the server hands out messages in order, the messages are delivered asynchronously to consumers, so they may arrive out of order on different consumers. This effectively



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

means the ordering of the messages is lost in the presence of parallel consumption. Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is no parallelism in processing.

Kafka does it better. By having a notion of parallelism—the partition—within the topics, Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer instances in a consumer group than partitions.

Kafka only provides a total order over messages *within* a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over messages this can be achieved with a topic that has only one partition, though this will mean only one consumer process per consumer group.

Guarantees

At a high-level Kafka gives the following guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a message M1 is sent by the same producer as a message M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without

losing any messages committed to the log.

More details on these guarantees are given in the design section of the documentation.

1.2 应用场景 Use Cases

本章节介绍几种主流的 Apache Kafka 的应用场景。关于几个场景实践的概述可以参考这篇博客.

信息系统 Messaging

Kafka 可以作为传统信息中间件的替代产品。消息中间件可能因为各种目的被引入到系统之中（解耦生产者和消费、堆积未处理的消息）。对比其他的信息中间件，Kafka 的高吞吐量、内建分区、副本、容错等特性，使得它在大规模伸缩性消息处理应用中成为了一个很好的解决方案。

根据我们的在消息系统场景的经验，系统常常需求的吞吐量并不高，但是要求很低的点到点的延迟并且依赖 Kafka 提供的强有力的持久化功能。

在这个领域 Kafka 常常被拿来与传统的消息中间件系统进行对比，例如 **ActiveMQ** 或者 **RabbitMQ**。

网站活动追踪 Website Activity Tracking

Kafka 原本的应用场景要求它能重建一个用户活动追踪管线作为一个实时的发布与订阅消息源。意思就是用户在网站上的动作事件（如浏览页面、搜索、或者其它操作）被发布到每个动作对应的中心化 Topic 上。使得这些数据源能被不同场景的需求订阅到，这些场景包括实时处理、实时监控、导入 Hadoop 或用于离线处理、报表的离线数据仓库中。

活动追踪通常情况下是非常高频的，因为很多活动消息是由每个用户的页面浏览产生的。

监控 Metrics

Kafka 常被用来处理操作监控数据。这涉及到聚合统计分布式应用的数据来产生一个中心化的操作数据数据源。

日志收集 Log Aggregation

很多人把 Kafka 用作日志收集服务的替换方案。日志收集基础就是从服务器收集物理日志文件并将其放到统一的地方（文件服务器或者 HDFS）存储以便后续处理。Kafka 抽象了文件的细节，为日志或者事件数据提供了一个消息流的抽象。这样就可以很好的支持低延迟处理需求、多数据源需求，分布式数据消费需求。与 Scribe 或 Flume 等其它的日志收集系统相比，Kafka 提供了同样优秀的性能，基于副本的更强的持久化保证和更低的点到点的延迟。

流处理 Stream Processing

许多 Kafka 用户是在一个多级组成的处理管道中处理数据的，他们的从 Kafka 的 Topic 上消费原始数据，然后对消息进行聚合、丰富、转发到新的 Topic 用于消费或者转入下一步处理。例如，一个推荐新闻文章的处理管线可能从 RSS 数据源爬取文章内容，然后将它发布到"articles" Topic；然后后续的处理器再对文章内容进行规范化、去重，然后将规整的文章内容发布到一个新的 Topic 上；最后的处理管线可能尝试将这个内容推荐给用户。这样的处理管线通过一个个独立的 topic 构建起了一个实时数据流图。从 0.10.0.0 开始，Kafka 提供了一个称为 **Kafka Streams** 的轻量级但强大的流处理包来实现如上所述的处理流程。从 Kafka Streams 开始，Kafka 成为了与 **Apache Storm** 和 **Apache Samza** 类似的开源流处理工具的新选择。

事件溯源 Event Sourcing

事件溯源 **Event sourcing** 是一种将状态变更记录成一个时序队列的应用设计模式。Kafka 对海量存储日志数据的支撑使得它可做这种应用非常好的后端支撑。

提交日志 Commit Log

Kafka 可以作为分布式系统的外部提交日志服务。这些日志可以用于节点间数据复制和失败阶段的数据重同步过程。Kafka 的 日志合并 **log compaction** 功能可以很好的支撑这种应用场景。Kafka 这种应用和 **Apache BookKeeper** 功能相似。

1.2 Use Cases

Here is a description of a few of the popular use cases for Apache Kafka. For an overview of a number of these areas in action, see **this blog post**.

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

In this domain Kafka is comparable to traditional messaging systems such as **ActiveMQ** or **RabbitMQ**.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view.

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a

cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and published the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called **Kafka Streams** is available in Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include **Apache Storm** and **Apache Samza**.

Event Sourcing

Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The **log compaction** feature in Kafka helps support this usage. In this usage Kafka is similar to **Apache BookKeeper** project.

1.3 快速入门 Quick Start

本教程假设你从零开始，没有 Kafka 和 ZooKeeper 历史数据。

Step 1: 下载代码

下载 0.10.0.0 的正式版本并解压。

```
> tar -xzf kafka_2.11-0.10.0.0.tgz
> cd kafka_2.11-0.10.0.0
```

Step 2: 启动服务器

Kafka 依赖 ZooKeeper 因此你首先启动一个 ZooKeeper 服务器。如果你没有一个现成的实例，你可以使用 Kafka 包里面的默认脚本快速安装并启动一个全新的单节点 ZooKeeper 实例。

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties
(org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
```

然后开始启动 Kafka 服务器：

```
> bin/kafka-server-start.sh config/server.properties
[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048
576 (kafka.utils.VerifiableProperties)
...
```

Step 3: 创建 Topic

现在我们开始创建一个名为“test”的单分区单副本的 Topic。

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --par
titions 1 --topic test
```

现在我们应该可以通过运行 `list topic` 命令查看到这个 topic:

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

另外，除了手工创建 topic 以外，你也可以将你的 brokers 配置成消息发布到一个不存在的 topic 时自动创建此 topics。

Step 4: 发送消息

Kafka 附带一个命令行客户端可以从文件或者标准输入中读取输入然后发送这个消息到 Kafka 集群。默认情况下每行信息被当做一个消息发送。

运行生产者脚本然后在终端中输入一些消息并发送到服务器。

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

Step 5: 启动消费者

Kafka 也附带了一个命令行的消费者可以导出这些消息到标准输出。

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
This is a message
This is another message
```

如果你在不同的终端运行以上两个命令，那么现在你就应该能在生产者的终端中键入消息同时在消费者的终端中看到。

所有的命令行工具都有很多可选的参数；不添加参数直接执行这些命令将会显示它们的使用方法，更多内容可以参考他们的使用手册。

Step 6: 配置一个多节点集群

我们已经成功的以单 broker 的模式运行起来了，但这并没有实际的意义。对于 Kafka 来说，一个单独的 broker 就是一个大小为 1 的集群，所以集群模式无非多启动几个 broker 实例。为了更好的理解，我们将集群扩展到 3 个节点。

首先为每个 broker 准备配置文件

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

修改新的配置文件的以下属性：

```
config/server-1.properties:
    broker.id=1
    listeners=PLAINTEXT://:9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    listeners=PLAINTEXT://:9094
    log.dir=/tmp/kafka-logs-2
```

`broker.id` 属性指定了节点在集群中的唯一的不变的名字。我们必须更改端口和日志目录主要是因为我们在同一个机器上运行所有的上述实例，我们必须要保证 brokers 不会去注册相同端口或者覆盖其它人的数据。

我们已经有了 ZooKeeper 并且已经有一个阶段启动了，接下来我们只要启动另外两个节点。

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

现在我们可以创建一个新的 topic 并制定副本数量为 3：

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic
```

现在我们启动了一个集群，我们如何知道每个 broker 具体的工作呢？为了回答这个问题，可以运行 `describe topics` 命令：

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 1    Replicas: 1,2,0    Isr: 1,2,0
```

解释一下输出的内容，第一行给出了所有 partition 的一个总结，每行给出了一个 partition 的信息。因为我们这个 topic 只有一个 partition 所以只有一行信息。

- “leader” 负责响应给定 partition 的所有读和写请求。每个节点都会是从所有 partition 集合随机选定的一个子集的“leader”
- “replicas” 是一个节点列表，包含所有复制了此 partition log 的节点，不管这个节点是否为 leader 也不管这个节点当前是否存活
- “isr” 是当前处于同步状态的副本。这是“replicas”列表的一个子集表示当前处于存活状态并且与 leader 一致的节点

注意在我们的例子中 node 1 是这个仅有一个 partition 的 topic 的 leader。

我们可以对我们原来创建的 topic 运行相同的命令，来观察它保存在哪里：

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
```

我们很明显的发现原来的那个 topic 没有副本而且它在我们创建它时集群仅有的一个节点 server 0 上。

现在我们发布几个消息到我们的新 topic 上：

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-top
ic
...
my test message 1
my test message 2
^C
```

现在让我们消费这几个消息：

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my
-replicated-topic
...
my test message 1
my test message 2
^C
```

现在让我们测试一下集群容错。Broker 1 正在作为 leader 所以我们杀掉它：

```
> ps | grep server-1.properties
7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/
bin/java...
> kill -9 7564
```

集群 leader 已经切换到一个从服务器上，node 1 节点也不在出现在同步副本列表中了：

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topi
c
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 2    Replicas: 1,2,0    Isr:
 2,0
```

即使原来负责写的节点已经失效，消息仍然可以被正常消费。

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my
-replicated-topic
...
my test message 1
my test message 2
^C
```

Step 7: 使用 Kafka Connect 进行数据导入导出 Use Kafka Connect to import/export data

从终端写入数据，数据也写回终端很方便快速开始学习和使用 Kafka 。但是你可能更希望从其它的数据源导入或者导出 Kafka 的数据到其它的系统。相比其它系统需要自己编写集成代码，你可以直接使用 Kafka 的 Connect 直接导入或者导出数据。Kafka Connect 是 Kafka 自带的用于数据导入和导出的工具。它是一个扩展的可运行连接器 (runs *connectors*) 工具，可使用自定义的逻辑来实现与外部系统的集成交互。在这个快速入门中我们将介绍如何通过一个简单的从文本导入数据、导出数据到文本的连接器来调用 Kafka Connect。首先我们从创建一些测试的基础数据开始：

```
> echo -e "foo\nbar" > test.txt
```

接下来我们采用 *standalone* 模式启动两个 connectors, 也就是让它们都运行在独立的、本地的、不同的进程中。我们提供三个参数化的配置文件，第一个提供共有的配置用于 Kafka Connect 处理，包含共有的配置比如连接哪个 Kafka broker 和数据的序列化格式。剩下的配置文件制定每个 connector 创建的特定信息。这些文件包括唯一的 connector 的名字，connector 要实例化的类和其它的一些 connector 必备的配置。

```
> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-s
ource.properties config/connect-file-sink.properties
```

上述简单的配置文件已经被包含在 Kafka 的发行包中，它们将使用默认的之前我们启动的本地集群配置创建两个 connector：第一个作为源 connector 从一个文件中读取每行数据然后将他们发送 Kafka 的 topic，第二个是一个输出 (sink)connector 从 Kafka 的 topic 读取消息，然后将它们一行行输出到输出文件中。在启动的过程你将看到一些日志消息，包括一些提示 connector 正在被实例化的信息。一旦 Kafka Connect 进程启动以后，源 connector 应该开始从 `test.txt` 中读取数据行，并将他们发送到 topic `connect-test` 上，然后 `输出 connector` 将会开始从 topic 读取消息然后把它们写入到 `test.sink.txt` 中。

我们可以查看输出文件来验证通过整个管线投递的数据：

```
> cat test.sink.txt
foo
bar
```

注意这些数据已经被保存到了 Kafka 的 `connect-test` topic 中，所以我们还可以运行一个终端消费者来看到这些数据（或者使用自定义的消费者代码来处理数据）：

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic connect-test --from
-beginning
{"schema":{"type":"string","optional":false},"payload":"foo"}
{"schema":{"type":"string","optional":false},"payload":"bar"}
...
```

connector 在持续的处理着数据，所以我们可以向文件中添加数据然后观察到它在这个管线中的传递：

```
> echo "Another line" >> test.txt
```

你应该可以观察到新的数据行出现在终端消费者中和输出文件中。

Step 8: 使用 Kafka Streams 来处理数据 Use Kafka Streams to process data

Kafka Streams 是一个用来对 Kafka brokers 中保存的数据进行实时处理和分析的客户端库。这个入门示例 将演示如何启动一个采用此类库实现的流处理程序。下面是 `WordCountDemo` 示例代码的 GIST（为了方便阅读已经转化成了 Java 8 的 lambda 表达式）。

```
KTable wordCounts = textLines
    // 按照空格将每个文本行拆分成单词
    // Split each text line, by whitespace, into words.
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // 确保每个单词作为记录的 key 值以便于下一步的聚合
    // Ensure the words are available as record keys for the next aggregate operation.
    .map((key, value) -> new KeyValue<>(value, value))
    // 计算每个单词的出现频率并将他们保存到“Counts”的表中
    // Count the occurrences of each word (record key) and store the results into a ta
ble named "Counts".
    .countByKey("Counts")
```

上述代码实现了计算每个单词出现频率直方图的单词计数算法。但是它与之前常见的操作有限数据的示例相比有明显的不同，它被设计成一个操作无边界限制的流数据的程序。与有界算法相似它是一个有状态算法，它可以跟踪并更新单词的计数。但是它必须支持处理无边界限制的数据输入的假设，它将在处理数据的过程持续的输出自身的状态和结果，因为它不能明确的知道合适已经完成了所有输入数据的处理。

接下来我们准备一些发送到 Kafka topic 的输入数据，随后它们将被 Kafka Streams 程序处理。

```
> echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" > file-i
nput.txt
```

ing):

接下来我们使用终端生产者发送这些输入数据到名为**streams-file-input**的输入 topic（在实际应用中，流数据会是不断流入处理程序启动和运行用的 Kafka）：

```
> bin/kafka-topics.sh --create \
            --zookeeper localhost:2181 \
            --replication-factor 1 \
            --partitions 1 \
            --topic streams-file-input
```

```
> cat file-input.txt | bin/kafka-console-producer.sh --broker-list localhost:9092 --to
pic streams-file-input
```

现在我们可以启动 WordCount 示例程序来处理这些数据了：

```
> bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

在 STDOUT 终端不会有任何日志输出，因为所有的结果被不断的写回了另外一个名为**streams-wordcount-output**的 topic 上。这个实例将会运行一会儿，之后与典型的流处理程序不同它将会自动退出。

现在我们可以通过读取这个单词计数示例程序的输出 topic 来验证结果：

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 \
            --topic streams-wordcount-output \
            --from-beginning \
            --formatter kafka.tools.DefaultMessageFormatter \
            --property print.key=true \
            --property print.value=true \
            --property key.deserializer=org.apache.kafka.common.serialization.StringDe
serializer \
            --property value.deserializer=org.apache.kafka.common.serialization.LongDe
serializer
```

以下输出数据将会被打印到终端上：

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
join    1
kafka   3
summit  1
```

可以看到，第一列是 Kafka 的消息的健，第二列是这个消息的值，他们都是 `java.lang.String` 格式的。注意这个输出结果实际上是一个持续更新的流，每一行（例如、上述原始输出的每一行）是一个单词更新之后的计数。对于 key 相同的多行记录，每行都是前面一行的更新。

现在你可以向**streams-file-input** topic 写入更多的消息并观察**streams-wordcount-output** topic 表述更新单词计数的新的消息。

你可以通过键入**Ctrl-C**来终止终端消费者。

1.3 Quick Start

This tutorial assumes you are starting fresh and have no existing Kafka or ZooKeeper data.

Step 1: Download the code

Download the 0.10.0.0 release and un-tar it.

```
> tar -xzf kafka_2.11-0.10.0.0.tgz
> cd kafka_2.11-0.10.0.0
```

Step 2: Start the server

Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties
 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
```

Now start the Kafka server:

```
> bin/kafka-server-start.sh config/server.properties
[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048
576 (kafka.utils.VerifiableProperties)
...
```

Step 3: Create a topic

Let's create a topic named "test" with a single partition and only one replica:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --par
titions 1 --topic test
```

We can now see that topic if we run the list topic command:

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.

Step 4: Send some messages

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default each line will be sent as a separate message.

Run the producer and then type a few messages into the console to send to the server.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

Step 5: Start a consumer

Kafka also has a command line consumer that will dump out messages to standard output.

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginni
ng
This is a message
This is another message
```

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

All of the command line tools have additional options; running the command with no arguments will display usage information documenting them in more detail.

Step 6: Setting up a multi-broker cluster

So far we have been running against a single broker, but that's no fun. For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers:

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

Now edit these new files and set the following properties:

```
config/server-1.properties:
    broker.id=1
    listeners=PLAINTEXT://:9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    listeners=PLAINTEXT://:9094
    log.dir=/tmp/kafka-logs-2
```

The `broker.id` property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each others data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

Now create a new topic with a replication factor of three:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --par
titions 1 --topic my-replicated-topic
```

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topi
c
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 1    Replicas: 1,2,0    Isr:
 1,2,0
```

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.

- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
```

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-top
ic
...
my test message 1
my test message 2
^C
```

Now let's consume these messages:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my
-replicated-topic
...
my test message 1
my test message 2
^C
```

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

```
> ps | grep server-1.properties
7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/
bin/java...
> kill -9 7564
```

Leadership has switched to one of the slaves and node 1 is no longer in the in-sync replica set:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topi
c
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 2    Replicas: 1,2,0    Isr:
 2,0
```

But the messages are still be available for consumption even though the leader that took the writes originally is down:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my
-replicated-topic
...
my test message 1
my test message 2
^C
```

Step 7: Use Kafka Connect to import\/export data

Writing data from the console and writing it back to the console is a convenient place to start, but you'll probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect to import or export data. Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs*connectors*, which implement the custom logic for interacting with an external system. In this quickstart we'll see how to run Kafka Connect with simple connectors that import data from a file to a Kafka topic and export data from a Kafka topic to a file. First, we'll start by creating some seed data to test with:

```
> echo -e "foo\nbar" > test.txt
```

Next, we'll start two connectors running in *standalone* mode, which means they run in a single, local, dedicated process. We provide three configuration files as parameters. The first is always the configuration for the Kafka Connect process, containing common configuration such as the Kafka brokers to connect to and the serialization format for data. The remaining configuration files each specify a connector to create. These files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

```
> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-s
ource.properties config/connect-file-sink.properties
```

These sample configuration files, included with Kafka, use the default local cluster configuration you started earlier and create two connectors: the first is a source connector that reads lines from an input file and produces each to a Kafka topic and the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file. During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated. Once the Kafka Connect process has started, the source connector should start reading lines from

```
test.txt
```

and producing them to the topic

```
connect-test
```

, and the sink connector should start reading messages from the topic

```
connect-test
```

and write them to the file

```
test.sink.txt
```

. We can verify the data has been delivered through the entire pipeline by examining the contents of the output file:

```
> cat test.sink.txt
foo
bar
```

Note that the data is being stored in the Kafka topic

```
connect-test
```

, so we can also run a console consumer to see the data in the topic (or use custom consumer code to process it):

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic connect-test --from
-beginning
{"schema":{"type":"string","optional":false},"payload":"foo"}
{"schema":{"type":"string","optional":false},"payload":"bar"}
...
```

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

```
> echo "Another line" >> test.txt
```

You should see the line appear in the console consumer output and in the sink file.

Step 8: Use Kafka Streams to process data

Kafka Streams is a client library of Kafka for real-time stream processing and analyzing data stored in Kafka brokers. This quickstart example will demonstrate how to run a streaming application coded in this library. Here is the gist of the `WordCountDemo` example code (converted to use Java 8 lambda expressions for easy reading).

```
KTable wordCounts = textLines
    // Split each text line, by whitespace, into words.
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

    // Ensure the words are available as record keys for the next aggregate operation.
    .map((key, value) -> new KeyValue<>(value, value))

    // Count the occurrences of each word (record key) and store the results into a ta
ble named "Counts".
    .countByKey("Counts")
```

It implements the WordCount algorithm, which computes a word occurrence histogram from the input text. However, unlike other WordCount examples you might have seen before that operate on bounded data, the WordCount demo application behaves slightly differently because it is designed to operate on an **infinite, unbounded stream** of data. Similar to the bounded variant, it is a stateful algorithm that tracks and updates the counts of words. However, since it must assume potentially unbounded input data, it will periodically output its current state and results while continuing to process more data because it cannot know when it has processed "all" the input data.

We will now prepare input data to a Kafka topic, which will subsequently processed by a Kafka Streams application.

```
> echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" > file-i
nput.txt
```

Next, we send this input data to the input topic named **streams-file-input** using the console producer (in practice, stream data will likely be flowing continuously into Kafka where the application will be up and running):

```
> bin/kafka-topics.sh --create \
            --zookeeper localhost:2181 \
            --replication-factor 1 \
            --partitions 1 \
            --topic streams-file-input
```

```
> cat file-input.txt | bin/kafka-console-producer.sh --broker-list localhost:9092 --to
pic streams-file-input
```

We can now run the WordCount demo application to process the input data:

```
> bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

There won't be any STDOUT output except log entries as the results are continuously written back into another topic named **streams-wordcount-output** in Kafka. The demo will run for a few seconds and then, unlike typical stream processing applications, terminate automatically.

We can now inspect the output of the WordCount demo application by reading from its output topic:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 \
            --topic streams-wordcount-output \
            --from-beginning \
            --formatter kafka.tools.DefaultMessageFormatter \
            --property print.key=true \
            --property print.value=true \
            --property key.deserializer=org.apache.kafka.common.serialization.StringDe
serializer \
            --property value.deserializer=org.apache.kafka.common.serialization.LongDe
serializer
```

with the following output data being printed to the console:

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
join    1
kafka   3
summit  1
```

Here, the first column is the Kafka message key, and the second column is the message value, both in in `java.lang.String` format. Note that the output is actually a continuous stream of updates, where each data record (i.e. each line in the original output above) is an updated count of a single word, aka record key such as "kafka". For multiple records with the same key, each later record is an update of the previous one.

Now you can write more input messages to the **streams-file-input** topic and observe additional messages added to **streams-wordcount-output** topic, reflecting updated word counts (e.g., using the console producer and the console consumer, as described above).

You can stop the console consumer via **Ctrl-C**.

1.4 生态 Ecosystem

在 Kafka 的官方分发包之外，还有很多各式各样的和 Kafka 整合的工具。生态页面 (ecosystem page) 列出了很多这样工具，包括流处理系统、Hadoop 整合、监控和部署工具等等。

1.4 Ecosystem

There are a plethora of tools that integrate with Kafka outside the main distribution. The **ecosystem page** lists many of these, including stream processing systems, Hadoop integration, monitoring, and deployment tools.

1.5 从早期版本升级

从 0.8.x 或 0.9.x 升级到 0.10.0.0

0.10.0.0 有一些潜在的不兼容变更)（在升级前请一定对其进行检查）和升级过程中性能下降的风险。遵照一下推荐的滚动升级方案，可以保证你在升级过程及之后都不需要停机并且没有性能下降。

注意：因为新的协议的引入，一定要先升级你的 Kafka 集群然后在升级客户端。

注意：*0.9.0.0* 版本客户端因为一个在 ***0.9.0.0*** 版本客户端引入的 ***bug*** 使得它不能与 ***0.10.0.x*** 版本中间件协作，这包括依赖 ***ZooKeeper*** 的客户端（原 ***Scala*** 上层（***high-level***）消费者和使用原消费者的 ***MirrorMaker***）。因此，***0.9.0.0*** 的客户端应该在中间件升级到 ***0.10.0.0***之前* 被升级到 0.9.0.1 上。这个步骤对于 0.8.X 和 0.9.0.1 的客户端不是不需要。

采用滚动升级：

1. 更新所有中间件的 server.properties 文件，添加如下配置：
    - inter.broker.protocol.version=CURRENT_KAFKA_VERSION (e.g. 0.8.2 or 0.9.0.0).
    - log.message.format.version=CURRENT_KAFKA_VERSION （参考 升级过程可能的性能影响 了解这个配置项的作用）
2. 升级中间件。这个过程可以逐个中间件的去完成，只要将它下线然后升级代码然后重启即可。
3. 当整个集群都升级完成以后，再去处理协议版本问题，通过编辑 inter.broker.protocol.version 并设置为 0.10.0.0 即可。注意：此时还不应该去修改日志格式版本参数 log.message.format.version- 这参数只能在所有的客户端都升级到 0.10.0.0 之后去修改。
4. 逐个重启中间件让新的协议版本生效。
5. 当所有的消费者都升级到 0.10.0 以后，逐个中间件去修改 log.message.format.version 为 0.10.0 并重启。

注意： 如果你接受停机，你可简单将所有的中间件下线，升级代码然后再重启。这样它们应该都会默认使用新的协议。

注意： 修改协议版本并重启的工作你可以在升级中间件之后的任何时间进行，这个过程没必要升级代码后立即进行。

升级 0.10.0.0 过程潜在的性能影响

0.10.0 版本的消息格式引入了一个新的 timestamp 字段并对压缩的消息使用了相对偏移量。磁盘的消息格式可以通过 server.properties 文件的 log.message.format.version 进行配置。默认的消息格式是 0.10.0。对一个 0.10.0 之前版本的客户端，它只能识别 0.10.0 之前的消息格式。在这种情况下消息中间件可以将消息在响应给客户端之前转换成老的消息格式。但如此

一来中间件就不能使用零拷贝传输了（zero-copy transfer）。根据 Kafka 社区的反馈包括，升级后这对性能的影响将会是 CPU 的使用率从 20% 提升到 100%，这将迫使你必须立即升级所有的客户端到 0.10.0.0 版本来恢复性能表现。为了避免客户端升级到 0.10.0.0 之前的消息转换。你可以在升级中间件版本到 0.10.0.0 的过程中，将消息的格式参数 og.message.format.version 设置成 0.8.2 或者 0.9.0 版本。这样一来中间件依旧可以使用零拷贝传输来将消息发送到客户端。在所有的消费者升级以后，就可以修改中间件上消息格式版本到 0.10.0，享受新消息格式带来的益处包括新引入的时间戳字段和更好的消息压缩。这个转换过程的支持是为了保证兼容性和支持少量未能及时升级到新版本客户端应用而存在的。如果想在一个即将超载的集群上来支持所有客户端的流量是不现实的。因此在消息中间升级以后但是主要的客户端还没有升级的时候应该尽可能的去避免消息转换。

对于已升级到 0.10.0.0 的客户端不存在这种性能上的负面影响。

注意：设置消息格式的版本，应该保证所有的已有的消息都是这个消息格式版本之下的版本。否则 0.10.0.0 之前的客户端可能出现故障。在实践中，一旦消息的格式被设置成了 0.10.0 之后就不应该把它再修改到早期的格式上，因为这可能造成 0.10.0.0 之前版本的消费者的故障。

注意：因为每个消息新时间戳字段的引入，生产者在发送小包消息可能出现因为负载上升造成的吞吐量的下降。同理，现在复制过程每个消息也要多传输 8 个比特。如果你的集群即将达到网络容量的瓶颈，这可能造成网卡打爆并因为超载引起失败和性能问题。

注意：如果你在生产者启动了消息压缩机制，你可能发现生产者吞吐量下降和 / 或中间件消息压缩比例的下降。在接受压缩过的消息时，0.10.0 的中间避免重新压缩信息，这样原意是为了降低延迟提供吞吐量。但是在某些场景下，这可能降低生产者批处理数量，并引起吞吐量上更差的表现。如果这种情况发生了，用户可以调节生产者的 linger.ms 和 batch.size 参数来获得更好的吞吐量。另外生产者在使用 snappy 进行消息压缩时它用来消息压缩的 buffer 相比中间件的要小，这可能给磁盘上的消息的压缩率带来一个负面影响，我们计划将在后续的版本中将这个参数修改为可配置的。

0.10.0.0 潜在的不兼容修改

- 从 0.10.0.0 开始，Kafka 消息格式的版本号将用 Kafka 版本号表示。例如，消息格式版本号 0.9.0 表示最高 Kafka 0.9.0 支持的消息格式。
- 引入了 0.10.0 版本消息格式并作为默认配置。它引入了一个时间戳字段并在压缩消息中使用相对偏移量。
- 引入了 ProduceRequest/Response v2 并用作 0.10.0 消息格式的默认支持。
- 引入了 FetchRequest\/Response v2 并用作 0.10.0 消息格式的默认支持。
- 接口 MessageFormatter 从 `def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)` 变更为 `def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)`
- 接口 MessageReader 从 `def readMessage(): KeyedMessage[Array[Byte], Array[Byte]]` 变更为 `def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]`

- MessageFormatter 的包从 `kafka.tools` 变更为 `kafka.common`
- MessageReader 的包 `kafka.tools` 变更为 `kafka.common`
- MirrorMakerMessageHandler 不再暴露 `handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])` 方法，因为它从没有被调用过。
- 0.7 KafkaMigrationTool 不再和 Kafka 打包。如果你需要从 0.8 迁移到 0.10.0，请先迁移到 0.8 然后在根据文档升级过程完成从 0.8 到 0.10.0 的升级。
- 新的消费者规范化了 API 来使用 `java.util.Collection` 作为序列类型方法参数。现存的代码可能需要进行修改来实现 0.10.0 版本客户端库的协作。
- LZ4 压缩消息的处理变更为使用互操作框架规范（LZ4f v1.5.1）（interoperable framing specification）。为了保证对老客户端的兼容，这个变更只应用于 0.10.0 或之后版本的消息格式上。v0/v1 （消息格式 0.9.0) 生产或者拉取 LZ4 压缩消息的客户端将依旧使用 0.9.0 的框架实现。使用 Produce/Fetch protocols v2 协议的客户端及之后客户端应该使用互操作 LZ4f 框架。互操作（interoperable）LZ4 类库列表可以在这里找到 http://www.lz4.org/

0.10.0.0 值得关注的变更

- 从 Kafka 0.10.0.0 开始，一个称为**Kafka Streams**的新客户端类库被引入，用于对存储于 Kafka Topic 的数据进行流处理。因为上文介绍的新消息格式变更，新的客户端类库只能与 0.10.x 或以上版本的中间件协作。详细信息请参考此章节。
- 新消费者的默认配置参数 `receive.buffer.bytes` 现在变更为 64K。
- 新的消费者暴露配置参数 `exclude.internal.topics` 限制内部话题 topic（例如消费者偏移量 topic) 被意外的包括到正则表达式订阅之中。默认启动。
- 不推荐再使用原来的 Scala 生产者。用户应该尽快迁移他们的代码到 kafka-clients Jar 包中 Java 生产者上。
- 新的消费者 API 被认定为进入稳定版本（stable）

从 0.8.0, 0.8.1.X 或 0.8.2.X 升级到 0.9.0.0

0.9.0.0 存在一些潜在的不兼容变更（在升级之前请一定查阅）和中间件间部协议与上一版本也发生了的变更。这意味升级中间件和客户端可能发生于老版本的不兼容情况。在升级客户端之前一定要先升级 Kafka 集群这一点很重要。如果你使用了 MirrorMaker 下游集群也应该对其先进行升级。

滚动升级：

1. 升级所有中间的 server.properties 文件，添加如下配置：
   inter.broker.protocol.version=0.8.2.X
2. 升级中间件。这个过程可以逐个中间件的去完成，只要将它下线然后升级代码然后重启即可。

3. 在整个集群升级完成以后，设置协议版本修改 inter.broker.protocol.version 设置成 0.9.0.0
4. 逐个重启中间件让新的协议版本生效。

注意：如果你接受停机，你可简单将所有的中间件下线，升级代码然后再重启。这样它们应该都会默认使用新的协议。

注意：修改协议版本并重启的工作你可以在升级中间件之后的任何时间进行，这个过程没必要升级代码后立即进行。

0.9.0.0 潜在的不兼容变更

- 不再支持 Java 1.6。
- 不再支持 Scala 2.9。
- 1000 以上的 Broker ID 被默认保留用于自动分配 broker id。如果你的集群现在有 broker id 大于此数值应该注意修改 reserved.broker.max.id 配置。
- 配置参数 replica.lag.max.messages was removed。分区领导判定副本是否同步不再考虑延迟的消息。
- 配置参数 replica.lag.time.max.ms 现在不仅仅代表自副本最后拉取请求到现在的时间间隔，它也是副本最后完成同步的时间。副本正在从 leader 拉取消息，但是在 replica.lag.time.max.ms 时间内还没有完成最后一条信息的拉取，它也将被认为不再是同步状态。
- 压缩话题（Compacted topics）不再接受没有 key 的消息，如果消费者尝试将抛出一个异常。在 0.8.x 版本，一个不包含 key 的消息会引起日志压缩线程（log compaction thread）异常并退出（造成所有压缩话题的压缩工作中断）。
- MirrorMaker 不再支持多个目标集群。这意味着它能只能接受一个 --consumer.config 参数配置。为了镜像多个源集群，你至少要为每个源集群配置一个 MirrorMaker 实例，每个实例配置他们的消费者信息。
- 原打包在*org.apache.kafka.clients.tools.**的工具类移动到了*org.apache.kafka.tools.**里面。所有包含的脚本能正常的使用，只有自定义代码直接 import 了那些类会受到影响。
- 默认的 Kafka JVM 性能配置 (KAFKA_JVM_PERFORMANCE_OPTS) 在 kafka-run-class.sh 发生了变更。
- kafka-topics.sh 脚本 (kafka.admin.TopicCommand) 在异常退出情况 exit code 变更为非零。
- 当 topic names 引起指标冲突时 kafka-topics.sh 脚本 (kafka.admin.TopicCommand) 将打印一个 warn，这是由 topic name 使用了'.'或者'_'并且在用例中实际发生了冲突引起的。
- kafka-console-producer.sh 脚本将默认使用新的生产者实例而不是老的，用户希望使用老的生产者必须指定'old-producer'。
- 默认所有的命令行工具将打印所有的日志信息到 stderr 而不是 stdout。

0.9.0.1 值得关注的变更

- 新的 broker id 生成功能可以通过 broker.id.generation.enable 设置为 flase 关闭。
- 配置参数 log.cleaner.enable 现在默认值为 true。这意味着使用 cleanup.policy=compact 的话题将不再默认被压缩，并且一个 128M 的堆会被分配给清理进程（cleaner process），这个大小由 log.cleaner.dedupe.buffer.size 决定。在使用了压缩话题（compacted topics）时你可能需要评估你的 log.cleaner.dedupe.buffer.size 和其它 log.cleaner 配置。
- 新的消费者参数 fetch.min.bytes 默认配置为 1

0.9.0.0 中弃用

- 通过 kafka-topics.sh 脚本修改 topic 信息已经弃用。今后请使用 kafka-configs.sh 完成此功能。
- kafka-consumer-offset-checker.sh（kafka.tools.ConsumerOffsetChecker）已经弃用。今后请用 kafka-consumer-groups.sh 完成此功能。
- The kafka.tools.ProducerPerformance class has been deprecated. Going forward, please use org.apache.kafka.tools.ProducerPerformance for this functionality (kafka-producer-perf-test.sh will also be changed to use the new class).
- kafka.tools.ProducerPerformance 类已经弃用。今后使用 org.apache.kafka.tools.ProducerPerformance 完成此功能 (kafka-producer-perf-test.sh 也将变更为使用新的类）。
- 生产者配置 block.on.buffer.full 已经被弃用并在后续版本中移除。当前它的默认值已经被修改为 false。Kafka 生产者不再抛出 BufferExhaustedException 取而代之使用 max.block.ms 来阻塞，在阻塞超时以后将抛出 TimeoutException。如果 block.on.buffer.full 属性被明确配置为 true，它将设置 max.block.ms 为 Long 最大值（Long.MAX_VALUE）并且 metadata.fetch.timeout.ms 配置将不再被参考。

从 0.8.1 升级到 0.8.2

0.8.2 与 0.8.1 完全兼容。升级过程可以通过简单的逐个下线、升级代码、重启完成。

从 0.8.0 升级到 0.8.1

0.8.1 与 0.8.0 完全兼容。升级过程可以通过简单的逐个下线、升级代码、重启完成。

从 0.7 升级

0.7 的发布版本与心得发布不兼容。核心的变更涉及到了 API、ZooKeeper 数据结构、协议以及实现复制的配置（之前 0.7 缺失的）。从 0.7 升级到后续的版本需要使用特殊的工具 来完成迁移。迁移过程可以实现不停机。

1.5 Upgrading From Previous Versions

Upgrading from 0.8.x or 0.9.x to 0.10.0.0

0.10.0.0 has **potential breaking changes** (please review before upgrading) and possible **performance impact following the upgrade**. By following the recommended rolling upgrade plan below, you guarantee no downtime and no performance impact during and following the upgrade.

Note: Because new protocols are introduced, it is important to upgrade your Kafka clusters before upgrading your clients.

Notes to clients with version 0.9.0.0: Due to a bug introduced in 0.9.0.0, clients that depend on ZooKeeper (old Scala high-level Consumer and MirrorMaker if used with the old consumer) will not work with 0.10.0.x brokers. Therefore, 0.9.0.0 clients should be upgraded to 0.9.0.1 **before** brokers are upgraded to 0.10.0.x. This step is not necessary for 0.8.X or 0.9.0.1 clients.

For a rolling upgrade:

1. Update server.properties file on all brokers and add the following properties:
    - inter.broker.protocol.version=CURRENT_KAFKA_VERSION (e.g. 0.8.2 or 0.9.0.0).
    - log.message.format.version=CURRENT_KAFKA_VERSION (See **potential performance impact following the upgrade** for the details on what this configuration does.)
2. Upgrade the brokers. This can be done a broker at a time by simply bringing it down, updating the code, and restarting it.
3. Once the entire cluster is upgraded, bump the protocol version by editing inter.broker.protocol.version and setting it to 0.10.0.0. NOTE: You shouldn't touch log.message.format.version yet - this parameter should only change once all consumers have been upgraded to 0.10.0.0
4. Restart the brokers one by one for the new protocol version to take effect.
5. Once all consumers have been upgraded to 0.10.0, change log.message.format.version to 0.10.0 on each broker and restart them one by one.

Note: If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

Note: Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

Potential performance impact following upgrade to 0.10.0.0

The message format in 0.10.0 includes a new timestamp field and uses relative offsets for compressed messages. The on disk message format can be configured through log.message.format.version in the server.properties file. The default on-disk message format is 0.10.0. If a consumer client is on a version before 0.10.0.0, it only understands message formats before 0.10.0. In this case, the broker is able to convert messages from the 0.10.0 format to an earlier format before sending the response to the consumer on an older version. However, the broker can't use zero-copy transfer in this case. Reports from the Kafka community on the performance impact have shown CPU utilization going from 20% before to 100% after an upgrade, which forced an immediate upgrade of all clients to bring performance back to normal. To avoid such message conversion before consumers are upgraded to 0.10.0.0, one can set log.message.format.version to 0.8.2 or 0.9.0 when upgrading the broker to 0.10.0.0. This way, the broker can still use zero-copy transfer to send the data to the old consumers. Once consumers are upgraded, one can change the message format to 0.10.0 on the broker and enjoy the new message format that includes new timestamp and improved compression. The conversion is supported to ensure compatibility and can be useful to support a few apps that have not updated to newer clients yet, but is impractical to support all consumer traffic on even an overprovisioned cluster. Therefore it is critical to avoid the message conversion as much as possible when brokers have been upgraded but the majority of clients have not.

For clients that are upgraded to 0.10.0.0, there is no performance impact.

Note: By setting the message format version, one certifies that all existing messages are on or below that message format version. Otherwise consumers before 0.10.0.0 might break. In particular, after the message format is set to 0.10.0, one should not change it back to an earlier format as it may break consumers on versions before 0.10.0.0.

Note: Due to the additional timestamp introduced in each message, producers sending small messages may see a message throughput degradation because of the increased overhead. Likewise, replication now transmits an additional 8 bytes per message. If you're running close to the network capacity of your cluster, it's possible that you'll overwhelm the network cards and see failures and performance issues due to the overload.

Note: If you have enabled compression on producers, you may notice reduced producer throughput and\/or lower compression rate on the broker in some cases. When receiving compressed messages, 0.10.0 brokers avoid recompressing the messages, which in general reduces the latency and improves the throughput. In certain cases, however, this may reduce the batching size on the producer, which could lead to worse throughput. If this happens, users can tune linger.ms and batch.size of the producer for better throughput. In

addition, the producer buffer used for compressing messages with snappy is smaller than the one used by the broker, which may have a negative impact on the compression ratio for the messages on disk. We intend to make this configurable in a future Kafka release.

Potential breaking changes in 0.10.0.0

- Starting from Kafka 0.10.0.0, the message format version in Kafka is represented as the Kafka version. For example, message format 0.9.0 refers to the highest message version supported by Kafka 0.9.0.
- Message format 0.10.0 has been introduced and it is used by default. It includes a timestamp field in the messages and relative offsets are used for compressed messages.
- ProduceRequest\/Response v2 has been introduced and it is used by default to support message format 0.10.0
- FetchRequest\/Response v2 has been introduced and it is used by default to support message format 0.10.0
- MessageFormatter interface was changed from `def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)` to `def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)`
- MessageReader interface was changed from `def readMessage(): KeyedMessage[Array[Byte], Array[Byte]]` to `def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]`
- MessageFormatter's package was changed from `kafka.tools` to `kafka.common`
- MessageReader's package was changed from `kafka.tools` to `kafka.common`
- MirrorMakerMessageHandler no longer exposes the `handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])` method as it was never called.
- The 0.7 KafkaMigrationTool is no longer packaged with Kafka. If you need to migrate from 0.7 to 0.10.0, please migrate to 0.8 first and then follow the documented upgrade process to upgrade from 0.8 to 0.10.0.
- The new consumer has standardized its APIs to accept `java.util.Collection` as the sequence type for method parameters. Existing code may have to be updated to work with the 0.10.0 client library.
- LZ4-compressed message handling was changed to use an interoperable framing specification (LZ4f v1.5.1). To maintain compatibility with old clients, this change only applies to Message format 0.10.0 and later. Clients that Produce\/Fetch LZ4-compressed messages using v0\/v1 (Message format 0.9.0) should continue to use the 0.9.0 framing implementation. Clients that use Produce\/Fetch protocols v2 or later should use interoperable LZ4f framing. A list of interoperable LZ4 libraries is available at http:\/\/www.lz4.org\/

Notable changes in 0.10.0.0

- Starting from Kafka 0.10.0.0, a new client library named **Kafka Streams** is available for stream processing on data stored in Kafka topics. This new client library only works with 0.10.x and upward versioned brokers due to message format changes mentioned above. For more information please read **this section**.
- The default value of the configuration parameter `receive.buffer.bytes` is now 64K for the new consumer.
- The new consumer now exposes the configuration parameter `exclude.internal.topics` to restrict internal topics (such as the consumer offsets topic) from accidentally being included in regular expression subscriptions. By default, it is enabled.
- The old Scala producer has been deprecated. Users should migrate their code to the Java producer included in the kafka-clients JAR as soon as possible.
- The new consumer API has been marked stable.

Upgrading from 0.8.0, 0.8.1.X or 0.8.2.X to 0.9.0.0

0.9.0.0 has **potential breaking changes** (please review before upgrading) and an inter-broker protocol change from previous versions. This means that upgraded brokers and clients may not be compatible with older versions. It is important that you upgrade your Kafka cluster before upgrading your clients. If you are using MirrorMaker downstream clusters should be upgraded first as well.

For a rolling upgrade:

1. Update server.properties file on all brokers and add the following property: inter.broker.protocol.version=0.8.2.X
2. Upgrade the brokers. This can be done a broker at a time by simply bringing it down, updating the code, and restarting it.
3. Once the entire cluster is upgraded, bump the protocol version by editing inter.broker.protocol.version and setting it to 0.9.0.0.
4. Restart the brokers one by one for the new protocol version to take effect

Note: If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

Note: Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

Potential breaking changes in 0.9.0.0

- Java 1.6 is no longer supported.
- Scala 2.9 is no longer supported.
- Broker IDs above 1000 are now reserved by default to automatically assigned broker IDs. If your cluster has existing broker IDs above that threshold make sure to increase

the reserved.broker.max.id broker configuration property accordingly.
- Configuration parameter replica.lag.max.messages was removed. Partition leaders will no longer consider the number of lagging messages when deciding which replicas are in sync.
- Configuration parameter replica.lag.time.max.ms now refers not just to the time passed since last fetch request from replica, but also to time since the replica last caught up. Replicas that are still fetching messages from leaders but did not catch up to the latest messages in replica.lag.time.max.ms will be considered out of sync.
- Compacted topics no longer accept messages without key and an exception is thrown by the producer if this is attempted. In 0.8.x, a message without key would cause the log compaction thread to subsequently complain and quit (and stop compacting all compacted topics).
- MirrorMaker no longer supports multiple target clusters. As a result it will only accept a single --consumer.config parameter. To mirror multiple source clusters, you will need at least one MirrorMaker instance per source cluster, each with its own consumer configuration.
- Tools packaged under *org.apache.kafka.clients.tools.** have been moved to *org.apache.kafka.tools.**. All included scripts will still function as usual, only custom code directly importing these classes will be affected.
- The default Kafka JVM performance options (KAFKA_JVM_PERFORMANCE_OPTS) have been changed in kafka-run-class.sh.
- The kafka-topics.sh script (kafka.admin.TopicCommand) now exits with non-zero exit code on failure.
- The kafka-topics.sh script (kafka.admin.TopicCommand) will now print a warning when topic names risk metric collisions due to the use of a '.' or '_' in the topic name, and error in the case of an actual collision.
- The kafka-console-producer.sh script (kafka.tools.ConsoleProducer) will use the new producer instead of the old producer be default, and users have to specify 'old-producer' to use the old producer.
- By default all command line tools will print all logging messages to stderr instead of stdout.

Notable changes in 0.9.0.1

- The new broker id generation feature can be disabled by setting broker.id.generation.enable to false.
- Configuration parameter log.cleaner.enable is now true by default. This means topics with a cleanup.policy=compact will now be compacted by default, and 128 MB of heap will be allocated to the cleaner process via log.cleaner.dedupe.buffer.size. You may want to review log.cleaner.dedupe.buffer.size and the other log.cleaner configuration values based on your usage of compacted topics.

- Default value of configuration parameter fetch.min.bytes for the new consumer is now 1 by default.

Deprecations in 0.9.0.0

- Altering topic configuration from the kafka-topics.sh script (kafka.admin.TopicCommand) has been deprecated. Going forward, please use the kafka-configs.sh script (kafka.admin.ConfigCommand) for this functionality.
- The kafka-consumer-offset-checker.sh (kafka.tools.ConsumerOffsetChecker) has been deprecated. Going forward, please use kafka-consumer-groups.sh (kafka.admin.ConsumerGroupCommand) for this functionality.
- The kafka.tools.ProducerPerformance class has been deprecated. Going forward, please use org.apache.kafka.tools.ProducerPerformance for this functionality (kafka-producer-perf-test.sh will also be changed to use the new class).
- The producer config block.on.buffer.full has been deprecated and will be removed in future release. Currently its default value has been changed to false. The KafkaProducer will no longer throw BufferExhaustedException but instead will use max.block.ms value to block, after which it will throw a TimeoutException. If block.on.buffer.full property is set to true explicitly, it will set the max.block.ms to Long.MAX_VALUE and metadata.fetch.timeout.ms will not be honoured

Upgrading from 0.8.1 to 0.8.2

0.8.2 is fully compatible with 0.8.1. The upgrade can be done one broker at a time by simply bringing it down, updating the code, and restarting it.

Upgrading from 0.8.0 to 0.8.1

0.8.1 is fully compatible with 0.8. The upgrade can be done one broker at a time by simply bringing it down, updating the code, and restarting it.

Upgrading from 0.7

Release 0.7 is incompatible with newer releases. Major changes were made to the API, ZooKeeper data structures, and protocol, and configuration in order to add replication (Which was missing in 0.7). The upgrade from 0.7 to later versions requires a **special tool** for migration. This migration can be done without downtime.

2. API

Apache Kafka 包含了新的 Java 客户端（在 org.apache.kafka.clients package 包）。它的目的是取代原来的 Scala 客户端，但是为了兼容它们将并存一段时间。老的 Scala 客户端还打包在服务器中，这些客户端在不同的 jar 保证并包含着最小的依赖。

2.1 生产者 API Producer API

我们鼓励所有新的开发都使用新的 Java 生产者。这个客户端经过了生产环境测试并且通常情况它比原来 Scals 客户端更加快速、功能更加齐全。你可以通过添加以下示例的 Maven 坐标到客户端依赖中来使用这个新的客户端（你可以修改版本号来使用新的发布版本）：

```
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.0.0</version>
    </dependency>
```

生产者的使用演示可以在这里找到 **javadocs**。

对老的 Scala 生产者 API 感兴趣的人，可以在这里 找到相关信息。

2.2 消费者 API

在 0.9.0 发布时我们添加了一个新的 Java 消费者来取代原来的上层的（high-level）基于 ZooKeeper 的消费者和底层的（low-level）消费者 API。这个客户端被认为是测试质量 (beta quality)。为了保证用户能平滑的升级，我们还会维护老的 0.8 的消费者客户端能与 0.9 的集群协作。在下面的章节中我们将分别介绍老的 0.8 消费者 API（包括上层消费者连机器和底层简单消费者）和新的 Java 消费者 API。

2.2.1 Old High Level Consumer API

```
class Consumer {
  /**
   *  Create a ConsumerConnector
   *  创建一个 ConsumerConnector
   *
   *  @param config  at the minimum, need to specify the groupid of the consumer and t
he zookeeper
   *                 connection string zookeeper.connect.
   *  配置参数最少要设置此消费者的 groupid 和 Zookeeper 的连接字符串 Zookeeper.connect
   */
  public static kafka.javaapi.consumer.ConsumerConnector createJavaConsumerConnector(C
```

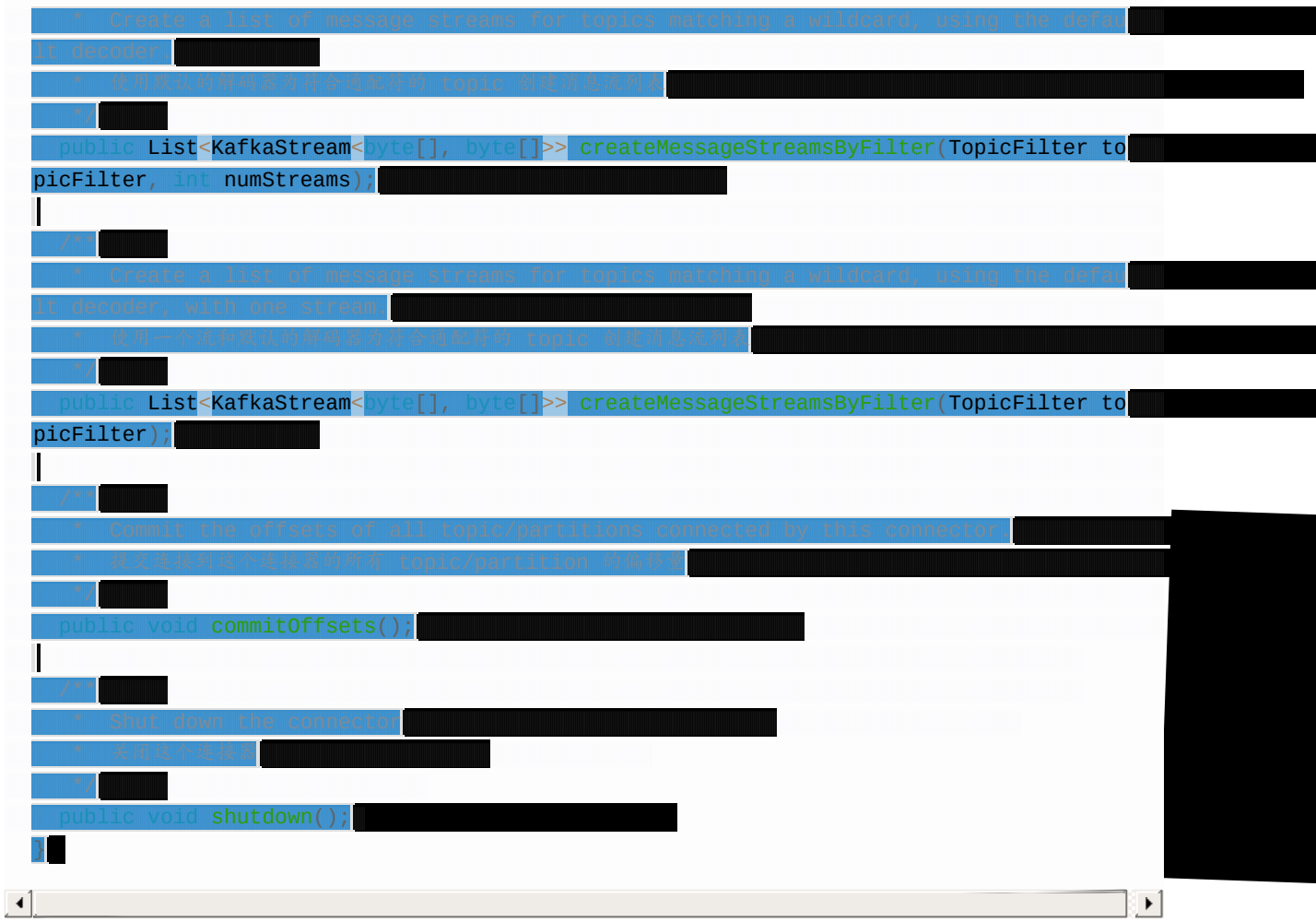
```
onsumerConfig config);
}

/**
 * V: type of the message 消息的类型
 * K: type of the optional key associated with the message 消息可选的 key 的类型
 */
public interface kafka.javaapi.consumer.ConsumerConnector {
  /**
   * Create a list of message streams of type T for each topic.
   * 为每个 topic 创建一个 T 类型的消息流
   *
   * @param topicCountMap  a map of (topic, #streams) pair
   *                       (topic, #streams) 对的 Map
   * @param decoder a decoder that converts from Message to T
   *                将消息转换为 T 类型的解码器
   * @return a map of (topic, list of  KafkaStream) pairs.
   *         The number of items in the list is #streams. Each stream supports
   *         an iterator over message/metadata pairs.
   *         返回一个 (topic, KafkaStream 列表) 对的 Map. list 的元素个数为#streams. 每个 s
tream 都支持一个对 message/metadata 对的迭代器
   */
  public <K,V> Map<String, List<KafkaStream<K,V>>>
    createMessageStreams(Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, De
coder<V> valueDecoder);

  /**
   * Create a list of message streams of type T for each topic, using the default dec
oder.
   * 使用默认的解码器为每个 topic 创建一个 T 类型的消息流
   */
  public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String
, Integer> topicCountMap);

  /**
   * Create a list of message streams for topics matching a wildcard.
   * 为符合通配符的 topics 创建一个消息流列表
   *
   * @param topicFilter a TopicFilter that specifies which topics to
   *                    subscribe to (encapsulates a whitelist or a blacklist).
   *                    指明哪些 topic 被订阅的 topic 过滤器 (封装一个白名单或者黑名单)
   * @param numStreams the number of message streams to return.
   *                   将返回的消息流的数量
   * @param keyDecoder a decoder that decodes the message key
   *                   用于解码消息键的解码器
   * @param valueDecoder a decoder that decodes the message itself
   *                     解码消息的解码器
   * @return a list of KafkaStream. Each stream supports an
   *         iterator over its MessageAndMetadata elements.
   *         KafkaStream 的列表. 每个流支持一个遍历消息元数据元素的迭代器
   */
  public <K,V> List<KafkaStream<K,V>>
    createMessageStreamsByFilter(TopicFilter topicFilter, int numStreams, Decoder<K> k
eyDecoder, Decoder<V> valueDecoder);

  /**
```



```
    *  Create a list of message streams for topics matching a wildcard, using the defau
lt decoder.
    *  使用默认的解码器为符合通配符的 topic 创建消息流列表
    */
   public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter to
picFilter, int numStreams);

   /**
    *  Create a list of message streams for topics matching a wildcard, using the defau
lt decoder, with one stream.
    *  使用一个流和默认的解码器为符合通配符的 topic 创建消息流列表
    */
   public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter to
picFilter);

   /**
    *  Commit the offsets of all topic/partitions connected by this connector.
    *  提交连接到这个连接器的所有 topic/partition 的偏移量
    */
   public void commitOffsets();

   /**
    *  Shut down the connector
    *  关闭这个连接器
    */
   public void shutdown();
}
```

你可以参见这个示例 来学习如何使用高层消费者 api。

2.2.2 老的简单消费者 API Old Simple Consumer API

```
class kafka.javaapi.consumer.SimpleConsumer {
  /**
   *  Fetch a set of messages from a topic.
   *  从一个 topic 上拉取抓取一堆消息
   *
   *  @param request specifies the topic name, topic partition, starting byte offset,
maximum bytes to be fetched.
   *          request 指定 topic 名称，topic 分区，起始的比特偏移量，最大的抓取的比特量
   *  @return a set of fetched messages
   *          抓取的消息集合
   */
  public FetchResponse fetch(kafka.javaapi.FetchRequest request);

  /**
   *  Fetch metadata for a sequence of topics.
   *  抓取一个 topic 序列的元数据
   *
   *  @param request specifies the versionId, clientId, sequence of topics.
   *          request 指明 versionId，clientId，topic 序列
   *  @return metadata for each topic in the request.
   *          request 中的每个 topic 的元数据
   */
  public kafka.javaapi.TopicMetadataResponse send(kafka.javaapi.TopicMetadataRequest r
equest);

  /**
   *  Get a list of valid offsets (up to maxSize) before the given time.
   *  获取一个在指定时间前有效偏移量（到最大数值）的列表
   *
   *  @param request a [[kafka.javaapi.OffsetRequest]] object.
   *                 一个 [[kafka.javaapi.OffsetRequest]] 对象
   *  @return a [[kafka.javaapi.OffsetResponse]] object.
   *          一个 [[kafka.javaapi.OffsetResponse]] 对象
   */
  public kafka.javaapi.OffsetResponse getOffsetsBefore(OffsetRequest request);

  /**
   * Close the SimpleConsumer.
   * 关闭 SimpleConsumer
   */
  public void close();
}
```

对于大多数应用，高层的消费者 Api 已经足够优秀了。一些应用需求的特性还没有暴露给高层消费者（比如在重启消费者时设置初始的 offset）。它们可以取代我们的底层 SimpleConsumer Api。这个逻辑可能更复杂一点，你可以参照这个示例。

2.2.3 新消费者 API New Consumer API

这个新的统一的消费者 API 移除了从 0.8 开始而来的上层和底层消费者 API 的差异。你可以通过添加如下示例 Maven 坐标来添加客户端 jar 依赖来使用此客户端。

```
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.0.0</version>
    </dependency>
```

关于消费者如何使用的示例在 **javadocs**。

2.3 Streams API

我们在 0.10.0 的发布中添加了一个新的称为**Kafka Streams**客户端类库来支持用户实现存储于 Kafka Topic 的数据的流处理程序。Kafka 流处理被认定为 alpha 阶段，它的公开 API 可能在后续的版本中变更。你可以通过添加如下的 Maven 坐标来添加流处理 jar 依赖，从而使用 Kafka 流处理（你可以改变版本为新的发布版本）：

```
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>0.10.0.0</version>
    </dependency>
```

如何使用这个类库的示例在 **javadocs** 给出（注意，被注解了**@InterfaceStability.Unstable** 的类标明他们的公开 API 可能在以后的发布中变更并不保证前向兼容）。

2. API

Apache Kafka includes new java clients (in the org.apache.kafka.clients package). These are meant to supplant the older Scala clients, but for compatibility they will co-exist for some time. These clients are available in a separate jar with minimal dependencies, while the old Scala clients remain packaged with the server.

2.1 Producer API

We encourage all new development to use the new Java producer. This client is production tested and generally both faster and more fully featured than the previous Scala client. You can use this client by adding a dependency on the client jar using the following example maven co-ordinates (you can change the version numbers with new releases):

```
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.10.0.0</version>
    </dependency>
```

Examples showing how to use the producer are given in the **javadocs**.

For those interested in the legacy Scala producer api, information can be found **here**.

2.2 Consumer API

As of the 0.9.0 release we have added a new Java consumer to replace our existing high-level ZooKeeper-based consumer and low-level consumer APIs. This client is considered beta quality. To ensure a smooth upgrade path for users, we still maintain the old 0.8 consumer clients that continue to work on an 0.9 Kafka cluster. In the following sections we introduce both the old 0.8 consumer APIs (both high-level ConsumerConnector and low-level SimpleConsumer) and the new Java consumer API respectively.

2.2.1 Old High Level Consumer API

```
class Consumer {
  /**
   *  Create a ConsumerConnector
   *
   *  @param config  at the minimum, need to specify the groupid of the consumer and t
he zookeeper
   *                 connection string zookeeper.connect.
```

```
    */
    public static kafka.javaapi.consumer.ConsumerConnector createJavaConsumerConnector(C
onsumerConfig config);
}

/**
 *  V: type of the message
 *  K: type of the optional key associated with the message
 */
public interface kafka.javaapi.consumer.ConsumerConnector {
    /**
     *  Create a list of message streams of type T for each topic
     *
     *  @param topicCountMap  a map of (topic, #streams) pair
     *  @param decoder a decoder that converts from Message to T
     *  @return a map of (topic, list of  KafkaStream) pairs.
     *          The number of items in the list is #streams. Each stream supports
     *          an iterator over message/metadata pairs.
     */
    public <K,V> Map<String, List<KafkaStream<K,V>>>
      createMessageStreams(Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, De
coder<V> valueDecoder);

    /**
     *  Create a list of message streams of type T for each topic, using the default dec
oder.
     */
    public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String
, Integer> topicCountMap);

    /**
     *  Create a list of message streams for topics matching a wildcard.
     *
     *  @param topicFilter a TopicFilter that specifies which topics to
     *                     subscribe to (encapsulates a whitelist or a blacklist).
     *  @param numStreams the number of message streams to return.
     *  @param keyDecoder a decoder that decodes the message key
     *  @param valueDecoder a decoder that decodes the message itself
     *  @return a list of KafkaStream. Each stream supports an
     *          iterator over its MessageAndMetadata elements.
     */
    public <K,V> List<KafkaStream<K,V>>
      createMessageStreamsByFilter(TopicFilter topicFilter, int numStreams, Decoder<K> k
eyDecoder, Decoder<V> valueDecoder);

    /**
     *  Create a list of message streams for topics matching a wildcard, using the defau
lt decoder.
     */
    public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter to
picFilter, int numStreams);

    /**
     *  Create a list of message streams for topics matching a wildcard, using the defau
lt decoder, with one stream.
     */
```

```
  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter to
picFilter);
  /**
   *  Commit the offsets of all topic/partitions connected by this connector
   */
  public void commitOffsets();
  /**
   *  Shut down the connector
   */
  public void shutdown();
}
```

You can follow **this example** to learn how to use the high level consumer api.

2.2.2 Old Simple Consumer API

```
class kafka.javaapi.consumer.SimpleConsumer {
  /**
   *  Fetch a set of messages from a topic.
   *
   *  @param request specifies the topic name, topic partition, starting byte offset,
maximum bytes to be fetched.
   *  @return a set of fetched messages
   */
  public FetchResponse fetch(kafka.javaapi.FetchRequest request);
  /**
   *  Fetch metadata for a sequence of topics.
   *
   *  @param request specifies the versionId, clientId, sequence of topics.
   *  @return metadata for each topic in the request.
   */
  public kafka.javaapi.TopicMetadataResponse send(kafka.javaapi.TopicMetadataRequest r
equest);
  /**
   *  Get a list of valid offsets (up to maxSize) before the given time.
   *
   *  @param request a [[kafka.javaapi.OffsetRequest]] object.
   *  @return a [[kafka.javaapi.OffsetResponse]] object.
   */
  public kafka.javaapi.OffsetResponse getOffsetsBefore(OffsetRequest request);
  /**
   * Close the SimpleConsumer.
   */
  public void close();
}
```

For most applications, the high level consumer Api is good enough. Some applications want features not exposed to the high level consumer yet (e.g., set initial offset when restarting the consumer). They can instead use our low level SimpleConsumer Api. The logic will be a bit more complicated and you can follow the example in **here**.

2.2.3 New Consumer API

This new unified consumer API removes the distinction between the 0.8 high-level and low-level consumer APIs. You can use this client by adding a dependency on the client jar using the following example maven co-ordinates (you can change the version numbers with new releases):

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

Examples showing how to use the consumer are given in the **javadocs**.

2.3 Streams API

As of the 0.10.0 release we have added a new client library named **Kafka Streams** to let users implement their stream processing applications with data stored in Kafka topics. Kafka Streams is considered alpha quality and its public APIs are likely to change in future releases. You can use Kafka Streams by adding a dependency on the streams jar using the following example maven co-ordinates (you can change the version numbers with new releases):

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

Examples showing how to use this library are given in the **javadocs** (note those classes annotated with**@InterfaceStability.Unstable**, indicating their public APIs may change without backward-compatibility in future releases).

3. 配置

Kafka 使用 **property file** 格式 键值对进行配置。这些数值可以通过文件或者编程形式指定。

3.1 Broker 配置

核心的必要配置信息如下：

- `broker.id`
- `log.dirs`
- `zookeeper.connect`

Topic 级别的配置和默认值在下面 进行更深入的讨论。

名称	描述
zookeeper.connect	Zookeeper 连接字符串 /ZK 地址
advertised.host.name	弃用：只有当'advertised.listeners'或者'listeners'没有设置时使用。使用'advertised.listeners'来取代。发布到 ZooKeeper 供客户端使用的的 Hostname IaaS 环境中，这个配置可能与 broker 绑定口不同。如果这个值没有设置那么它将使配置的'host.name'，如果'host.name'也没置它将返回 java.net.InetAddress.getCanonicalHostNa 的值。 only used when 'advertised.listeners' or 'listeners' are not set. Use 'advertised.liste instead. Hostname to publish to ZooKeep clients to use. In IaaS environments, this need to be different from the interface to the broker binds. If this is not set, it will us value for 'host.name' if configured. Otherv will use the value returned from java.net.InetAddress.getCanonicalHostNa
advertised.listeners	发布到 ZooKeeper 供客户端使用的监听器者注：broker 监听的网络接口，例如 PLAINTEXT://192.168.71.31:9092 或者 security_protocol://host_name:port），如上面配置的'listeners'不同。在 Iaas 环境，值可能要和 broker 绑定的接口不同。假如设置将使用'listeners'的值。 Listeners to publish to ZooKeeper for clie use, if different than the listeners above. I IaaS environments, this may need to be different from the interface to which the b

	binds. If this is not set, the value for 'liste will be used.
advertised.port	弃用：只有当'advertised.listeners'或者'listeners'没有配置的时候起效。使用'advertised.listeners'代替。发布到 ZooKe 供客户端使用的端口。在 IaaS 环境中，这口后可能需要和 Broker 绑定的不同。如果配置将使用 broker 绑定的端口值。 DEPRECATED: only used when 'advertised.listeners' or 'listeners' are not Use 'advertised.listeners' instead. The po publish to ZooKeeper for clients to use. In environments, this may need to be differe from the port to which the broker binds. If is not set, it will publish the same port tha broker binds to.
auto.create.topics.enable	允许在服务器上自动创建主题 (topic) Enable auto creation of topic on the serve
auto.leader.rebalance.enable	是否启动自动 leader 均衡，一个后台线程期检查然后按需触发 leader 重选 Enables auto leader balancing. A backgro thread checks and triggers leader balanc required at regular intervals
background.threads	后台处理任务使用的线程数 The number of threads to use for various background processing tasks
broker.id	这个服务器上 broker 的 id。如果不设置，成一个唯一的 broker id。为了避免基于 Zookeeper 生成的 broker id 与用户配置的冲突，生成的 broker id 从 reserved.broker.max.id + 1 开始。 The broker id for this server. If unset, a un broker id will be generated.To avoid confl between zookeeper generated broker id's user configured broker id's, generated bro idsstart from reserved.broker.max.id + 1.
compression.type	为特定的 topic 指定最终的压缩类型。这个接受标准的压缩编码 ('gzip', 'snappy', 'lz4' 外它接受'uncompressed'参数指明不进行压同时'producer'值表示保留生产者指定的压型 Specify the final compression type for a g topic. This configuration accepts the stan compression codecs ('gzip', 'snappy', 'lz4 additionally accepts 'uncompressed' whic equivalent to no compression; and 'produ which means retain the original compress codec set by the producer.

delete.topic.enable	是否允许删除 topic。如果这个配置被关闭过 admin tool 删除 topic 将失效。 Enables delete topic. Delete topic through admin tool will have no effect if this config turned off
host.name	弃用：只有当'listeners'没有配置的时候起如果这个配置被设置，它将仅仅绑定到这址，如果没有设置将绑定到所有的接口上 DEPRECATED: only used when 'listeners not set. Use 'listeners' instead. hostname broker. If this is set, it will only bind to this address. If this is not set, it will bind to all interfaces
leader.imbalance.check.interval.seconds	控制器触发分区重分配检查的周期 The frequency with which the partition rebalance check is triggered by the contr
leader.imbalance.per.broker.percentage	每个 broker 上的 leader 不均衡比例。如果 broker 上的 leader 不均衡比例超过此数值器将触发一个 leader 均衡。这个数值按照比指定。（译者注：由于 Replication 机制 Partition 都有多个副本，这个副本列表 AR 个称为 Preferred Replica，Kafka 要保证 Preferred Replica 均匀分到 broker 上，因于某个 Partition 的读写操作都由这个副本成。） The ratio of leader imbalance allowed per broker. The controller would trigger a lead balance if it goes above this value per bro The value is specified in percentage.
listeners	监听接口列表 - 一个逗号分隔的 URI 列表将使用指定的协议监听这些地址。指定 hostname 为 0.0.0.0 将绑定所有的接口。hostname 将绑定到默认的接口。实例合法听接口列表为： PLAINTEXT://myhost:9092,TRACE://:909 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093 Listener List - Comma-separated list of U we will listen on and their protocols. Spec hostname as 0.0.0.0 to bind to all interfac Leave hostname empty to bind to default interface. Examples of legal listener lists: PLAINTEXT://myhost:9092,TRACE://:909 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093
log.dir	日志文件存储的位置（log.dirs 配置的补充 The directory in which the log data is kept (supplemental for log.dirs property)
	日志文件存储的位置列表，如果没有设置

log.dirs	用 log.dir The directories in which the log data is ke not set, the value in log.dir is used
log.flush.interval.messages	在消息被刷新到磁盘之前允许在单个日志 上堆积的消息的数量 The number of messages accumulated o log partition before messages are flushed disk
log.flush.interval.ms	任意 topic 上的消息在刷新到硬盘之前允 存在内存中的最大 ms 数。如果没有设置 log.flush.scheduler.interval.ms 配置 The maximum time in ms that a message any topic is kept in memory before flushe disk. If not set, the value in log.flush.scheduler.interval.ms is used
log.flush.offset.checkpoint.interval.ms	更新最后一次 flush 的持久化消息为日志恢 的频率 The frequency with which we update the persistent record of the last flush which a the log recovery point
log.flush.scheduler.interval.ms	刷新检查是否有 log 需要被刷新到硬盘的 ms 表示 The frequency in ms that the log flusher c whether any log needs to be flushed to di
log.retention.bytes	保留的日志最大的大小 The maximum size of the log before delet
log.retention.hours	在删除之前日志文件保存的最长时间（小 计），相比 log.retention.ms 第三优先 The number of hours to keep a log file be deleting it (in hours), tertiary to log.retenti property
log.retention.minutes	在删除之前日志文件保存的最长时间（分 计），相比 log.retention.ms 第二优先。如 有设置 log.retention.hours 中的值将被使用 The number of minutes to keep a log file deleting it (in minutes), secondary to log.retention.ms property. If not set, the v in log.retention.hours is used
log.retention.ms	在删除之前日志文件保存的最长微秒数（ 计），如果没有设置 log.retention.minutes 值将被使用 The number of milliseconds to keep a log before deleting it (in milliseconds), If not s the value in log.retention.minutes is used
log.roll.hours	切分新日志段的最长时间间隔（小时计） 比 log.roll.ms 第二优先 The maximum time before a new log segm

	is rolled out (in hours), secondary to log.r property
log.roll.jitter.hours	切分新日志段时间的随机偏移量（小时计 相比 log.roll.jitter.ms 第二优先级（译者注 加时间随机偏移量是为了防止惊群问题。 The maximum jitter to subtract from logRollTimeMillis (in hours), secondary to log.roll.jitter.ms property
log.roll.jitter.ms	切分新日志段时间的随机偏移量（毫秒计 未设置则使用 log.roll.jitter.hours The maximum jitter to subtract from logRollTimeMillis (in milliseconds). If not s the value in log.roll.jitter.hours is used
log.roll.ms	切分新日志段的最长时间间隔（毫秒计） 果不设置则使用 log.roll.hours The maximum time before a new log segr is rolled out (in milliseconds). If not set, th value in log.roll.hours is used
log.segment.bytes	单个日志文件最大大小 The maximum size of a single log file
log.segment.delete.delay.ms	日志分段从文件系统删除前的延迟 The amount of time to wait before deletin file from the filesystem
message.max.bytes	服务器可接受的最大消息的大小 The maximum size of message that the s can receive
min.insync.replicas	定义 ACK 生产者请求前 ISR 中最小的副本 define the minimum number of replicas in needed to satisfy a produce request with acks=all (or -1)
num.io.threads	服务器为网络请求服务的 IO 线程数 The number of io threads that the server for carrying out network requests
num.network.threads	服务器处理网络请求的网络线程数 the number of network threads that the se uses for handling network requests
num.recovery.threads.per.data.dir	每个日志文件夹在启停时日志恢复使用的 数（译者注：RAID 可以增大数值） The number of threads per data directory used for log recovery at startup and flush shutdown
num.replica.fetchers	从源节点抓取副本消息的抓取线程数。这 越高从节点的 IO 同步性越好 Number of fetcher threads used to replica messages from a source broker. Increasi

	this value can increase the degree of I/O parallelism in the follower broker.
offset.metadata.max.bytes	offset 队列中元数据的最大大小（译者注：前版本，Kafka 其实存在一个比较大的隐患是利用 Zookeeper 来存储记录每个消费者的消费进度。从 0.10.1.1 版本已默认将消 offset 迁入到了 Kafka 一个名为 `__consumer_offsets` 的 Topic 中。这个 To 使用 group、topic、partition 三元组维护 offset 信息。） The maximum size for a metadata entry associated with an offset commit
offsets.commit.required.acks	offset 队列 commit 时需要的 ack 数量。一 况下默认值 -1 不应该被修改（译者注：因 部分消息是非常重要，以至于是不能容忍 据的，生产者要等到所有的 ISR 都收到消 才会得到 ack。数据安全性好自然其速度会 影响。） The required acks before the commit can accepted. In general, the default (-1) shou not be overridden
offsets.commit.timeout.ms	offset 队列接受 commit 的超时时间。这个 与生产者请求超时时间相似 Offset commit will be delayed until all rep for the offsets topic receive the commit or timeout is reached. This is similar to the producer request timeout.
offsets.load.buffer.size	从 offset 日志段读取信息到缓存的批量加 小 Batch size for reading from the offsets segments when loading offsets into the ca
offsets.retention.check.interval.ms	检查旧的偏移量的频率，单位是毫秒 Frequency at which to check for stale offs
offsets.retention.minutes	偏移量主题的日志保留窗口，单位是分钟 Log retention window in minutes for offse topic
offsets.topic.compression.codec	offsets topic 的压缩编码，压缩编码可以用 子化提交 Compression codec for the offsets topic - compression may be used to achieve "ato commits
offsets.topic.num.partitions	offset topic 的分区数量（在部署后不应该 改） The number of partitions for the offset con topic (should not change after deploymen
	offsets topic 的复制因子（设置高一些要保 用性）。为了保证 offsets topic 的复制因

offsets.topic.replication.factor	效，在 offsets topic 处理第一个请求的时候 活的节点数必须大于或者等于复制因子。 offsets topic 创建将会失败或者使用两者之 最小值（存活节点数、配置的复制因子） The replication factor for the offsets topic higher to ensure availability). To ensure th the effective replication factor of the offse topic is the configured value, the number alive brokers has to be at least the replica factor at the time of the first request for th offsets topic. If not, either the offsets topic creation will fail or it will get a replication f of min(alive brokers, configured replicatio factor)
offsets.topic.segment.bytes	offsets topic 的段大小应该设置的相对较小 证快速的日志压缩和缓存加载 The offsets topic segment bytes should b relatively small in order to facilitate faster compaction and cache loads
port	已过时：仅仅当”listeners“没有被配置时有 使用”listeners“来取代此配置。监听和接受 的端口 DEPRECATED: only used when 'listeners not set. Use 'listeners' instead. the port to and accept connections on
queued.max.requests	在阻塞网络线程之前允许的请求队列最大 度 The number of queued requests allowed before blocking the network threads
quota.consumer.default	按照 clientId/consumer group 定义的任何 者每秒允许拉取的限额（byte） Any consumer distinguished by clientId/consumer group will get throttled fetches more bytes than this value per-se
quota.producer.default	按照 clientId 定义的生产者每秒允许生产的 （byte） Any producer distinguished by clientId wil throttled if it produces more bytes than th value per-second
replica.fetch.max.bytes	允许复制者拉取消息集的最大小小 The number of bytes of messages to atte fetch
replica.fetch.min.bytes	复制者拉取消息响应最小的大小。如果不 大小响应延迟到 replicaMaxWaitTimeMs 这 （译者注：防止小包问题） Minimum bytes expected for each fetch response. If not enough bytes, wait up to replicaMaxWaitTimeMs

replica.fetch.wait.max.ms	复制者拉取消息响应最大的等待时间。这 置必须小于 replica.lag.time.max.ms 来防 吞吐量 Topic 出现经常性的 ISR 丢失 max wait time for each fetcher request iss by follower replicas. This value should alv be less than the replica.lag.time.max.ms times to prevent frequent shrinking of ISR low throughput topics
replica.high.watermark.checkpoint.interval.ms	replica 将最高水位进行 flush 到磁盘的时 The frequency with which the high waterr is saved out to disk
replica.lag.time.max.ms	如果在这个时间点前复制者没有发送任何 请求或者复制者没有消费到 Leader 日志最 移量，Leader 将会将此复制者从 ISR 中移 If a follower hasn't sent any fetch request hasn't consumed up to the leaders log en offset for at least this time, the leader will remove the follower from isr
replica.socket.receive.buffer.bytes	网络请求的 Socket 接收缓存 The socket receive buffer for network req
replica.socket.timeout.ms	网络请求的 Socket 超时时间。这个值应该 大于 replica.fetch.wait.max.ms The socket timeout for network requests. value should be at least replica.fetch.wait.max.ms
request.timeout.ms	这个配置控制着客户端等待请求响应的最 间。如果在这个时间之后依旧没有收到响 客户端将按需重发或者当禁用重试时直接 请求失败 The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is received before the timeout elapses the c will resend the request if necessary or fai request if retries are exhausted.
socket.receive.buffer.bytes	The SO_RCVBUF buffer of the socket se sockets
socket.request.max.bytes	The maximum number of bytes in a sock request
socket.send.buffer.bytes	The SO_SNDBUF buffer of the socket se sockets
unclean.leader.election.enable	Indicates whether to enable replicas not i ISR set to be elected as leader as a last r even though doing so may result in data l
zookeeper.connection.timeout.ms	The max time that the client waits to esta a connection to zookeeper. If not set, the

	in zookeeper.session.timeout.ms is used
zookeeper.session.timeout.ms	Zookeeper session timeout
zookeeper.set.acl	Set client to use secure ACLs
broker.id.generation.enable	Enable automatic broker id generation on server? When enabled the value configur reserved.broker.max.id should be reviewe
broker.rack	Rack of the broker. This will be used in ra aware replication assignment for fault tolerance. Examples: 'RACK1', 'us-east-1
connections.max.idle.ms	Idle connections timeout: the server sock processor threads close the connections idle more than this
controlled.shutdown.enable	Enable controlled shutdown of the server
controlled.shutdown.max.retries	Controlled shutdown can fail for multiple reasons. This determines the number of r when such failure happens
controlled.shutdown.retry.backoff.ms	Before each retry, the system needs time recover from the state that caused the pr failure (Controller fail over, replica lag etc config determines the amount of time to w before retrying.
controller.socket.timeout.ms	The socket timeout for controller-to-broke channels
default.replication.factor	default replication factors for automaticall created topics
fetch.purgatory.purge.interval.requests	The purge interval (in number of requests the fetch request purgatory
group.max.session.timeout.ms	The maximum allowed session timeout fo registered consumers. Longer timeouts g consumers more time to process messag between heartbeats at the cost of a longe to detect failures.
group.min.session.timeout.ms	The minimum allowed session timeout fo registered consumers. Shorter timeouts l to quicker failure detection at the cost of r frequent consumer heartbeating, which ca overwhelm broker resources.
inter.broker.protocol.version	Specify which version of the inter-broker protocol will be used. This is typically bun after all brokers were upgraded to a new version. Example of some valid values ar 0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2. 0.9.0.0, 0.9.0.1 Check ApiVersion for the

	list.
log.cleaner.backoff.ms	The amount of time to sleep when there a logs to clean
log.cleaner.dedupe.buffer.size	The total memory used for log deduplicat across all cleaner threads
log.cleaner.delete.retention.ms	How long are delete records retained?
log.cleaner.enable	Enable the log cleaner process to run on server? Should be enabled if using any to with a cleanup.policy=compact including internal offsets topic. If disabled those top will not be compacted and continually gro size.
log.cleaner.io.buffer.load.factor	Log cleaner dedupe buffer load factor. Th percentage full the dedupe buffer can bec A higher value will allow more log to be cleaned at once but will lead to more has collisions
log.cleaner.io.buffer.size	The total memory used for log cleaner I/C buffers across all cleaner threads
log.cleaner.io.max.bytes.per.second	The log cleaner will be throttled so that th of its read and write i/o will be less than th value on average
log.cleaner.min.cleanable.ratio	The minimum ratio of dirty log to total log log to eligible for cleaning
log.cleaner.threads	The number of background threads to us log cleaning
log.cleanup.policy	The default cleanup policy for segments beyond the retention window, must be eit "delete" or "compact"
log.index.interval.bytes	The interval with which we add an entry t offset index
log.index.size.max.bytes	The maximum size in bytes of the offset i
log.message.format.version	Specify the message format version the b will use to append messages to the logs. value should be a valid ApiVersion. Some examples are: 0.8.2, 0.9.0.0, 0.10.0, chec ApiVersion for more details. By setting a particular message format version, the us certifying that all the existing messages o are smaller or equal than the specified ve Setting this value incorrectly will cause consumers with older versions to break a will receive messages with a format that t don't understand.

log.message.timestamp.difference.max.ms	The maximum difference allowed betwee timestamp when a broker receives a mes and the timestamp specified in the messa message.timestamp.type=CreateTime, a message will be rejected if the difference timestamp exceeds this threshold. This configuration is ignored if message.timestamp.type=LogAppendTim
log.message.timestamp.type	Define whether the timestamp in the mes is message create time or log append tim The value should be either 'CreateTime' o 'LogAppendTime'
log.preallocate	Should pre allocate file when create new segment? If you are using Kafka on Wind you probably need to set it to true.
log.retention.check.interval.ms	The frequency in milliseconds that the log cleaner checks whether any log is eligible deletion
max.connections.per.ip	The maximum number of connections we from each ip address
max.connections.per.ip.overrides	Per-ip or hostname overrides to the defau maximum number of connections
num.partitions	The default number of log partitions per t
principal.builder.class	The fully qualified name of a class that implements the PrincipalBuilder interface which is currently used to build the Princi connections with the SSL SecurityProtoc
producer.purgatory.purge.interval.requests	The purge interval (in number of requests the producer request purgatory
replica.fetch.backoff.ms	The amount of time to sleep when fetch partition error occurs.
reserved.broker.max.id	Max number that can be used for a broke
sasl.enabled.mechanisms	The list of SASL mechanisms enabled in Kafka server. The list may contain any mechanism for which a security provider available. Only GSSAPI is enabled by de
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.
	A list of rules for mapping from principal r to short names (typically operating syster usernames). The rules are evaluated in o and the first rule that matches a principal

sasl.kerberos.principal.to.local.rules	is used to map it to a short name. Any lat rules in the list are ignored. By default, principal names of the form {username}/{hostname}@{REALM} are mapped to {username}. For more details the format please see **security authoriza and acls**.
sasl.kerberos.service.name	The Kerberos principal name that Kafka r as. This can be defined either in Kafka's J config or in Kafka's config.
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which it will try to renew the ticket.
sasl.mechanism.inter.broker.protocol	SASL mechanism used for inter-broker communication. Default is GSSAPI.
security.inter.broker.protocol	Security protocol used to communicate between brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption and key exchange algorithm used to nego the security settings for a network connec using TLS or SSL network protocol.By de all the available cipher suites are support
ssl.client.auth	Configures kafka broker to request client authentication. The following settings are common: `ssl.client.auth=required` If set required client authentication is required. `ssl.client.auth=requested` This means client authentication is optional. u requested , if this option is set client can choose not to provide authentication information about itself `ssl.client.auth=n` This means client authentication is not ne
ssl.enabled.protocols	The list of protocols enabled for SSL connections.
ssl.key.password	The password of the private key in the ke store file. This is optional for client.
ssl.keymanager.algorithm	The algorithm used by key manager facto SSL connections. Default value is the key manager factory algorithm configured for Java Virtual Machine.

ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two authentication for client.
ssl.keystore.password	The store password for the key store file. optional for client and only needed if ssl.keystore.location is configured.
ssl.keystore.type	The file format of the key store file. This is optional for client.
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which fine for most cases. Allowed values in rec JVMs are TLS, TLSv1.1 and TLSv1.2. SS SSLv2 and SSLv3 may be supported in c JVMs, but their usage is discouraged due known security vulnerabilities.
ssl.provider	The name of the security provider used fo connections. Default value is the default security provider of the JVM.
ssl.trustmanager.algorithm	The algorithm used by trust manager fact for SSL connections. Default value is the manager factory algorithm configured for Java Virtual Machine.
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
ssl.truststore.type	The file format of the trust store file.
authorizer.class.name	The authorizer class that should be used authorization
metric.reporters	A list of classes to use as metrics reporte Implementing the `MetricReporter` interfac allows plugging in classes that will be not of new metric creation. The JmxReporter always included to register JMX statistics
metrics.num.samples	The number of samples maintained to co metrics.
metrics.sample.window.ms	The window of time a metrics sample is computed over.
quota.window.num	The number of samples to retain in memo
quota.window.size.seconds	The time span of each sample
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to va server hostname using server certificate.
zookeeper.sync.time.ms	How far a ZK follower can be behind a ZK leader

更多关于 broker 的配置信息可以在 scala 类 `kafka.server.KafkaConfig` 中找到。

Topic 级别配置 与 Topic 相关的配置既包含服务器默认值，也包含可选的每个 Topic 覆盖值。如果没有给出 Topic 单独的配置，那么服务器全局配置就会被使用。Topic 的配置可以在创建时通过一个或者多个 `--config` 选项来提供。本示例使用自定义的最大消息大小和刷新率创建一个名为 *my-topic* 的 topic:

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --partitio
ns 1
        --replication-factor 1 --config max.message.bytes=64000 --config flush.message
s=1
```

也可以在使用 `alter configs` 命令稍后更改或设置覆盖值。本示例重置 *my-topic* 的最大消息的大小：

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic my-topic
    --config max.message.bytes=128000
```

移除一个设置可以通过如下方法：

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic my-topic
    --delete-config max.message.bytes
```

下面是 Topic 级别配置，服务器默认属性列是该属性的默认配置，修改该属性的服务器级别设置可以用来改变 Topic 的属性。

名称	默认值	服务器默认属性	
cleanup.policy	delete	log.cleanup.policy	A string that is "compact". Thi the retention p log segments. ("delete") will d segments whe time or size lim reached. The " will enable log the topic.
delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms	The amount of delete tombsto compacted to also gives a bo which a consu a read if they b to ensure that snapshot of th

			(otherwise dele may be collect complete their
flush.messages	None	log.flush.interval.messages	This setting all interval at whic fsync of data w For example if we would fsyn message; if it v fsync after eve In general we r not set this and durability and a system's backg capabilities as This setting ca a per-topic bas **topic configu**
flush.ms	None	log.flush.interval.ms	This setting all time interval at force an fsync the log. For ex set to 1000 we 1000 ms had p we recommen and use replica and allow the c background flu is more efficien
index.interval.bytes	4096	log.index.interval.bytes	This setting co frequently Kafk entry to it's offs default setting index a messa 4096 bytes. M reads to jump position in the index larger. Y need to chang
max.message.bytes	1,000,000	message.max.bytes	This is largest Kafka will allow to this topic. N increase this s increase your c size so they ca this large.
			This configurat frequently the attempt to clea

min.cleanable.dirty.ratio	0.5	log.cleaner.min.cleanable.ratio	(assuming log enabled). By d cleaning a log 50% of the log compacted. Th maximum spac log by duplicat 50% of the log duplicates). A l mean fewer, m cleanings but v wasted space
min.insync.replicas	1	min.insync.replicas	When a produ "all", min.insyn the minimum n that must ackn the write to be successful. If t cannot be met will raise an ex NotEnoughRe NotEnoughRe When used tog min.insync.rep allow you to en durability guara scenario would topic with a rep set min.insync produce with a will ensure tha raises an exce of replicas do n
retention.bytes	None	log.retention.bytes	This configurat maximum size before we will segments to fr are using the " policy. By defa limit only a tim
retention.ms	7 days	log.retention.minutes	This configurat maximum time before we will segments to fr are using the " policy. This rep how soon cons their data.
			This configurat segment file si

segment.bytes	1 GB	log.segment.bytes	Retention and done a file at a segment size r but less granul retention.
segment.index.bytes	10 MB	log.index.size.max.bytes	This configurat size of the inde offsets to file p preallocate this shrink it only a generally shou change this se
segment.ms	7 days	log.roll.hours	This configurat period of time will force the lo segment file is that retention c compact old da
segment.jitter.ms	0	log.roll.jitter.{ms,hours}	The maximum from logRollTir

3.2 Producer 配置

下面是 Java 生产者的配置：

名称	描述
bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective c which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...` . Since these servers are just used for the initial connection t discover the full cluster membership (which ma change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).
key.serializer	Serializer class for key that implements the `Serializer` interface.
value.serializer	Serializer class for value that implements the `Serializer` interface.
	The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The

acks	following settings are common: `acks=0` If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the `retries` configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1. `acks=1` This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost. `acks=all` This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.
buffer.memory	The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will block for `max.block.ms` after which it will throw an exception.This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.
compression.type	The compression type for all data generated by the producer. The default is none (i.e. no compression). Valid values are `none` , `gzip` , `snappy` , or `lz4` . Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression).
retries	Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries without setting `max.in.flight.requests.per.connection` to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the

	second succeeds, then the records in the second batch may appear first.
ssl.key.password	The password of the private key in the key stor file. This is optional for client.
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.
ssl.keystore.password	The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
batch.size	The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes.No attempt will be made to batch records larger than this size.Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/por by allowing a logical application name to be included in server-side request logging.
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.
	The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding small amount of artificial delay—that is, rather than immediately sending out a record the producer will wait for up to the given delay to allow other records to be sent so that the send

linger.ms	can be batched together. This can be thought o as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get `batch.size` worth of records for a partition it will be sent immediatel regardless of this setting, however if we have fewer than this many bytes accumulated for thi partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting `linger.ms=5` , for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absense of load.
max.block.ms	The configuration controls how long `KafkaProducer.send()` and `KafkaProducer.partitionsFor()` will block.These methods can be blocked either because the buffer is full or metadata unavailable.Blocking in the user-supplied serializers or partitioner will not be counted against this timeout.
max.request.size	The maximum size of a request in bytes. This i also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.
partitioner.class	Partitioner class that implements the `Partitioner` interface.
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.
	Protocol used to communicate with brokers.

security.protocol	Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF to use when sending data.
ssl.enabled.protocols	The list of protocols enabled for SSL connections.
ssl.keystore.type	The file format of the key store file. This is optional for client.
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
ssl.truststore.type	The file format of the trust store file.
timeout.ms	The configuration controls the maximum amount of time the server will wait for acknowledgments from followers to meet the acknowledgment requirements the producer has specified with the `acks` configuration. If the requested number of acknowledgments are not met when the timeout elapses an error will be returned. This timeout is measured on the server side and does not include the network latency of the request.
block.on.buffer.full	When our memory buffer is exhausted we must either stop accepting new records (block) or throw errors. By default this setting is false and the producer will no longer throw a BufferExhaustException but instead will use the `max.block.ms` value to block, after which it will throw a TimeoutException. Setting this property to true will set the `max.block.ms` to Long.MAX*VALUE. Also if this property is set to true, parameter __ `metadata.fetch.timeout.ms`* is not longer honored._This parameter is deprecated and will be removed in a future release. Parameter `max.block.ms` should be used instead.
interceptor.classes	A list of classes to use as interceptors. Implementing the `ProducerInterceptor` interface allows you to intercept (and possibly mutate) the records received by the producer

	before they are published to the Kafka cluster. By default, there are no interceptors.
max.in.flight.requests.per.connection	The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).
metadata.fetch.timeout.ms	The first time data is sent to a topic we must fetch metadata about that topic to know which servers host the topic's partitions. This fetch to succeed before throwing an exception back to the client.
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified o new metric creation. The JmxReporter is alway included to register JMX statistics.
metrics.num.samples	The number of samples maintained to compute metrics.
metrics.sample.window.ms	The window of time a metrics sample is computed over.
reconnect.backoff.ms	The amount of time to wait before attempting t reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.
retry.backoff.ms	The amount of time to wait before attempting t retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket expiry has been reached, at which time it will tr

	to renew the ticket.
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

For those interested in the legacy Scala producer configs, information can be found **here**.

3.3 Consumer Configs

We introduce both the old 0.8 consumer configs and the new consumer configs respectively below.

3.3.1 Old Consumer Configs

The essential old consumer configurations are the following:

- `group.id`
- `zookeeper.connect`

PropertyDefaultDescription		
group.id		A string that uniquely identifies the group of co to which this consumer belongs. By setting the multiple processes indicate that they are all pa consumer group.
zookeeper.connect		Specifies the ZooKeeper connection string in t `hostname:port` where host and port are the ho ZooKeeper server. To allow connecting throug nodes when that ZooKeeper machine is down specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:por` may also have a ZooKeeper chroot path as pa ZooKeeper connection string which puts its da path in the global ZooKeeper namespace. If so

		should use the same chroot path in its connec example to give a chroot path of `/chroot/path` the connection string as `hostname1:port1,hostname2:port2,hostname3:`
consumer.id	null	Generated automatically if not set.
socket.timeout.ms	30 * 1000	The socket timeout for network requests. The will be max.fetch.wait + socket.timeout.ms.
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests
fetch.message.max.bytes	1024 * 1024	The number of bytes of messages to attempt t topic-partition in each fetch request. These byt into memory for each partition, so this helps co used by the consumer. The fetch request size as large as the maximum message size the se it is possible for the producer to send message consumer can fetch.
num.consumer.fetchers	1	The number fetcher threads used to fetch data
auto.commit.enable	true	If true, periodically commit to ZooKeeper the o already fetched by the consumer. This commit used when the process fails as the position fro consumer will begin.
auto.commit.interval.ms	60 * 1000	The frequency in ms that the consumer offsets zookeeper.
queued.max.message.chunks	2	Max number of message chunks buffered for c chunk can be up to fetch.message.max.bytes.
rebalance.max.retries	4	When a new consumer joins a consumer grou consumers attempt to "rebalance" the load to a each consumer. If the set of consumers chang assignment is taking place the rebalance will fa setting controls the maximum number of attem up.
fetch.min.bytes	1	The minimum amount of data the server shoul request. If insufficient data is available the requ that much data to accumulate before answerin
fetch.wait.max.ms	100	The maximum amount of time the server will b answering the fetch request if there isn't suffici immediately satisfy fetch.min.bytes
rebalance.backoff.ms	2000	Backoff time between retries during rebalance explicitly, the value in zookeeper.sync.time.ms
refresh.leader.backoff.ms	200	Backoff time to wait before trying to determine partition that has just lost its leader.
auto.offset.reset	largest	What to do when there is no initial offset in Zoo offset is out of range:

- smallest : automatically reset the offset to the smallest offset
- largest : automatically reset the offset to the largest offset
- anything else: throw exception to the consumer | | consumer.timeout.ms | -1 | Throw a timeout exception to the consumer if no message is available for consumption after the specified interval | | exclude.internal.topics | true | Whether messages from internal topics (such as offsets) should be exposed to the consumer. | | client.id | group id value | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. | | zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur. | | zookeeper.connection.timeout.ms | 6000 | The max time that the client waits while establishing a connection to zookeeper. | | zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader | | offsets.storage | zookeeper | Select where offsets should be stored (zookeeper or kafka). | | offsets.channel.backoff.ms | 1000 | The backoff period when reconnecting the offsets channel or retrying failed offset fetch/commit requests. | | offsets.channel.socket.timeout.ms | 10000 | Socket timeout when reading responses for offset fetch/commit requests. This timeout is also used for ConsumerMetadata requests that are used to query for the offset manager. | | offsets.commit.max.retries | 5 | Retry the offset commit up to this many times on failure. This retry count only applies to offset commits during shut-down. It does not apply to commits originating from the auto-commit thread. It also does not apply to attempts to query for the offset coordinator before committing offsets. i.e., if a consumer metadata request fails for any reason, it will be retried and that retry does not count toward this limit. | | dual.commit.enabled | true | If you are using "kafka" as offsets.storage, you can dual commit offsets to ZooKeeper (in addition to Kafka). This is required during migration from zookeeper-based offset storage to kafka-based offset storage. With respect to any given consumer group, it is safe to turn this off after all instances within that group have been migrated to the new version that commits offsets to the broker (instead of directly to ZooKeeper). | | partition.assignment.strategy | range | Select between the "range" or "roundrobin" strategy for assigning partitions to consumer streams.The round-robin partition assignor lays out all the available partitions and all the available consumer threads. It then proceeds to do a round-robin assignment from partition to consumer thread. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts will be within a delta of exactly one across all consumer threads.) Round-robin assignment is permitted only if: (a) Every topic has the same number of streams within a consumer instance (b) The set of subscribed topics is identical for every consumer instance within the group.Range partitioning works on a per-topic basis. For each topic, we lay out the available partitions in numeric order and the consumer threads in lexicographic order.

We then divide the number of partitions by the total number of consumer streams (threads) to determine the number of partitions to assign to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition. |

More details about consumer configuration can be found in the scala class `kafka.consumer.ConsumerConfig`.

3.3.2 New Consumer Configs

Since 0.9.0.0 we have been working on a replacement for our existing simple and high-level consumers. The code is considered beta quality. Below is the configuration for the new consumer:

NameDescriptionTypeDefaultValid ValuesImportance	
bootstrap.servers	A list of host/port pairs to use for establishing tl to the Kafka cluster. The client will make use o irrespective of which servers are specified here this list only impacts the initial hosts used to dis servers. This list should be in the form `host1:port1,host2:port2,...`. Since these used for the initial connection to discover the fu membership (which may change dynamically), contain the full set of servers (you may want m though, in case a server is down).
key.deserializer	Deserializer class for key that implements the `Deserializer` interface.
value.deserializer	Deserializer class for value that implements th `Deserializer` interface.
fetch.min.bytes	The minimum amount of data the server shoul request. If insufficient data is available the requ much data to accumulate before answering the setting of 1 byte means that fetch requests are as a single byte of data is available or the fetch waiting for data to arrive. Setting this to someth will cause the server to wait for larger amounts accumulate which can improve server through some additional latency.
group.id	A unique string that identifies the consumer gro belongs to. This property is required if the cons group management functionality by using `subs` Kafka-based offset management strategy.
heartbeat.interval.ms	The expected time between heartbeats to the c when using Kafka's group management facilitie used to ensure that the consumer's session sta facilitate rebalancing when new consumers joi

	The value must be set lower than `session.time` should be set no higher than 1/3 of that value. even lower to control the expected time for nor
max.partition.fetch.bytes	The maximum amount of data per-partition the The maximum total memory used for a reques `* max.partition.fetch.bytes` . This size must be the maximum message size the server allows for the producer to send messages larger than fetch. If that happens, the consumer can get st large message on a certain partition.
session.timeout.ms	The timeout used to detect failures when using management facilities. When a consumer's hea received within the session timeout, the broker consumer as failed and rebalance the group. S sent only when poll() is invoked, a higher sessi more time for message processing in the cons the cost of a longer time to detect hard failures `max.poll.records` for another option to control in the poll loop. Note that the value must be in as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max`
ssl.key.password	The password of the private key in the key stor for client.
ssl.keystore.location	The location of the key store file. This is option be used for two-way authentication for client.
ssl.keystore.password	The store password for the key store file.This i and only needed if ssl.keystore.location is conf
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
auto.offset.reset	What to do when there is no initial offset in Kaf offset does not exist any more on the server (e has been deleted):earliest: automatically reset earliest offsetlatest: automatically reset the offs offsetnone: throw exception to the consumer if found for the consumer's groupanything else: t the consumer.
connections.max.idle.ms	Close idle connections after the number of mill this config.
enable.auto.commit	If true the consumer's offset will be periodically background.
exclude.internal.topics	Whether records from internal topics (such as exposed to the consumer. If set to `true` the o records from an internal topic is subscribing to
max.poll.records	The maximum number of records returned in a

partition.assignment.strategy	The class name of the partition assignment str will use to distribute partition ownership among instances when group management is used
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVB reading data.
request.timeout.ms	The configuration controls the maximum amou will wait for the response of a request. If the re received before the timeout elapses the client request if necessary or fail the request if retries
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs a defined either in Kafka's JAAS config or in Kafk
sasl.mechanism	SASL mechanism used for client connections. mechanism for which a security provider is ava default mechanism.
security.protocol	Protocol used to communicate with brokers. Va PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF sending data.
ssl.enabled.protocols	The list of protocols enabled for SSL connectio
ssl.keystore.type	The file format of the key store file. This is opti
ssl.protocol	The SSL protocol used to generate the SSLCo is TLS, which is fine for most cases. Allowed va are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 a supported in older JVMs, but their usage is dis known security vulnerabilities.
ssl.provider	The name of the security provider used for SSL Default value is the default security provider of
ssl.truststore.type	The file format of the trust store file.
auto.commit.interval.ms	The frequency in milliseconds that the consum committed to Kafka if `enable.auto.commit` is se
check.crcs	Automatically check the CRC32 of the records ensures no on-the-wire or on-disk corruption to occurred. This check adds some overhead, so cases seeking extreme performance.
client.id	An id string to pass to the server when making purpose of this is to be able to track the source just ip/port by allowing a logical application nam server-side request logging.
fetch.max.wait.ms	The maximum amount of time the server will bl answering the fetch request if there isn't sufficie immediately satisfy the requirement given by fe
	A list of classes to use as interceptors. Implem

interceptor.classes	the `ConsumerInterceptor` interface allows you t possibly mutate) records received by the consu there are no interceptors.
metadata.max.age.ms	The period of time in milliseconds after which v metadata even if we haven't seen any partition to proactively discover any new brokers or part
metric.reporters	A list of classes to use as metrics reporters. Im the `MetricReporter` interface allows plugging ir notified of new metric creation. The JmxReport to register JMX statistics.
metrics.num.samples	The number of samples maintained to compute
metrics.sample.window.ms	The window of time a metrics sample is compu
reconnect.backoff.ms	The amount of time to wait before attempting t given host. This avoids repeatedly connecting loop. This backoff applies to all requests sent b the broker.
retry.backoff.ms	The amount of time to wait before attempting t request to a given topic partition. This avoids r requests in a tight loop under some failure sce
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attem
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renew
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified windo last refresh to ticket's expiry has been reached try to renew the ticket.
ssl.cipher.suites	A list of cipher suites. This is a named combina authentication, encryption, MAC and key excha to negotiate the security settings for a network TLS or SSL network protocol.By default all the suites are supported.
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate using server certificate.
ssl.keymanager.algorithm	The algorithm used by key manager factory for Default value is the key manager factory algori the Java Virtual Machine.
ssl.trustmanager.algorithm	The algorithm used by trust manager factory fo Default value is the trust manager factory algor the Java Virtual Machine.

3.4 Kafka Connect Configs

Below is the configuration of the Kafka Connect framework.

NameDescriptionTypeDefaultValid ValuesImportance		
config.storage.topic	kafka topic to store configs	string
group.id	A unique string that identifies the Connect cluster group this worker belongs to.	string
internal.key.converter	Converter class for internal key Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.	class
internal.value.converter	Converter class for offset value Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.	class
key.converter	Converter class for key Connect data that implements the `Converter` interface.	class
offset.storage.topic	kafka topic to store connector offsets in	string
status.storage.topic	kafka topic to track connector and task status	string
value.converter	Converter class for value Connect data that implements the `Converter` interface.	class
bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list
	ID for this cluster, which is used to	

cluster	provide a namespace so multiple Kafka Connect clusters or instances may co-exist while sharing a single Kafka cluster.	string
heartbeat.interval.ms	The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int
session.timeout.ms	The timeout used to detect failures when using Kafka's group management facilities.	int
ssl.key.password	The password of the private key in the key store file. This is optional for client.	passwo
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string
ssl.keystore.password	The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.	passwo
ssl.truststore.location	The location of the trust store file.	string
ssl.truststore.password	The password for the trust store file.	passwo
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	int
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not	int

request.timeout.ms	received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data.	int
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list
ssl.keystore.type	The file format of the key store file. This is optional for client.	string
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string
ssl.truststore.type	The file format of the trust store file.	string
worker.sync.timeout.ms	When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before	int

	waiting a backoff period before rejoining.	
worker.unsync.backoff.ms	When the worker is out of sync with other workers and fails to catch up within worker.sync.timeout.ms, leave the Connect cluster for this long before rejoining.	int
access.control.allow.methods	Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD.	string
access.control.allow.origin	Value to set the Access-Control-Allow-Origin header to for REST API requests.To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '*' to allow access from any domain. The default value only allows access from the domain of the REST API.	string
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list
	The number of samples	

metrics.num.samples	The number of samples maintained to compute metrics.	int
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long
offset.flush.interval.ms	Interval at which to try committing offsets for tasks.	long
offset.flush.timeout.ms	Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt.	long
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	long
rest.advertised.host.name	If this is set, this is the hostname that will be given out to other workers to connect to.	string
rest.advertised.port	If this is set, this is the port that will be given out to other workers to connect to.	int
rest.host.name	Hostname for the REST API. If this is set, it will only bind to this interface.	string
rest.port	Port for the REST API to listen on.	int
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry	double

	has been reached, at which time it will try to renew the ticket.	
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.	list
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string
task.shutdown.graceful.timeout.ms	Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially.	long

3.5 Kafka Streams Configs

Below is the configuration of the Kafka Streams client library.

NameDescriptionTypeDefaultValid ValuesImportance			
application.id	An identifier for the stream processing application. Must be unique within the Kafka cluster. It is used as 1) the default client-id prefix, 2) the group-id for membership management, 3) the changelog topic prefix.	string	
	A list of host/port pairs to use for		

bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list	
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string	""
zookeeper.connect	Zookeeper connect string for Kafka topics management.	string	""
key.serde	Serializer / deserializer class for key that implements the `Serde` interface.	class	cla
partition.grouper	Partition grouper class that implements the `PartitionGrouper` interface.	class	cla
replication.factor	The replication factor for change log topics and repartition topics created by the stream processing application.	int	1
state.dir	Directory location for state store.	string	/tn
timestamp.extractor	Timestamp extractor class that implements the `TimestampExtractor` interface.	class	cla or
value.serde	Serializer / deserializer class for value that implements the `Serde` interface.	class	cla
	The maximum number of records		

commit.interval.ms	The frequency with which to save the position of the processor.	long	30
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30
num.standby.replicas	The number of standby replicas for each task.	int	0
num.stream.threads	The number of threads to execute stream processing.	int	1
poll.ms	The amount of time in milliseconds to block waiting for input.	long	10
state.cleanup.delay.ms	The amount of time in milliseconds to wait before deleting state when a partition has migrated.	long	60

3. Configuration

Kafka uses key-value pairs in the **property file format** for configuration. These values can be supplied either from a file or programmatically.

3.1 Broker Configs

The essential configurations are the following:

- `broker.id`
- `log.dirs`
- `zookeeper.connect`

Topic-level configurations and defaults are discussed in more detail **below**.

Name	Description
zookeeper.connect	Zookeeper host string
advertised.host.name	DEPRECATED: only used when `advertised.listeners` or `listeners` are no Use `advertised.listeners` instead. Hostna to publish to ZooKeeper for clients to use IaaS environments, this may need to be different from the interface to which the b binds. If this is not set, it will use the valu `host.name` if configured. Otherwise it wi the value returned from java.net.InetAddress.getCanonicalHostNa
advertised.listeners	Listeners to publish to ZooKeeper for clie use, if different than the listeners above. I IaaS environments, this may need to be different from the interface to which the b binds. If this is not set, the value for `liste will be used.
advertised.port	DEPRECATED: only used when `advertised.listeners` or `listeners` are no Use `advertised.listeners` instead. The po publish to ZooKeeper for clients to use. In environments, this may need to be differe from the port to which the broker binds. If is not set, it will publish the same port tha broker binds to.
auto.create.topics.enable	Enable auto creation of topic on the serve
auto.leader.rebalance.enable	Enables auto leader balancing. A backgro thread checks and triggers leader balanc

	required at regular intervals
background.threads	The number of threads to use for various background processing tasks
broker.id	The broker id for this server. If unset, a u broker id will be generated.To avoid confl between zookeeper generated broker id's user configured broker id's, generated br idsstart from reserved.broker.max.id + 1.
compression.type	Specify the final compression type for a g topic. This configuration accepts the stan compression codecs ('gzip', 'snappy', 'lz4 additionally accepts 'uncompressed' whic equivalent to no compression; and 'produ which means retain the original compress codec set by the producer.
delete.topic.enable	Enables delete topic. Delete topic through admin tool will have no effect if this config turned off
host.name	DEPRECATED: only used when `listener not set. Use `listeners` instead. hostname broker. If this is set, it will only bind to this address. If this is not set, it will bind to all interfaces
leader.imbalance.check.interval.seconds	The frequency with which the partition rebalance check is triggered by the contr
leader.imbalance.per.broker.percentage	The ratio of leader imbalance allowed per broker. The controller would trigger a lead balance if it goes above this value per br The value is specified in percentage.
listeners	Listener List - Comma-separated list of U we will listen on and their protocols. Spec hostname as 0.0.0.0 to bind to all interfac Leave hostname empty to bind to default interface. Examples of legal listener lists: PLAINTEXT:\/\/myhost:9092,TRACE:\/\/:9 PLAINTEXT:\/\/0.0.0.0:9092, TRACE:\/\/localhost:9093
log.dir	The directory in which the log data is kep (supplemental for log.dirs property)
log.dirs	The directories in which the log data is ke not set, the value in log.dir is used
log.flush.interval.messages	The number of messages accumulated o log partition before messages are flushed disk
	The maximum time in ms that a message

log.flush.interval.ms	any topic is kept in memory before flushe disk. If not set, the value in log.flush.scheduler.interval.ms is used
log.flush.offset.checkpoint.interval.ms	The frequency with which we update the persistent record of the last flush which a the log recovery point
log.flush.scheduler.interval.ms	The frequency in ms that the log flusher c whether any log needs to be flushed to di
log.retention.bytes	The maximum size of the log before delet
log.retention.hours	The number of hours to keep a log file be deleting it (in hours), tertiary to log.retenti property
log.retention.minutes	The number of minutes to keep a log file deleting it (in minutes), secondary to log.retention.ms property. If not set, the v in log.retention.hours is used
log.retention.ms	The number of milliseconds to keep a log before deleting it (in milliseconds), If not s the value in log.retention.minutes is used
log.roll.hours	The maximum time before a new log segr is rolled out (in hours), secondary to log.r property
log.roll.jitter.hours	The maximum jitter to subtract from logRollTimeMillis (in hours), secondary to log.roll.jitter.ms property
log.roll.jitter.ms	The maximum jitter to subtract from logRollTimeMillis (in milliseconds). If not s the value in log.roll.jitter.hours is used
log.roll.ms	The maximum time before a new log segr is rolled out (in milliseconds). If not set, th value in log.roll.hours is used
log.segment.bytes	The maximum size of a single log file
log.segment.delete.delay.ms	The amount of time to wait before deletin file from the filesystem
message.max.bytes	The maximum size of message that the s can receive
min.insync.replicas	define the minimum number of replicas in needed to satisfy a produce request with acks=all (or -1)
num.io.threads	The number of io threads that the server for carrying out network requests
num.network.threads	the number of network threads that the se

num.network.threads	uses for handling network requests
num.recovery.threads.per.data.dir	The number of threads per data directory used for log recovery at startup and flush shutdown
num.replica.fetchers	Number of fetcher threads used to replica messages from a source broker. Increasi this value can increase the degree of I\/O parallelism in the follower broker.
offset.metadata.max.bytes	The maximum size for a metadata entry associated with an offset commit
offsets.commit.required.acks	The required acks before the commit can accepted. In general, the default (-1) shou not be overridden
offsets.commit.timeout.ms	Offset commit will be delayed until all repl for the offsets topic receive the commit or timeout is reached. This is similar to the producer request timeout.
offsets.load.buffer.size	Batch size for reading from the offsets segments when loading offsets into the ca
offsets.retention.check.interval.ms	Frequency at which to check for stale offs
offsets.retention.minutes	Log retention window in minutes for offse topic
offsets.topic.compression.codec	Compression codec for the offsets topic - compression may be used to achieve "ato commits
offsets.topic.num.partitions	The number of partitions for the offset com topic (should not change after deploymer
offsets.topic.replication.factor	The replication factor for the offsets topic higher to ensure availability). To ensure th the effective replication factor of the offse topic is the configured value, the number alive brokers has to be at least the replica factor at the time of the first request for th offsets topic. If not, either the offsets topic creation will fail or it will get a replication f of min(alive brokers, configured replicatio factor)
offsets.topic.segment.bytes	The offsets topic segment bytes should b relatively small in order to facilitate faster compaction and cache loads
port	DEPRECATED: only used when `listener not set. Use `listeners` instead. the port to listen and accept connections on

queued.max.requests	The number of queued requests allowed before blocking the network threads
quota.consumer.default	Any consumer distinguished by clientId\/consumer group will get throttled fetches more bytes than this value per-se
quota.producer.default	Any producer distinguished by clientId wil throttled if it produces more bytes than th value per-second
replica.fetch.max.bytes	The number of bytes of messages to atte fetch
replica.fetch.min.bytes	Minimum bytes expected for each fetch response. If not enough bytes, wait up to replicaMaxWaitTimeMs
replica.fetch.wait.max.ms	max wait time for each fetcher request iss by follower replicas. This value should alv be less than the replica.lag.time.max.ms times to prevent frequent shrinking of ISR low throughput topics
replica.high.watermark.checkpoint.interval.ms	The frequency with which the high waterr is saved out to disk
replica.lag.time.max.ms	If a follower hasn't sent any fetch request hasn't consumed up to the leaders log en offset for at least this time, the leader will remove the follower from isr
replica.socket.receive.buffer.bytes	The socket receive buffer for network req
replica.socket.timeout.ms	The socket timeout for network requests. value should be at least replica.fetch.wait.max.ms
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is received before the timeout elapses the c will resend the request if necessary or fai request if retries are exhausted.
socket.receive.buffer.bytes	The SO_RCVBUF buffer of the socket se sockets
socket.request.max.bytes	The maximum number of bytes in a socke request
socket.send.buffer.bytes	The SO_SNDBUF buffer of the socket se sockets
unclean.leader.election.enable	Indicates whether to enable replicas not i ISR set to be elected as leader as a last r even though doing so may result in data l

zookeeper.connection.timeout.ms	The max time that the client waits to esta a connection to zookeeper. If not set, the in zookeeper.session.timeout.ms is used
zookeeper.session.timeout.ms	Zookeeper session timeout
zookeeper.set.acl	Set client to use secure ACLs
broker.id.generation.enable	Enable automatic broker id generation on server? When enabled the value configur reserved.broker.max.id should be reviewe
broker.rack	Rack of the broker. This will be used in ra aware replication assignment for fault tolerance. Examples: `RACK1`, `us-east-
connections.max.idle.ms	Idle connections timeout: the server sock processor threads close the connections idle more than this
controlled.shutdown.enable	Enable controlled shutdown of the server
controlled.shutdown.max.retries	Controlled shutdown can fail for multiple reasons. This determines the number of r when such failure happens
controlled.shutdown.retry.backoff.ms	Before each retry, the system needs time recover from the state that caused the pr failure (Controller fail over, replica lag etc config determines the amount of time to w before retrying.
controller.socket.timeout.ms	The socket timeout for controller-to-broke channels
default.replication.factor	default replication factors for automaticall created topics
fetch.purgatory.purge.interval.requests	The purge interval (in number of requests the fetch request purgatory
group.max.session.timeout.ms	The maximum allowed session timeout fo registered consumers. Longer timeouts g consumers more time to process messag between heartbeats at the cost of a longe to detect failures.
group.min.session.timeout.ms	The minimum allowed session timeout fo registered consumers. Shorter timeouts l to quicker failure detection at the cost of r frequent consumer heartbeating, which c overwhelm broker resources.
	Specify which version of the inter-broker protocol will be used. This is typically bun after all brokers were upgraded to a new version. Example of some valid values ar

	0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2. 0.9.0.0, 0.9.0.1 Check ApiVersion for the list.
log.cleaner.backoff.ms	The amount of time to sleep when there a logs to clean
log.cleaner.dedupe.buffer.size	The total memory used for log deduplicat across all cleaner threads
log.cleaner.delete.retention.ms	How long are delete records retained?
log.cleaner.enable	Enable the log cleaner process to run on server? Should be enabled if using any to with a cleanup.policy=compact including internal offsets topic. If disabled those top will not be compacted and continually gro size.
log.cleaner.io.buffer.load.factor	Log cleaner dedupe buffer load factor. Th percentage full the dedupe buffer can bec A higher value will allow more log to be cleaned at once but will lead to more has collisions
log.cleaner.io.buffer.size	The total memory used for log cleaner I\/O buffers across all cleaner threads
log.cleaner.io.max.bytes.per.second	The log cleaner will be throttled so that th of its read and write i\/o will be less than t value on average
log.cleaner.min.cleanable.ratio	The minimum ratio of dirty log to total log log to eligible for cleaning
log.cleaner.threads	The number of background threads to us log cleaning
log.cleanup.policy	The default cleanup policy for segments beyond the retention window, must be eit "delete" or "compact"
log.index.interval.bytes	The interval with which we add an entry t offset index
log.index.size.max.bytes	The maximum size in bytes of the offset i
log.message.format.version	Specify the message format version the b will use to append messages to the logs. value should be a valid ApiVersion. Some examples are: 0.8.2, 0.9.0.0, 0.10.0, chec ApiVersion for more details. By setting a particular message format version, the us certifying that all the existing messages o are smaller or equal than the specified ve Setting this value incorrectly will cause consumers with older versions to break a

	consumers with older versions to break a will receive messages with a format that t don't understand.
log.message.timestamp.difference.max.ms	The maximum difference allowed betwee timestamp when a broker receives a mes and the timestamp specified in the messa message.timestamp.type=CreateTime, a message will be rejected if the difference timestamp exceeds this threshold. This configuration is ignored if message.timestamp.type=LogAppendTim
log.message.timestamp.type	Define whether the timestamp in the mes is message create time or log append tim The value should be either `CreateTime` `LogAppendTime`
log.preallocate	Should pre allocate file when create new segment? If you are using Kafka on Wind you probably need to set it to true.
log.retention.check.interval.ms	The frequency in milliseconds that the log cleaner checks whether any log is eligible deletion
max.connections.per.ip	The maximum number of connections we from each ip address
max.connections.per.ip.overrides	Per-ip or hostname overrides to the defau maximum number of connections
num.partitions	The default number of log partitions per t
principal.builder.class	The fully qualified name of a class that implements the PrincipalBuilder interface which is currently used to build the Princi connections with the SSL SecurityProtoc
producer.purgatory.purge.interval.requests	The purge interval (in number of requests the producer request purgatory
replica.fetch.backoff.ms	The amount of time to sleep when fetch partition error occurs.
reserved.broker.max.id	Max number that can be used for a broke
sasl.enabled.mechanisms	The list of SASL mechanisms enabled in Kafka server. The list may contain any mechanism for which a security provider available. Only GSSAPI is enabled by de
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.

sasl.kerberos.principal.to.local.rules	usernames). The rules are evaluated in o and the first rule that matches a principal is used to map it to a short name. Any lat rules in the list are ignored. By default, principal names of the form {username}\/{hostname}@{REALM} are mapped to {username}. For more details the format please see **security authoriza and acls**.
sasl.kerberos.service.name	The Kerberos principal name that Kafka r as. This can be defined either in Kafka's J config or in Kafka's config.
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which it will try to renew the ticket.
sasl.mechanism.inter.broker.protocol	SASL mechanism used for inter-broker communication. Default is GSSAPI.
security.inter.broker.protocol	Security protocol used to communicate between brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption and key exchange algorithm used to nego the security settings for a network connec using TLS or SSL network protocol.By de all the available cipher suites are support
ssl.client.auth	Configures kafka broker to request client authentication. The following settings are common: `ssl.client.auth=required` If set required client authentication is required. `ssl.client.auth=requested` This means client authentication is optional. u requested , if this option is set client can choose not to provide authentication information about itself `ssl.client.auth=n` This means client authentication is not ne
ssl.enabled.protocols	The list of protocols enabled for SSL connections.
ssl.key.password	The password of the private key in the ke store file. This is optional for client.
	The algorithm used by key manager facto SSL connections. Default value is the key

ssl.keymanager.algorithm	The algorithm used by key manager facto SSL connections. Default value is the key manager factory algorithm configured for Java Virtual Machine.
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two authentication for client.
ssl.keystore.password	The store password for the key store file. optional for client and only needed if ssl.keystore.location is configured.
ssl.keystore.type	The file format of the key store file. This is optional for client.
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which fine for most cases. Allowed values in rec JVMs are TLS, TLSv1.1 and TLSv1.2. SS SSLv2 and SSLv3 may be supported in o JVMs, but their usage is discouraged due known security vulnerabilities.
ssl.provider	The name of the security provider used fo connections. Default value is the default security provider of the JVM.
ssl.trustmanager.algorithm	The algorithm used by trust manager fact for SSL connections. Default value is the manager factory algorithm configured for Java Virtual Machine.
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
ssl.truststore.type	The file format of the trust store file.
authorizer.class.name	The authorizer class that should be used authorization
metric.reporters	A list of classes to use as metrics reporte Implementing the `MetricReporter` interfac allows plugging in classes that will be not of new metric creation. The JmxReporter always included to register JMX statistics
metrics.num.samples	The number of samples maintained to co metrics.
metrics.sample.window.ms	The window of time a metrics sample is computed over.
quota.window.num	The number of samples to retain in memo
quota.window.size.seconds	The time span of each sample

ssl.endpoint.identification.algorithm	server hostname using server certificate.
zookeeper.sync.time.ms	How far a ZK follower can be behind a ZK leader

More details about broker configuration can be found in the scala class `kafka.server.KafkaConfig`.

Topic-level configuration Configurations pertinent to topics have both a global default as well an optional per-topic override. If no per-topic configuration is given the global default is used. The override can be set at topic creation time by giving one or more `--config` options. This example creates a topic named *my-topic* with a custom max message size and flush rate:

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --partitio
ns 1
        --replication-factor 1 --config max.message.bytes=64000 --config flush.message
s=1
```

Overrides can also be changed or set later using the alter topic command. This example updates the max message size for *my-topic*:

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic my-topic
    --config max.message.bytes=128000
```

To remove an override you can do

```
 > bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic my-topic
    --delete-config max.message.bytes
```

The following are the topic-level configurations. The server's default configuration for this property is given under the Server Default Property heading, setting this default in the server config allows you to change the default given to topics that have no override specified.

PropertyDefaultServer Default PropertyDescription			
cleanup.policy	delete	log.cleanup.policy	A string that is "compact". Th the retention p log segments. ("delete") will d segments whe time or size lin reached. The '

			will enable log the topic.
delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms	The amount of delete tombsto compacted to also gives a bo which a consu a read if they b to ensure that snapshot of th (otherwise del may be collect complete their
flush.messages	None	log.flush.interval.messages	This setting all interval at whic fsync of data w For example if we would fsyn message; if it w fsync after eve In general we not set this an durability and system's back capabilities as This setting ca a per-topic bas topic configu
flush.ms	None	log.flush.interval.ms	This setting all time interval a force an fsync the log. For ex set to 1000 we 1000 ms had p we recommen and use replic and allow the background flu is more efficien
index.interval.bytes	4096	log.index.interval.bytes	This setting co frequently Kafk entry to it's offs default setting index a messa 4096 bytes. M reads to jump position in the index larger. Y need to chang

max.message.bytes	1,000,000	message.max.bytes	This is largest Kafka will allo to this topic. N increase this s increase your size so they ca this large.
min.cleanable.dirty.ratio	0.5	log.cleaner.min.cleanable.ratio	This configura frequently the attempt to clea (assuming log enabled). By c cleaning a log 50% of the log compacted. Th maximum spa log by duplicat 50% of the log duplicates). A mean fewer, m cleanings but wasted space
min.insync.replicas	1	min.insync.replicas	When a produ "all", min.insyr the minimum r that must ackr the write to be successful. If t cannot be met will raise an ex NotEnoughRe NotEnoughRe When used to min.insync.rep allow you to en durability guar scenario woul topic with a rep set min.insync produce with a will ensure tha raises an exce of replicas do
retention.bytes	None	log.retention.bytes	This configura maximum size before we will segments to fr are using the ' policy. By defa limit only a tim

retention.ms	7 days	log.retention.minutes	This configura maximum time before we will segments to fr are using the " policy. This rep how soon cons their data.
segment.bytes	1 GB	log.segment.bytes	This configura segment file si Retention and done a file at a segment size r but less granu retention.
segment.index.bytes	10 MB	log.index.size.max.bytes	This configura size of the ind offsets to file p preallocate thi shrink it only a generally shou change this se
segment.ms	7 days	log.roll.hours	This configura period of time will force the lo segment file is that retention c compact old d
segment.jitter.ms	0	log.roll.jitter.{ms,hours}	The maximum from logRollTi

3.2 Producer Configs

Below is the configuration of the Java producer:

NameDescriptionTypeDefaultValid ValuesImportance	
bootstrap.servers	A list of host\/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective o which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection t discover the full cluster membership (which ma change dynamically), this list need not contain

	one, though, in case a server is down).
key.serializer	Serializer class for key that implements the `Serializer` interface.
value.serializer	Serializer class for value that implements the `Serializer` interface.
acks	The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are common: `acks=0` If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the `retries` configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1. `acks=1` This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost. `acks=all` This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.
buffer.memory	The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will block for `max.block.ms` after which it will throw an exception.This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.
compression.type	The compression type for all data generated by the producer. The default is none (i.e. no compression). Valid values are `none` , `gzip` , `snappy` , or `lz4` . Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more

	also impact the compression ratio (more batching means better compression).
retries	Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries without setting `max.in.flight.requests.per.connection` 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first.
ssl.key.password	The password of the private key in the key store file. This is optional for client.
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.
ssl.keystore.password	The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
batch.size	The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes.No attempt will be made to batch records larger than this size.Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip\/por by allowing a logical application name to be included in server-side request logging.
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.

linger.ms	The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding small amount of artificial delay—that is, rather than immediately sending out a record the producer will wait for up to the given delay to allow other records to be sent so that the send can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get `batch.size` worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting `linger.ms=5`, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absense of load.
max.block.ms	The configuration controls how long `KafkaProducer.send()` and `KafkaProducer.partitionsFor()` will block.These methods can be blocked either because the buffer is full or metadata unavailable.Blocking in the user-supplied serializers or partitioner will not be counted against this timeout.
max.request.size	The maximum size of a request in bytes. This is also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.
partitioner.class	Partitioner class that implements the `Partitioner` interface.
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client

	will resend the request if necessary or fail the request if retries are exhausted.
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF to use when sending data.
ssl.enabled.protocols	The list of protocols enabled for SSL connections.
ssl.keystore.type	The file format of the key store file. This is optional for client.
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
ssl.truststore.type	The file format of the trust store file.
timeout.ms	The configuration controls the maximum amount of time the server will wait for acknowledgments from followers to meet the acknowledgment requirements the producer has specified with the `acks` configuration. If the requested number of acknowledgments are not met when the timeout elapses an error will be returned. This timeout is measured on the server side and does not include the network latency of the request.
	When our memory buffer is exhausted we must either stop accepting new records (block) or throw errors. By default this setting is false and the producer will no longer throw a BufferExhaustException but instead will use the `max.block.ms` value to block, after which it

block.on.buffer.full	will throw a TimeoutException. Setting this property to true will set the `max.block.ms` to Long.MAX_VALUE. *Also if this property is set t true, parameter _ `metadata.fetch.timeout.ms` is* not longer honored._This parameter is deprecated and will be removed in a future release. Parameter `max.block.ms` should be used instead.
interceptor.classes	A list of classes to use as interceptors. Implementing the `ProducerInterceptor` interface allows you to intercept (and possibly mutate) the records received by the producer before they are published to the Kafka cluster. By default, there are no interceptors.
max.in.flight.requests.per.connection	The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).
metadata.fetch.timeout.ms	The first time data is sent to a topic we must fetch metadata about that topic to know which servers host the topic's partitions. This fetch to succeed before throwing an exception back to the client.
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified o new metric creation. The JmxReporter is alway included to register JMX statistics.
metrics.num.samples	The number of samples maintained to compute metrics.
metrics.sample.window.ms	The window of time a metrics sample is computed over.
reconnect.backoff.ms	The amount of time to wait before attempting t reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.
	The amount of time to wait before attempting t retry a failed request to a given topic partition.

retry.backoff.ms	This avoids repeatedly sending requests in a tight loop under some failure scenarios.
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket expiry has been reached, at which time it will tr to renew the ticket.
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.
ssl.trustmanager.algorithm	The algorithm used by trust manager factory fo SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

For those interested in the legacy Scala producer configs, information can be found **here**.

3.3 Consumer Configs

We introduce both the old 0.8 consumer configs and the new consumer configs respectively below.

3.3.1 Old Consumer Configs

The essential old consumer configurations are the following:

- `group.id`
- `zookeeper.connect`

PropertyDefaultDescription		

group.id		to which this consumer belongs. By setting the multiple processes indicate that they are all pa consumer group.
zookeeper.connect		Specifies the ZooKeeper connection string in t `hostname:port` where host and port are the ho ZooKeeper server. To allow connecting throug nodes when that ZooKeeper machine is down specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:por` may also have a ZooKeeper chroot path as pa ZooKeeper connection string which puts its da path in the global ZooKeeper namespace. If so should use the same chroot path in its connec example to give a chroot path of `/chroot/path` the connection string as `hostname1:port1,hostname2:port2,hostname3:`
consumer.id	null	Generated automatically if not set.
socket.timeout.ms	30 * 1000	The socket timeout for network requests. The will be max.fetch.wait + socket.timeout.ms.
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests
fetch.message.max.bytes	1024 * 1024	The number of bytes of messages to attempt t topic-partition in each fetch request. These byt into memory for each partition, so this helps co used by the consumer. The fetch request size as large as the maximum message size the se it is possible for the producer to send message consumer can fetch.
num.consumer.fetchers	1	The number fetcher threads used to fetch data
auto.commit.enable	true	If true, periodically commit to ZooKeeper the o already fetched by the consumer. This commit used when the process fails as the position fro consumer will begin.
auto.commit.interval.ms	60 * 1000	The frequency in ms that the consumer offsets zookeeper.
queued.max.message.chunks	2	Max number of message chunks buffered for c chunk can be up to fetch.message.max.bytes.
rebalance.max.retries	4	When a new consumer joins a consumer grou consumers attempt to "rebalance" the load to a each consumer. If the set of consumers chang assignment is taking place the rebalance will fa setting controls the maximum number of attem up.
fetch.min.bytes	1	The minimum amount of data the server shoul request. If insufficient data is available the requ

		request. If insufficient data is available the requ that much data to accumulate before answerin
fetch.wait.max.ms	100	The maximum amount of time the server will b answering the fetch request if there isn't suffici immediately satisfy fetch.min.bytes
rebalance.backoff.ms	2000	Backoff time between retries during rebalance explicitly, the value in zookeeper.sync.time.ms
refresh.leader.backoff.ms	200	Backoff time to wait before trying to determine partition that has just lost its leader.
auto.offset.reset	largest	What to do when there is no initial offset in Zoo offset is out of range:

* smallest : automatically reset the offset to the smallest offset

* largest : automatically reset the offset to the largest offset

* anything else: throw exception to the consumer | | consumer.timeout.ms | -1 | Throw a timeout exception to the consumer if no message is available for consumption after the specified interval | | exclude.internal.topics | true | Whether messages from internal topics (such as offsets) should be exposed to the consumer. | | client.id | group id value | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. | | zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur. | | zookeeper.connection.timeout.ms | 6000 | The max time that the client waits while establishing a connection to zookeeper. | | zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader | | offsets.storage | zookeeper | Select where offsets should be stored (zookeeper or kafka). | | offsets.channel.backoff.ms | 1000 | The backoff period when reconnecting the offsets channel or retrying failed offset fetch\/commit requests. | | offsets.channel.socket.timeout.ms | 10000 | Socket timeout when reading responses for offset fetch\/commit requests. This timeout is also used for ConsumerMetadata requests that are used to query for the offset manager. | | offsets.commit.max.retries | 5 | Retry the offset commit up to this many times on failure. This retry count only applies to offset commits during shut-down. It does not apply to commits originating from the auto-commit thread. It also does not apply to attempts to query for the offset coordinator before committing offsets. i.e., if a consumer metadata request fails for any reason, it will be retried and that retry does not count toward this limit. | | dual.commit.enabled | true | If you are using "kafka" as offsets.storage, you can dual commit offsets to ZooKeeper (in addition to Kafka). This is required during migration from zookeeper-based offset storage to kafka-based offset storage. With respect to any given consumer group, it is safe to turn this off after all instances within that group have been migrated to the new version that commits offsets to the broker (instead of directly to ZooKeeper). | | partition.assignment.strategy | range | Select between the "range" or

"roundrobin" strategy for assigning partitions to consumer streams.The round-robin partition assignor lays out all the available partitions and all the available consumer threads. It then proceeds to do a round-robin assignment from partition to consumer thread. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts will be within a delta of exactly one across all consumer threads.) Round-robin assignment is permitted only if: (a) Every topic has the same number of streams within a consumer instance (b) The set of subscribed topics is identical for every consumer instance within the group.Range partitioning works on a per-topic basis. For each topic, we lay out the available partitions in numeric order and the consumer threads in lexicographic order. We then divide the number of partitions by the total number of consumer streams (threads) to determine the number of partitions to assign to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition. |

More details about consumer configuration can be found in the scala class `kafka.consumer.ConsumerConfig`.

3.3.2 New Consumer Configs

Since 0.9.0.0 we have been working on a replacement for our existing simple and high-level consumers. The code is considered beta quality. Below is the configuration for the new consumer:

NameDescriptionTypeDefaultValid ValuesImportance	
bootstrap.servers	A list of host\/port pairs to use for establishing t to the Kafka cluster. The client will make use o irrespective of which servers are specified here this list only impacts the initial hosts used to dis servers. This list should be in the form `host1:port1,host2:port2,...`. Since these used for the initial connection to discover the fu membership (which may change dynamically), contain the full set of servers (you may want m though, in case a server is down).
key.deserializer	Deserializer class for key that implements the `Deserializer` interface.
value.deserializer	Deserializer class for value that implements th `Deserializer` interface.
fetch.min.bytes	The minimum amount of data the server shoul request. If insufficient data is available the requ much data to accumulate before answering the setting of 1 byte means that fetch requests are as a single byte of data is available or the fetch

	will cause the server to wait for larger amounts accumulate which can improve server throughp some additional latency.
group.id	A unique string that identifies the consumer gro belongs to. This property is required if the cons group management functionality by using `subs` Kafka-based offset management strategy.
heartbeat.interval.ms	The expected time between heartbeats to the c when using Kafka's group management facilitie used to ensure that the consumer's session sta facilitate rebalancing when new consumers join The value must be set lower than `session.time` should be set no higher than 1\/3 of that value. even lower to control the expected time for nor
max.partition.fetch.bytes	The maximum amount of data per-partition the The maximum total memory used for a request `* max.partition.fetch.bytes` . This size must be the maximum message size the server allows for the producer to send messages larger than fetch. If that happens, the consumer can get st large message on a certain partition.
session.timeout.ms	The timeout used to detect failures when using management facilities. When a consumer's hea received within the session timeout, the broker consumer as failed and rebalance the group. S sent only when poll() is invoked, a higher sessi more time for message processing in the cons the cost of a longer time to detect hard failures `max.poll.records` for another option to control in the poll loop. Note that the value must be in as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max`
ssl.key.password	The password of the private key in the key stor for client.
ssl.keystore.location	The location of the key store file. This is option be used for two-way authentication for client.
ssl.keystore.password	The store password for the key store file.This is and only needed if ssl.keystore.location is conf
ssl.truststore.location	The location of the trust store file.
ssl.truststore.password	The password for the trust store file.
auto.offset.reset	What to do when there is no initial offset in Kaf offset does not exist any more on the server (e has been deleted):earliest: automatically reset earliest offsetlatest: automatically reset the offs offsetnone: throw exception to the consumer if found for the consumer's groupanything else: t

	found for the consumer's groupanything else: t the consumer.
connections.max.idle.ms	Close idle connections after the number of mill this config.
enable.auto.commit	If true the consumer's offset will be periodically background.
exclude.internal.topics	Whether records from internal topics (such as exposed to the consumer. If set to `true` the o records from an internal topic is subscribing to
max.poll.records	The maximum number of records returned in a
partition.assignment.strategy	The class name of the partition assignment str will use to distribute partition ownership among instances when group management is used
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVB reading data.
request.timeout.ms	The configuration controls the maximum amou will wait for the response of a request. If the re received before the timeout elapses the client request if necessary or fail the request if retries
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs a defined either in Kafka's JAAS config or in Kafk
sasl.mechanism	SASL mechanism used for client connections. mechanism for which a security provider is ava default mechanism.
security.protocol	Protocol used to communicate with brokers. Va PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF sending data.
ssl.enabled.protocols	The list of protocols enabled for SSL connectio
ssl.keystore.type	The file format of the key store file. This is opti
ssl.protocol	The SSL protocol used to generate the SSLCo is TLS, which is fine for most cases. Allowed v are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 a supported in older JVMs, but their usage is dis known security vulnerabilities.
ssl.provider	The name of the security provider used for SSL Default value is the default security provider of
ssl.truststore.type	The file format of the trust store file.
auto.commit.interval.ms	The frequency in milliseconds that the consum committed to Kafka if `enable.auto.commit` is se
	Automatically check the CRC32 of the records

check.crcs	ensures no on-the-wire or on-disk corruption to occurred. This check adds some overhead, so cases seeking extreme performance.
client.id	An id string to pass to the server when making purpose of this is to be able to track the source just ip\/port by allowing a logical application na server-side request logging.
fetch.max.wait.ms	The maximum amount of time the server will bl answering the fetch request if there isn't sufficie immediately satisfy the requirement given by fe
interceptor.classes	A list of classes to use as interceptors. Implem the `ConsumerInterceptor` interface allows you t possibly mutate) records received by the consu there are no interceptors.
metadata.max.age.ms	The period of time in milliseconds after which w metadata even if we haven't seen any partition to proactively discover any new brokers or part
metric.reporters	A list of classes to use as metrics reporters. Im the `MetricReporter` interface allows plugging in notified of new metric creation. The JmxReport to register JMX statistics.
metrics.num.samples	The number of samples maintained to compute
metrics.sample.window.ms	The window of time a metrics sample is compu
reconnect.backoff.ms	The amount of time to wait before attempting t given host. This avoids repeatedly connecting loop. This backoff applies to all requests sent b the broker.
retry.backoff.ms	The amount of time to wait before attempting t request to a given topic partition. This avoids re requests in a tight loop under some failure sce
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attem
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renew
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified windo last refresh to ticket's expiry has been reached try to renew the ticket.
ssl.cipher.suites	A list of cipher suites. This is a named combina authentication, encryption, MAC and key excha to negotiate the security settings for a network TLS or SSL network protocol.By default all the suites are supported.
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate using server certificate.

ssl.keymanager.algorithm	The algorithm used by key manager factory for Default value is the key manager factory algori the Java Virtual Machine.
ssl.trustmanager.algorithm	The algorithm used by trust manager factory fo Default value is the trust manager factory algor the Java Virtual Machine.

3.4 Kafka Connect Configs

Below is the configuration of the Kafka Connect framework.

NameDescriptionTypeDefaultValid ValuesImportance		
config.storage.topic	kafka topic to store configs	string
group.id	A unique string that identifies the Connect cluster group this worker belongs to.	string
internal.key.converter	Converter class for internal key Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.	class
internal.value.converter	Converter class for offset value Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.	class
key.converter	Converter class for key Connect data that implements the `Converter` interface.	class
offset.storage.topic	kafka topic to store connector offsets in	string
status.storage.topic	kafka topic to track connector and task status	string
value.converter	Converter class for value Connect data that implements the `Converter` interface.	class
	A list of host\/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial	

bootstrap.servers	this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list
cluster	ID for this cluster, which is used to provide a namespace so multiple Kafka Connect clusters or instances may co-exist while sharing a single Kafka cluster.	string
heartbeat.interval.ms	The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1\/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int
session.timeout.ms	The timeout used to detect failures when using Kafka's group management facilities.	int
ssl.key.password	The password of the private key in the key store file. This is optional for client.	passwo
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string
ssl.keystore.password	The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.	passwo
ssl.truststore.location	The location of the trust store file.	string

ssl.truststore.password	The password for the trust store file.	passwo
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	int
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data.	int
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list
ssl.keystore.type	The file format of the key store file. This is optional for client.	string
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known	string

	security vulnerabilities.	
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string
ssl.truststore.type	The file format of the trust store file.	string
worker.sync.timeout.ms	When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before giving up, leaving the group, and waiting a backoff period before rejoining.	int
worker.unsync.backoff.ms	When the worker is out of sync with other workers and fails to catch up within worker.sync.timeout.ms, leave the Connect cluster for this long before rejoining.	int
access.control.allow.methods	Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD.	string
access.control.allow.origin	Value to set the Access-Control-Allow-Origin header to for REST API requests.To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '*' to allow access from any domain. The default value only allows access from the domain of the REST API.	string
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip\/port by allowing a logical application name to be included in server-side request logging.	string
	The period of time in milliseconds	

metadata.max.age.ms	after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list
metrics.num.samples	The number of samples maintained to compute metrics.	int
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long
offset.flush.interval.ms	Interval at which to try committing offsets for tasks.	long
offset.flush.timeout.ms	Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt.	long
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	long
rest.advertised.host.name	If this is set, this is the hostname that will be given out to other workers to connect to.	string
rest.advertised.port	If this is set, this is the port that will be given out to other workers to connect to.	int
rest.host.name	Hostname for the REST API. If this is set, it will only bind to this interface.	string
rest.port	Port for the REST API to listen on.	int
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This	long

retry.backoff.ms	avoids repeatedly sending requests in a tight loop under some failure scenarios.	long
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.	list
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string
task.shutdown.graceful.timeout.ms	Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially.	long

3.5 Kafka Streams Configs

Below is the configuration of the Kafka Streams client library.

NameDescriptionTypeDefaultValid ValuesImportance			
application.id	An identifier for the stream processing application. Must be unique within the Kafka cluster. It is used as 1) the default client-id prefix, 2) the group-id for membership management, 3) the changelog topic prefix.	string	
bootstrap.servers	A list of host\/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,... .` Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list	
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip\/port by allowing a logical application name to be included in server-side request logging.	string	""
zookeeper.connect	Zookeeper connect string for Kafka topics management.	string	""
key.serde	Serializer \/ deserializer class for key that implements the `Serde` interface.	class	cla
partition.grouper	Partition grouper class that implements the `PartitionGrouper` interface.	class	cla
	The replication factor for change log topics and repartition topics		

replication.factor	created by the stream processing application.	int	1
state.dir	Directory location for state store.	string	\/t
timestamp.extractor	Timestamp extractor class that implements the `TimestampExtractor` interface.	class	cl or
value.serde	Serializer \/ deserializer class for value that implements the `Serde` interface.	class	cl
buffered.records.per.partition	The maximum number of records to buffer per partition.	int	1(
commit.interval.ms	The frequency with which to save the position of the processor.	long	3(
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	3(
num.standby.replicas	The number of standby replicas for each task.	int	0
num.stream.threads	The number of threads to execute stream processing.	int	1
poll.ms	The amount of time in milliseconds to block waiting for input.	long	1(
state.cleanup.delay.ms	The amount of time in milliseconds to wait before deleting state when a partition has migrated.	long	6(

4. 设计

4.1 设计初衷

我们将 Kafka 设计为一个能处理大公司可能存在的所有实时数据流的统一平台。为了实现这个目标我们考虑了各式各样的应用场景。

它必须有很高的吞吐量来支撑像实时日志合并这类大规模的事件流。

它必须能很好的处理大规模数据积压问题来支撑像线下系统周期性的导入数据的场景。

同时它也需要可以胜任低延迟的消息分发来支撑与传统消息机制类似的应用场景。

我们还希望它支持分区、分布式、消息流的实时创建、分发处理。这些需求促成了我们现在的分区和消费者模型。

最后在作为信息流上游为其它数据系统提供服务的场景下，我们也深知系统必须能够提供在主机故障时的容错担保。

为了实现对上述应用场景的支持最终导致我们设计了一系列更相似于数据库日志而不是传统消息系统的元素。我们将在后续段落中概述其中的某些设计元素。

4.2 持久化

不要惧怕文件系统！

Kafka 在消息的存储和缓存中重度依赖文件系统。因为“磁盘慢”这个普遍性的认知，常常使人们怀疑一个这样的持久化结构是否能提供所需的性能。但实际上磁盘因为使用的方式不同，它可能比人们预想的慢很多也可能比人们预想的快很多；而且一个合理设计的磁盘文件结构常常可以使磁盘运行得和网络一样快。

磁盘性能的核心指标在过去的十年间已经从磁盘的寻道延迟变成了硬件驱动的吞吐量。故此在一个**JBOD** 操作的由 6 张 7200 转磁盘组成的 RAID-5 阵列之上的线性写操作的性能能达到 600MB/sec 左右，但是它的随机写性能却只有 100k/sec 左右，两者之间相差了 6000 倍以上。因为线性的读操作和写操作是最常见的磁盘应用模式，并且这也被操作系统进行了高度的优化。现在的操作系统都提供了预读取和写合并技术、即预读取数倍于数据的大文件块和将多个小的逻辑写操作合并成一个大的物理写操作的技术。关于这个话题的进一步的讨论可以参照 **ACM Queue article**；他们发现实际上线性的磁盘访问在某些场景下比随机的内存访问还快！

为了填补这个性能的差异，现在操作系统越来越激进地使用主内存来作为磁盘的缓冲。现代的操作系统都非常乐意将 所有的 空闲的内存作为磁盘的缓存，虽然这将在内存重分配期间带来一点性能影响。所有的磁盘的读写操作都会经过这块统一的缓存。而且这个特性除非使用 direct I/O 技术很难被关闭掉，所以即使一个进程在进程内维护了数据的缓存，实际上这些数据依旧在操作系统的页缓存上存在一个副本，实际上所有的数据都被存储了两次。

另外我们是基于 JVM 进行建设的，任何一个稍微了解 Java 内存模型的人都知道以下两点：

1. 对象的内存占用是非常高，常常是数据存储空间的两倍以上（甚至更差）。
2. Java 的内存回收随着堆内数据的增长会变得更加繁琐和缓慢。

综上所述，使用文件系统和页缓存相较于维护一个内存缓存或者其它结构更占优势 -- 我们通过自动化的访问所有空闲内存的能力将缓存的空间扩大了至少两倍，之后又因为保存压缩的字节结构而不是单独对象结构又将此扩充了两倍以上。最终这使我们在一个 32G 的主机之上拥有了一个高达 28-30G 的没有 GC 问题的缓存。而且这个缓存即使在服务重启之后也能保持热度，相反进程内的缓存要么还需要重建预热（10G 的缓存可能耗时 10 分钟）要么就从一个完全空白的缓冲开始服务（这意味着初始化期间性能将很差）。同时这也很大的简化了代码，因为所有的维护缓存和文件系统之间正确性逻辑现在都在操作系统中了，这常常比重复造轮子更加高效和正确。如果你的磁盘使用方式更倾向与线性读取，预读取技术将在每次磁盘读操作时将有效的数据高效的预填充到这些缓存中。

这使人想到一个非常简单的设计：相对于竭尽所能的维护内存内结构而且要时刻注意在空间不足时谨记要将它们 flush 到文件系统中，我们可以颠覆这种做法。所有的数据被直接写入文件系统上一个可暂不执行磁盘 flush 操作的持久化日志文件中。实际上这意味着这些数据是被传送到了内核的页缓存上。

这种基于页缓存的设计可以参见在 这篇关于 **Varnish** 的论文

常量时间就足够

消息系统使用的持久化数据结构通常是和 BTree 相关联的消费者队列或者其他用于存储消息元数据的通用随机访问数据结构。BTree 是最通用的数据结构选择，它可以在消息系统中支持各种事务性和非事务性语义。虽然 BTree 的操作复杂度是 O(logN) ，但是成本也很高。通常我们认为 O(logN) 基本等同于常数时间，但这条在磁盘操作中不成立。磁盘寻址是每 10ms 一跳，并且每个磁盘同时只能够执行一次寻址，因此并行受到了限制。因此即使是少量的磁盘寻址也会有很高的开销。由于存储系统将非常快的缓存操作和非常慢的物理磁盘操作混在一起，在确定缓存大小的情况下树结构的实际性能随着数据的增长是非线性的 -- 比如数据翻倍时性能下降不止两倍。

所以直观来看，持久化队列可以建立在简单的读取和向文件后追加两种操作之上，这和日志解决方案相同。这种结构的优点在于所有的操作复杂度都是 O(1)，而且读操作不会阻塞写操作，读操作之间也不会互相影响。这有着明显的性能优势，由于性能和数据大小完全不相关 -

- 服务器现在可以充分利用大量廉价、低转速的 1+TB SATA 硬盘。 虽然这些硬盘的寻址性能很差，但他们在大规模读写方面的性能是可以接受的，而且价格是原来的三分之一、容量是原来的三倍。

在不产生任何性能损失的情况下能够访问几乎无限的硬盘空间，这意味着我们可以提供一些其它消息系统不常见的特性。例如：在 Kafka 中，我们可以让消息保留相对较长的一段时间（比如一周），而不是试图在被消费后立即删除。正如我们后面将要提到的，这给消费者带来了很大的灵活性。

4.3 性能 Efficiency

我们在性能提升上做了很大的努力。我们的主要使用场景之一是处理网页活动信息，这个数据量非常巨大，因为每个页面都可能有大量的写入。此外我们假设发布每个 message 至少被一个 consumer （通常是多个） 来消费，因此我们尽可能去降低消费的代价。

从构建和运行很多相似系统的经验中我们还发现，性能是多租户系统运营的关键。如果下游的基础设施服务很轻易被应用层冲击形成瓶颈，那么小的改变也会造成问题。足够快的处理速度使我们可以保证在应用被负载压垮之前基础组建不会出问题。当尝试去运行一个集中式集群来承载成百上千个应用程序时，这一点非常重要，因为应用层使用的方式几乎每天都会发生变化。

我们在上一节讨论了磁盘性能。 一旦消除了磁盘访问模式不佳的情况，该类系统性能低下的主要原因就剩下了两个：大量的小型 I/O 操作（译者注：小包问题），以及过多的字节拷贝（译者注：ZeroCopy需求）。

小型的 I/O 操作发生在客户端和服务端之间以及服务端自身的持久化操作中。

为了避免这种情况，我们的协议是建立在一个“消息块”的抽象基础上，合理将消息分组。 这使得网络请求将多个消息打包成一组，而不是每次发送一条消息，从而使整组消息分担网络中往返的开销。服务器一次性的将多个消息快依次追加到日志文件中， Consumer 也是每次获取多个大型有序的消息块。

这个简单的优化对速度有着数量级的提升。批处理允许更大的网络数据包，更大的顺序读写磁盘操作，连续的内存块等等，所有这些都使 KafKa 能将随机性突发性的消息写操作变成顺序性的写操作最终流向消费者。

另一个低效率的操作是字节拷贝，在消息量少时，这不是什么问题。但是在高负载的情况下，影响就不容忽视。为了避免这种情况，我们让 producer ，broker 和 consumer 都共享的标准化的二进制消息格式，这样数据块不用修改就能在他们之间传递。

broker 维护的消息日志本身就是一个文件目录，每个文件都由一系列以相同格式写入到磁盘的消息集合组成，这种写入格式被 producer 和 consumer 共用。保持这种通用格式可以对一些很重要的操作进行优化：持久化日志块的网络传输。 现代的 unix 操作系统提供了高度优化

的数据路径，用于将数据从 pagecache 转移到 socket 网络连接中；在 Linux 中系统调用 sendfile 做到这一点。

为了理解 sendfile 的意义，首先要了解数据从文件到套接字的一般数据传输路径：

1. 操作系统从磁盘读取数据到内核空间的 pagecache
2. 应用程序读取内核空间的数据到用户空间的缓冲区
3. 应用程序将数据（用户空间的缓冲区）写回内核空间的套接字缓冲区（内核空间）
4. 操作系统将数据从套接字缓冲区（内核空间）复制到通过网络发送的 NIC 缓冲区

这显然是低效的，有四次 copy 操作和两次系统调用。使用 sendfile 方法，可以允许操作系统将数据从 pagecache 直接发送到网络，这样避免重复数据复制。所以这种优化方式，只需要最后一步的 copy 操作，将数据复制到 NIC 缓冲区。

我们预期的使用场景是一个 topic 被多个消费者消费。使用 zero-copy （零拷贝）优化，数据仅仅会被复制到 pagecache 一次，在后续的消费过程中都可以复用，而不是保存在内存中在每次消费时再复制到内核空间。这使得消息能够以接近网络连接的速度被消费。

pagecache 和 sendfile 的组合使用意味着，在一个 Kafka 集群中，大多数的(紧跟生产者的)consumer 消费时，将看不到磁盘上的读取活动，因为数据完全由缓存提供。

Java 中更多关于 sendfile 方法和 zero-copy （零拷贝） 相关的资料，可以参考这里的文章

端到端批量压缩

某些情况下，数据传输的瓶颈并不是 CPU或者磁盘，而是网络带宽。尤其是当数据消息通道需要在数据中心通过广域网进行传输时。当然用户可以在不需要 Kafka 支持下一次一个压缩消息，但这样会造成非常差的压缩率和消息重复类型冗余，比如 JSON 中字段名称或者是 Web 日志中用户代理或者是公共字符串值。高性能的压缩是一次压缩多个消息，而不是单独压缩。

Kafka 以高效的批处理格式支持一批消息可以压缩在一起发送到服务器。这批消息将以压缩格式写入，并且在日志中保持压缩，只会在 consumer 消费时解压缩。

Kafka 支持 GZIP，Snappy 和 LZ4 压缩协议，更多有关压缩的资料参看这里。

4.4 The Producer

Load balancing

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this all Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriately direct its requests.

生产者直接发送数据到主分区的服务器上，不需要经过任何中间路由。为了让生产者实现这个功能，所有的 kafka 服务器节点都能响应这样的元数据请求：哪些服务器是活着的，主题的哪些分区是主分区，分配在哪个服务器上，这样生产者就能适当地直接发送它的请求到服务器上。

The client controls which partition it publishes messages to. This can be done at random, implementing a kind of random load balancing, or it can be done by some semantic partitioning function. We expose the interface for semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition (there is also an option to override the partition function if need be). For example if the key chosen was a user id then all data for a given user would be sent to the same partition. This in turn will allow consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in consumers.

客户端控制消息发送数据到哪个分区，这个可以实现随机的负载均衡方式，或者使用一些特定语义的分区函数。我们有提供特定分区的接口让用于根据指定的键值进行 hash 分区（当然也有选项可以重写分区函数），例如，如果使用用户 ID 作为 key，则用户相关的所有数据都会被分发到同一个分区上。这允许消费者在消费数据时做一些特定的本地化处理。这样的分区风格经常被设计用于一些对本地处理比较敏感的消费者。

Asynchronous send

Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say 64k or 10 ms). This allows the accumulation of more bytes to send, and few larger I/O operations on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

批处理是提升性能的一个主要驱动，为了允许批量处理，kafka 生产者会尝试在内存中汇总数据，并用一次请求批次提交信息。批处理，不仅仅可以配置指定的消息数量，也可以指定等待特定的延迟时间（如 64k 或 10ms)，这允许汇总更多的数据后再发送，在服务器端也会减少更多的 IO 操作。该缓冲是可配置的，并给出了一个机制，通过权衡少量额外的延迟时间获取更好的吞吐量。

更多的细节可以在 producer 的 **configuration** 和 **api** 文档中进行详细的了解。

4.5 消费者

The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

Kafka consumer 通过向 broker 发出一个“fetch”请求来获取它想要消费的 partition。consumer 的每个请求都在 log 中指定了对应的 offset，并接收从该位置开始的一大块数据。因此，consumer 对于该位置的控制就显得极为重要，并且可以在需要的时候通过回退到该位置再次消费对应的数据。

Push vs. pull

An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as **Scribe** and **Apache Flume**, follow a very different push-based path where data is pushed downstream. There are pros and cons to both approaches. However, a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal is generally for the consumer to be able to consume at the maximum possible rate; unfortunately, in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

我们最初考虑的问题是：究竟是由 consumer 从 broker 那边 pull 数据，还是由 broker 将数据 push 到 consumer。Kafka 在这里采取了一种较为传统的设计方式，也是大多数的消息系统所共享的方式：即 producer 把数据 push 到 broker，然后 consumer 从 broker 中 pull 数据。也有一些 logging-centric 的系统，比如 Scribe 和 Apache Flume ，沿着一条完全不同的 push-based 的路径，将数据 push 到下游节点。这两种方法都有优缺点。然而，由于 broker 控制着数据传输速率，所以 push-based 系统很难处理不同的 consumer。让 broker 控制数据传输速率主要是为了让 consumer 能够以可能的最大速率消费；然而不幸的是，这导致着在 push-based 的系统中，当消费速率低于生产速率时，consumer 往往会不堪重负（本质上类似于拒绝服务攻击）。pull-based 系统有一个很好的特性，那就是当 consumer 速率落后于 producer 时，可以在适当的时间赶上来。还可以通过使用某种 backoff 协议来减少这种现

象：即 consumer 可以通过 backoff 表示它已经不堪重负了，然而通过获得负载情况来充分使用 consumer（但永远不超载）这一方式实现起来比它看起来更棘手。之前以这种方式构建系统的尝试，引导着 Kafka 走向了更传统的 pull 模型。

Another advantage of a pull-based system is that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency, this will result in sending a single message at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the consumer always pulls all available messages after its current position in the log (or up to some configurable max size). So one gets optimal batching without introducing unnecessary latency.

另一个 pull-based 系统的优点在于：它可以大批量生产要发送给 consumer 的数据。而 push-based 系统必须选择立即发送请求或者积累更多的数据，然后在不知道下游的 consumer 能否立即处理它的情况下发送这些数据。如果系统调整为低延迟状态，这就会导致一次只发送一条消息，以至于传输的数据不再被缓冲，这种方式是极度浪费的。 而 pull-based 的设计修复了该问题，因为 consumer 总是将所有可用的（或者达到配置的最大长度）消息 pull 到 log 当前位置的后面，从而使得数据能够得到最佳的处理而不会引入不必要的延迟。

The deficiency of a naive pull-based system is that if the broker has no data the consumer may end up polling in a tight loop, effectively busy-waiting for data to arrive. To avoid this we have parameters in our pull request that allow the consumer request to block in a "long poll" waiting until data arrives (and optionally waiting until a given number of bytes is available to ensure large transfer sizes).

简单的 pull-based 系统的不足之处在于：如果 broker 中没有数据，consumer 可能会在一个紧密的循环中结束轮询，实际上 busy-waiting 直到数据到来。为了避免 busy-waiting，我们在 pull 请求中加入参数，使得 consumer 在一个“long pull”中阻塞等待，直到数据到来（还可以选择等待给定字节长度的数据来确保传输长度）。

You could imagine other possible designs which would be only pull, end-to-end. The producer would locally write to a local log, and brokers would pull from that with consumers pulling from them. A similar type of "store-and-forward" producer is often proposed. This is intriguing but we felt not very suitable for our target use cases which have thousands of producers. Our experience running persistent data systems at scale led us to feel that involving thousands of disks in the system across many applications would not actually make things more reliable and would be a nightmare to operate. And in practice we have found that we can run a pipeline with strong SLAs at large scale without a need for producer persistence.

你可以想象其它可能只基于 pull 、end-to-end 的设计，例如 producer 直接将数据写入一个本地的 log，然后 broker 从 producer 那里 pull 数据，最后 consumer 从 broker 中 pull 数据。通常提到的还有“store-and-forward”式 producer， 这是一种很有趣的设计，但我们觉得它跟我们设定的有数以千计的生产者的应用场景不太相符。我们在运行大规模持久化数据系统方面的经验使我们感觉到，横跨多个应用、涉及数千磁盘的系统事实上并不会让事情更可靠，反而会成为操作时的噩梦。在实践中，我们发现可以通过大规模运行的带有强大的 SLAs 的 pipeline，而省略 producer 的持久化过程。

消费者位置

Keeping track of *what* has been consumed is, surprisingly, one of the key performance points of a messaging system.

令人惊讶的是，持续追踪已经被消费的内容是消息系统的关键性能点之一。

Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally immediately or it may wait for acknowledgement from the consumer. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else this state could go. Since the data structures used for storage in many messaging systems scale poorly, this is also a pragmatic choice--since the broker knows what is consumed it can immediately delete it, keeping the data size small.

大多数消息系统都在 broker 上保存被消费消息的元数据。也就是说，当消息被传递给 consumer，broker 要么立即在本地记录该事件，要么等待 consumer 的确认后再记录。这是一种相当直接的选择，而且事实上对于单机服务器来说，也没其它地方能够存储这些状态信息。由于大多数消息系统用于存储的数据结构规模都很小，所以这也是一个很实用的选择 -- 因为只要 broker 知道哪些消息被消费了，就可以在本地立即进行删除，一直保持较小的数据量。

What is perhaps not obvious is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as **consumed** immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as **sent** not **consumed** when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as**consumed**. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about

every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

也许不太明显，但要让 broker 和 consumer 就被消费的数据保持一致性也不是一个小问题。如果 broker 在每条消息被发送到网络的时候，立即将其标记为 **consumed**，那么一旦 consumer 无法处理该消息（可能由 consumer 崩溃或者请求超时或者其他原因导致），该消息就会丢失。 为了解决消息丢失的问题，许多消息系统增加了确认机制：即当消息被发送出去的时候，消息仅被标记为 **sent** 而不是 **consumed**；然后 broker 会等待一个来自 consumer 的特定确认，再将消息标记为 **consumed**。这个策略修复了消息丢失的问题，但也产生了新问题。首先，如果 consumer 处理了消息但在发送确认之前出错了，那么该消息就会被消费两次。第二个是关于性能的，现在 broker 必须为每条消息保存多个状态（首先对其加锁，确保该消息只被发送一次，然后将其永久的标记为 consumed，以便将其移除）。还有更棘手的问题要处理，比如如何处理已经发送但一直得不到确认的消息。

Kafka handles this differently. Our topic is divided into a set of totally ordered partitions, each of which is consumed by one consumer at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.

Kafka 使用完全不同的方式解决消息丢失问题。Kafka 的 topic 被分割成了一组完全有序的 **partition**，其中每一个 partition 在任意给定的时间内只能被每个订阅了这个 **topic** 的 **consumer** 组中的一个 **consumer** 消费。这意味着 partition 中 每一个 consumer 的位置仅仅是一个数字，即下一条要消费的消息的 offset。这使得被消费的消息的状态信息相当少，每个 partition 只需要一个数字。这个状态信息还可以作为周期性的 checkpoint。这以非常低的代价实现了和消息确认机制等同的效果。

There is a side benefit of this decision. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

这种方式还有一个附加的好处。consumer 可以回退到之前的 offset 来再次消费之前的数据，这个操作违反了队列的基本原则，但事实证明对大多数 consumer 来说这是一个必不可少的特性。 例如，如果 consumer 的代码有 bug，并且在 bug 被发现前已经有一部分数据被消费了，那么 consumer 可以在 bug 修复后通过回退到之前的 offset 来再次消费这些数据。

离线数据加载

Scalable persistence allows for the possibility of consumers that only periodically consume such as batch data loads that periodically bulk-load data into an offline system such as Hadoop or a relational data warehouse.

可伸缩的持久化特性允许 consumer 只进行周期性的消费，例如批量数据加载，周期性将数据加载到诸如 Hadoop 和关系型数据库之类的离线系统中。

In the case of Hadoop we parallelize the data load by splitting the load over individual map tasks, one for each node/topic/partition combination, allowing full parallelism in the loading. Hadoop provides the task management, and tasks which fail can restart without danger of duplicate data—they simply restart from their original position.

在 Hadoop 的应用场景中，我们通过将数据加载分配到多个独立的 map 任务来实现并行化，每一个 map 任务负责一个 node/topic/partition，从而达到充分并行化。Hadoop 提供了任务管理机制，失败的任务可以重新启动而不会有重复数据的风险，只需要简单的从原来的位置重启即可。

4.6 消息交付语义

Now that we understand a little about how producers and consumers work, let's discuss the semantic guarantees Kafka provides between producer and consumer. Clearly there are multiple possible message delivery guarantees that could be provided:

现在我们对于 producer 和 consumer 的工作原理已将有了一点了解，让我们接着讨论 Kafka 在 producer 和 consumer 之间提供的语义保证。显然，Kafka 可以提供的消息交付语义保证有多种：

- *At most once* - 消息可能会丢失但绝不重传
- *At least once* - 消息可以重传但绝不丢失
- *Exactly once* - 这可能是用户真正想要的，每条消息只被传递一次

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

值得注意的是，这个问题被分成了两部分：发布消息的持久性保证和消费消息的保证。

Many systems claim to provide "exactly once" delivery semantics, but it is important to read the fine print, most of these claims are misleading (i.e. they don't translate to the case where consumers or producers can fail, cases where there are multiple consumer processes, or cases where data written to disk can be lost).

很多系统声称提供了“Exactly once”的消息交付语义，然而阅读它们的细则很重要，因为这些声称大多数都是误导性的 （即它们没有考虑 consumer 或 producer 可能失败的情况，以及存在多个 consumer 进行处理的情况，或者写入磁盘的数据可能丢失的情况。).

Kafka's semantics are straight-forward. When publishing a message we have a notion of the message being "committed" to the log. Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive". The definition of alive as well as a description of which types of failures we attempt to handle will be described in more detail in the next section. For now let's assume a perfect, lossless broker and try to understand the guarantees to the producer and consumer. If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed. This is similar to the semantics of inserting into a database table with an autogenerated key.

Kafka 的语义是直截了当的。发布消息时，我们会有一个消息的概念被“committed”到 log 中。一旦消息被提交，只要有一个 broker 备份了该消息写入的 partition，并且保持“**alive**”状态，该消息就不会丢失。有关 committed message 和 alive partition 的定义，以及我们试图解决的故障类型都将在下一节进行细致描述。现在让我们假设存在完美无缺的 broker，然后来试着理解 Kafka 对 producer 和 consumer 的语义保证。如果一个 producer 在试图发送消息的时候发生了网络故障，则不确定网络错误发生在消息提交之前还是之后。这与使用自动生成的键插入到数据库表中的语义场景很相似。

These are not the strongest possible semantics for publishers. Although we cannot be sure of what happened in the case of a network error, it is possible to allow the producer to generate a sort of "primary key" that makes retrying the produce request idempotent. This feature is not trivial for a replicated system because of course it must work even (or especially) in the case of a server failure. With this feature it would suffice for the producer to retry until it receives acknowledgement of a successfully committed message at which point we would guarantee the message had been published exactly once. We hope to add this in a future Kafka version.

Not all use cases require such strong guarantees. For uses which are latency sensitive we allow the producer to specify the durability level it desires. If the producer specifies that it wants to wait on the message being committed this can take on the order of 10 ms. However the producer can also specify that it wants to perform the send completely asynchronously or that it wants to wait only until the leader (but not necessarily the followers) have the message.

并非所有使用场景都需要这么强的保证。对于延迟敏感的应用场景，我们允许生产者指定它需要的持久性级别。如果 producer 指定了它想要等待消息被提交，则可以使用 10ms 的量级。然而， producer 也可以指定它想要完全异步地执行发送，或者它只想等待直到 leader 节点拥有该消息（follower 节点有没有无所谓）。

Now let's describe the semantics from the point-of-view of the consumer. All replicas have the exact same log with the same offsets. The consumer controls its position in this log. If the consumer never crashed it could just store this position in memory, but if the consumer fails and we want this topic partition to be taken over by another process the new process

will need to choose an appropriate position from which to start processing. Let's say the consumer reads some messages -- it has several options for processing the messages and updating its position.

现在让我们从 consumer 的视角来描述该语义。所有的副本都有相同的 log 和相同的 offset。consumer 负责控制它在 log 中的位置。如果 consumer 永远不崩溃，那么它可以将这个位置信息只存储在内存中。但如果 consumer 发生了故障，我们希望这个 topic partition 被另一个进程接管，那么新进程需要选择一个合适的位置开始进行处理。假设 consumer 要读取一些消息——它有几个处理消息和更新位置的选项。

1. It can read the messages, then save its position in the log, and finally process the messages. In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing. In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed. This corresponds to "at-most-once" semantics as in the case of a consumer failure messages may not be processed.

2. Consumer 可以先读取消息，然后将它的位置保存到 log 中，最后再对消息进行处理。在这种情况下，消费者进程可能会在保存其位置之后，在还没有保存消息处理的输出之前发生崩溃。而在这种情况下，即使在此位置之前的一些消息没有被处理，接管处理的进程将从保存的位置开始。在 consumer 发生故障的情况下，这对应于“at-most-once”的语义，可能会有消息得不到处理。

3. It can read the messages, process the messages, and finally save its position. In this case there is a possibility that the consumer process crashes after processing messages but before saving its position. In this case when the new process takes over the first few messages it receives will already have been processed. This corresponds to the "at-least-once" semantics in the case of consumer failure. In many cases messages have a primary key and so the updates are idempotent (receiving the same message twice just overwrites a record with another copy of itself).

4. Consumer 可以先读取消息，然后处理消息，最后再保存它的位置。在这种情况下，消费者进程可能会在处理了消息之后，但还没有保存位置之前发生崩溃。而在这种情况下，当新的进程接管后，它最初收到的一部分消息都已经被处理过了。在 consumer 发生故障的情况下，这对应于“at-least-once”的语义。在许多应用场景中，消息都设有一个主键，所以更新操作是幂等的（相同的消息接收两次时，第二次写入会覆盖掉第一次写入的记录）。

5. So what about exactly once semantics (i.e. the thing you actually want)? The limitation here is not actually a feature of the messaging system but rather the need to co-ordinate the consumer's position with what is actually stored as output. The classic way of achieving this would be to introduce a two-phase commit between the storage for the

consumer position and the storage of the consumers output. But this can be handled more simply and generally by simply letting the consumer store its offset in the same place as its output. This is better because many of the output systems a consumer might want to write to will not support a two-phase commit. As an example of this, our Hadoop ETL that populates data in HDFS stores its offsets in HDFS with the data it reads so that it is guaranteed that either data and offsets are both updated or neither is. We follow similar patterns for many other data systems which require these stronger semantics and for which the messages do not have a primary key to allow for deduplication.

那么 exactly once 语义（即你真正想要的东西）呢？这里的问题其实不是消息系统的特性而是需要协调 consumer 的位置和实际上被存储的结果。传统的做法是在 consumer 位置存储和 consumer 输出存储之间引入 two-phase commit。但通过让 consumer 在相同的位置保存 offset 和输出可以让这一问题变得简单。 这也是一种更好的方式，因为大多数 consumer 想写入的输出系统都不支持 two-phase commit。举个例子，Kafka Connect 连接器，它将所读取的数据和数据的 offset 一起写入到 HDFS，以保证数据和 offset 都被更新，或者两者都不被更新。 对于其它很多需要这些较强语义，并且没有主键来避免消息重复的数据系统，我们也遵循类似的模式。

So effectively Kafka guarantees at-least-once delivery by default and allows the user to implement at most once delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages. Exactly-once delivery requires co-operation with the destination storage system but Kafka provides the offset which makes implementing this straight-forward.

Kafka 默认保证 at-least-once 的消息交付， 并且 Kafka 允许用户通过禁用 producer 的重传功能和让 consumer 在处理一批消息之前提交 offset，来实现 at-most-once 的消息交付。

4.7 Replication

Kafka replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

Kafka 允许 topic 的 partition 拥有若干副本，你可以在 server 端配置 partition 的副本数量。当集群中的节点出现故障时，能自动进行故障转移，保证数据的可用性。

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: slaves are inactive, throughput is heavily impacted, it requires fiddly manual configuration, etc. Kafka is meant to be used with replication by default—in fact we implement un-replicated topics as replicated topics where the replication factor is one.

其他的消息系统也提供了副本相关的特性，但是在我们（带有偏见）看来，他们的副本功能不常用，而且有很大缺点：slaves 处于非活动状态，导致吞吐量受到严重影响，并且还需要手动配置副本机制。Kafka 默认使用备份机制，事实上，我们将没有设置副本数的 topic 实现为副本数为 1 的 topic 。

The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log—all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).

创建副本的单位是 topic 的 partition ，正常情况下，每个分区都有一个 leader 和零或多个 followers 。总的副本数是包含 leader 的总和。所有的读写操作都由 leader 处理，一般 partition 的数量都比 broker 的数量多的多，各分区的 leader 均匀的分布在 brokers 中。所有的 followers 节点都同步 leader 节点的日志，日志中的消息和偏移量都和 leader 中的一致。（当然，在任何给定时间，leader 节点的日志末尾时可能有几个消息尚未被备份完成）。

Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.

Followers 节点就像普通的 consumer 那样从 leader 节点那里拉取消息并保存在自己的日志文件中。Followers 节点可以从 leader 节点那里批量拉取消息日志到自己的日志文件中。

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions

与大多数分布式系统一样，自动处理故障需要精确定义节点 “alive” 的概念。Kafka 判断节点是否存活有两种方式。

1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)
2. If it is a slave it must replicate the writes happening on the leader and not fall "too far" behind
3. 节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接。
4. 如果节点是个 follower ，它必须能及时的同步 leader 的写操作，并且延时不能太久。

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The determination of stuck and lagging replicas is controlled by the replica.lag.time.max.ms configuration.

我们认为满足这两个条件的节点处于 “in sync” 状态，区别于 “alive” 和 “failed” 。Leader 会追踪所有 “in sync” 的节点。如果有节点挂掉了，或是写超时，或是心跳超时，leader 就会把它从同步副本列表中移除。 同步超时和写超时的时间由 replica.lag.time.max.ms 配置确定。

In distributed systems terminology we only attempt to handle a "fail/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). Kafka does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).

分布式系统中，我们只尝试处理 “fail/recover” 模式的故障，即节点突然停止工作，然后又恢复（节点可能不知道自己曾经挂掉）的状况。Kafka 没有处理所谓的 “Byzantine” 故障，即一个节点出现了随意响应和恶意响应（可能由于 bug 或 非法操作导致）。

A message is considered "committed" when all in sync replicas for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the acks setting that the producer uses.

当所有的分区上 in sync repicas 都应用到 log 上时，消息可以认为是 "committed"，只有 committed 消息才会给 consumer。这意味着 consumer 不需要担心潜在因为 leader 失败而丢失消息。而对于 producer 来说，可以依据 latency 和 durability 来权衡选择是否等待消息被 committed ，这个行动由 producer 使用的 acks 设置来决定。

The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.

在所有时间里，Kafka 保证只要有至少一个同步中的节点存活，提交的消息就不会丢失。

Kafka will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

节点挂掉后，经过短暂的故障转移后，Kafka 将仍然保持可用性，但在网络分区（ network partitions ）的情况下可能不能保持可用性。

Replicated Logs: Quorums, ISRs, and State Machines (Oh my!)

At its heart a Kafka partition is a replicated log. The replicated log 是分布式数据系统重基础的要素之一，实现方法有很多种。A replicated log can be used by other systems as a primitive for implementing other distributed systems in the **state-machine style**.

A replicated log models the process of coming into consensus on the order of a series of values (generally numbering the log entries 0, 1, 2, ...). There are many ways to implement this, but the simplest and fastest is with a leader who chooses the ordering of values provided to it. As long as the leader remains alive, all followers need to only copy the values and ordering the leader chooses.

备份日志按照一系列有序的值（通常是编号为 0、1、2、...) 进行建模。有很多方法可以实现这一点，但最简单和最快的方法是由 leader 节点选择需要提供的有序的值，只要 leader 节点还存活，所有的 follower 只需要拷贝数据并按照 leader 节点的顺序排序。

Of course if leaders didn't fail we wouldn't need followers! When the leader does die we need to choose a new leader from among the followers. But followers themselves may fall behind or crash so we must ensure we choose an up-to-date follower. The fundamental guarantee a log replication algorithm must provide is that if we tell the client a message is committed, and the leader fails, the new leader we elect must also have that message. This yields a tradeoff: if the leader waits for more followers to acknowledge a message before declaring it committed then there will be more potentially electable leaders.

If you choose the number of acknowledgements required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap, then this is called a Quorum.

A common approach to this tradeoff is to use a majority vote for both the commit decision and the leader election. This is not what Kafka does, but let's explore it anyway to understand the tradeoffs. Let's say we have 2*f*+1 replicas. If *f*+1 replicas must receive a message prior to a commit being declared by the leader, and if we elect a new leader by electing the follower with the most complete log from at least *f*+1 replicas, then, with no more than *f* failures, the leader is guaranteed to have all committed messages. This is because among any *f*+1 replicas, there must be at least one replica that contains all committed messages. That replica's log will be the most complete and therefore will be selected as the new leader. There are many remaining details that each algorithm must handle (such as precisely defined what makes a log more complete, ensuring log consistency during leader failure or changing the set of servers in the replica set) but we will ignore these for now.

This majority vote approach has a very nice property: the latency is dependent on only the fastest servers. That is, if the replication factor is three, the latency is determined by the faster slave not the slower one.

There are a rich variety of algorithms in this family including ZooKeeper's **Zab**, **Raft**, and **Viewstamped Replication**. The most similar academic publication we are aware of to Kafka's actual implementation is**PacificA** from Microsoft.

The downside of majority vote is that it doesn't take many failures to leave you with no electable leaders. To tolerate one failure requires three copies of the data, and to tolerate two failures requires five copies of the data. In our experience having only enough redundancy to tolerate a single failure is not enough for a practical system, but doing every write five times, with 5x the disk space requirements and 1/5th the throughput, is not very practical for large volume data problems. This is likely why quorum algorithms more commonly appear for shared cluster configuration such as ZooKeeper but are less common for primary data storage. For example in HDFS the namenode's high-availability feature is built on a **majority-vote-based journal**, but this more expensive approach is not used for the data itself.

大多数投票的缺点是，多数的节点挂掉让你不能选择 leader。要冗余单点故障需要三份数据，并且要冗余两个故障需要五份的数据。根据我们的经验，在一个系统中，仅仅靠冗余来避免单点故障是不够的，但是每写 5 次，对磁盘空间需求是 5 倍， 吞吐量下降到 1/5，这对于处理海量数据问题是不切实际的。这可能是为什么 quorum 算法更常用于共享集群配置（如 ZooKeeper ）， 而不适用于原始数据存储的原因，例如 HDFS 中 namenode 的高可用是建立在基于投票的元数据，这种代价高昂的存储方式不适用数据本身。

Kafka takes a slightly different approach to choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important. With this ISR model and *f+1* replicas, a Kafka topic can tolerate *f* failures without losing committed messages.

Kafka 采取了一种稍微不同的方法来选择它的投票集。 Kafka 不是用大多数投票选择 leader。Kafka 动态维护了一个同步状态的备份的集合 （a set of in-sync replicas），简称 ISR ，在这个集合中的节点都是和 leader 保持高度一致的，只有这个集合的成员才 有资格被选举为 leader，一条消息必须被这个集合 所有 节点读取并追加到日志中了，这条消息才能视为提交。这个 ISR 集合发生变化会在 ZooKeeper 持久化，正因为如此，这个集合中的任何一个节点都有资格被选为 leader 。这对于 Kafka 使用模型中，有很多分区和并确保主从关系是很重要的。因为 ISR 模型和 f+1 副本，一个 Kafka topic 冗余 f 个节点故障而不会丢失任何已经提交的消息。

For most use cases we hope to handle, we think this tradeoff is a reasonable one. In practice, to tolerate _f_failures, both the majority vote and the ISR approach will wait for the same number of replicas to acknowledge before committing a message (e.g. to survive one

failure a majority quorum needs three replicas and one acknowledgement and the ISR approach requires two replicas and one acknowledgement). The ability to commit without the slowest servers is an advantage of the majority vote approach. However, we think it is ameliorated by allowing the client to choose whether they block on the message commit or not, and the additional throughput and disk space due to the lower required replication factor is worth it.

我们认为对于希望处理的大多数场景这种策略是合理的。在实际中，为了冗余 f 节点故障，大多数投票和 ISR 都会在提交消息前确认相同数量的备份被收到（例如在一次故障生存之后，大多数的 quorum 需要三个备份节点和一次确认，ISR 只需要两个备份节点和一次确认），多数投票方法的一个优点是提交时能避免最慢的服务器。但是，我们认为通过允许客户端选择是否阻塞消息提交来改善，和所需的备份数较低而产生的额外的吞吐量和磁盘空间是值得的。

Another important design distinction is that Kafka does not require that crashed nodes recover with all their data intact. It is not uncommon for replication algorithms in this space to depend on the existence of "stable storage" that cannot be lost in any failure-recovery scenario without potential consistency violations. There are two primary problems with this assumption. First, disk errors are the most common problem we observe in real operation of persistent data systems and they often do not leave data intact. Secondly, even if this were not a problem, we do not want to require the use of fsync on every write for our consistency guarantees as this can reduce performance by two to three orders of magnitude. Our protocol for allowing a replica to rejoin the ISR ensures that before rejoining, it must fully re-sync again even if it lost unflushed data in its crash.

另一个重要的设计区别是，Kafka 不要求崩溃的节点恢复所有的数据，在这种空间中的复制算法经常依赖于存在“稳定存储”，在没有违反潜在的一致性的情况下，出现任何故障再恢复情况下都不会丢失。 这个假设有两个主要的问题。首先，我们在持久性数据系统的实际操作中观察到的最常见的问题是磁盘错误，并且它们通常不能保证数据的完整性。其次，即使磁盘错误不是问题，我们也不希望在每次写入时都要求使用 fsync 来保证一致性， 因为这会使性能降低两到三个数量级。我们的协议能确保备份节点重新加入 ISR 之前，即使它挂时没有新的数据，它也必须完整再一次同步数据。

Unclean leader election: 如果节点全挂？

Note that Kafka's guarantee with respect to data loss is predicated on at least one replica remaining in sync. If all the nodes replicating a partition die, this guarantee no longer holds.

请注意，Kafka 对于数据不会丢失的保证，是基于至少一个节点在保持同步状态，一旦分区上的所有备份节点都挂了，就无法保证了。

However a practical system needs to do something reasonable when all the replicas die. If you are unlucky enough to have this occur, it is important to consider what will happen. There are two behaviors that could be implemented:

但是，实际在运行的系统需要去考虑假设一旦所有的备份都挂了，怎么去保证数据不会丢失，这里有两种实现的方法

1. Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).
2. Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

3. 等待一个 ISR 的副本重新恢复正常服务，并选择这个副本作为领 leader （它有极大可能拥有全部数据）。

4. 选择第一个重新恢复正常服务的副本（不一定是 ISR 中的）作为 leader。

This is a simple tradeoff between availability and consistency. If we wait for replicas in the ISR, then we will remain unavailable as long as those replicas are down. If such replicas were destroyed or their data was lost, then we are permanently down. If, on the other hand, a non-in-sync replica comes back to life and we allow it to become leader, then its log becomes the source of truth even though it is not guaranteed to have every committed message. By default Kafka chooses the second strategy and favor choosing a potentially inconsistent replica when all replicas in the ISR are dead. This behavior can be disabled using configuration property unclean.leader.election.enable, to support use cases where downtime is preferable to inconsistency.

这是可用性和一致性之间的简单妥协，如果我只等待 ISR 的备份节点，那么只要 ISR 备份节点都挂了，我们的服务将一直会不可用，如果它们的数据损坏了或者丢失了，那就会是长久的宕机。另一方面，如果不是 ISR 中的节点恢复服务并且我们允许它成为 leader ，那么它的数据就是可信的来源，即使它不能保证记录了每一个已经提交的消息。kafka 默认选择第二种策略，当所有的 ISR 副本都挂掉时，会选择一个可能不同步的备份作为 leader ，可以配置属性 unclean.leader.election.enable 禁用此策略，那么就会使用第 一种策略即停机时间优于不同步。

This dilemma is not specific to Kafka. It exists in any quorum-based scheme. For example in a majority voting scheme, if a majority of servers suffer a permanent failure, then you must either choose to lose 100% of your data or violate consistency by taking what remains on an existing server as your new source of truth.

这种困境不只有 Kafka 遇到，它存在于任何 quorum-based 规则中。例如，在大多数投票算法当中，如果大多数服务器永久性的挂了，那么您要么选择丢失 100% 的数据，要么违背数据的一致性选择一个存活的服务器作为数据可信的来源。

可用性和持久性保证

When writing to Kafka, producers can choose whether they wait for the message to be acknowledged by 0,1 or all (-1) replicas. Note that "acknowledgement by all replicas" does not guarantee that the full set of assigned replicas have received the message. By default, when acks=all, acknowledgement happens as soon as all the current in-sync replicas have received the message. For example, if a topic is configured with only two replicas and one fails (i.e., only one in sync replica remains), then writes that specify acks=all will succeed. However, these writes could be lost if the remaining replica also fails. Although this ensures maximum availability of the partition, this behavior may be undesirable to some users who prefer durability over availability. Therefore, we provide two topic-level configurations that can be used to prefer message durability over availability:

向 Kafka 写数据时，producers 设置 ack 是否提交完成，0：不等待 broker 返回确认消息，1: leader 保存成功返回或，-1(all): 所有备份都保存成功返回。请注意。设置 “ack = all” 并不能保证所有的副本都写入了消息。默认情况下，当 acks = all 时，只要 ISR 副本同步完成，就会返回消息已经写入。例如，一个 topic 仅仅设置了两个副本，那么只有一个 ISR 副本，那么当设置 acks = all 时返回写入成功时，剩下了的那个副本数据也可能数据没有写入。尽管这确保了分区的最大可用性，但是对于偏好数据持久性而不是可用性的一些用户，可能不想用这种策略，因此，我们提供了两个 topic 配置，可用于优先配置消息数据持久性：

1. Disable unclean leader election - if all replicas become unavailable, then the partition will remain unavailable until the most recent leader becomes available again. This effectively prefers unavailability over the risk of message loss. See the previous section on Unclean Leader Election for clarification.
2. Specify a minimum ISR size - the partition will only accept writes if the size of the ISR is above a certain minimum, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses acks=all and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between consistency and availability. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.

- 禁用 unclean leader 选举机制 - 如果所有的备份节点都挂了，分区数据就会不可用，直到最近的 leader 恢复正常。这种策略优先于数据丢失的风险，参看上一节的 unclean leader 选举机制。
- 指定最小的 ISR 集合大小，只有当 ISR 的大小大于最小值，分区才能接受写入操作，以防止仅写入单个备份的消息丢失造成消息不可用的情况，这个设置只有在生产者使用 acks = all 的情况下才会生效，这至少保证消息被 ISR 副本写入。此设置是一致性和可用

性之间的折衷，对于设置更大的最小 ISR 大小保证了更好的一致性，因为它保证将消息被写入了更多的备份，减少了消息丢失的可能性。但是，这会降低可用性，因为如果 ISR 副本的数量低于最小阈值，那么分区将无法写入。

备份管理

The above discussion on replicated logs really covers only a single log, i.e. one topic partition. However a Kafka cluster will manage hundreds or thousands of these partitions. We attempt to balance partitions within a cluster in a round-robin fashion to avoid clustering all partitions for high-volume topics on a small number of nodes. Likewise we try to balance leadership so that each node is the leader for a proportional share of its partitions.

上面关于 replicated logs 的讨论仅仅局限于单一 log ，比如一个 topic 分区。但是 Kafka 集群需要管理成百上千个这样的分区。我们尝试轮流的方式来在集群中平衡分区来避免在小节点上处理大容量的 topic。Likewise we try to balance leadership so that each node is the leader for a proportional share of its partitions.

It is also important to optimize the leadership election process as that is the critical window of unavailability. A naive implementation of leader election would end up running an election per partition for all partitions a node hosted when that node failed. Instead, we elect one of the brokers as the "controller". This controller detects failures at the broker level and is responsible for changing the leader of all affected partitions in a failed broker. The result is that we are able to batch together many of the required leadership change notifications which makes the election process far cheaper and faster for a large number of partitions. If the controller fails, one of the surviving brokers will become the new controller.

同样关于 leadership 选举的过程也同样的重要，这段时间可能是无法服务的间隔。一个原始的 leader 选举实现是当一个节点失败时会在所有的分区节点中选主。相反，我们选用 broker 之一作为 "controller", 这个 controller 检测 broker 失败，并且为所有受到影响的分区改变 leader。这个结果是我们能够将许多需要变更 leadership 的通知整合到一起，让选举过程变得更加容易和快速。如果 controller 失败了，存活的 broker 之一会变成新的 controller。

4.8 日志压缩

Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance. Let's dive into these use cases in more detail and then describe how compaction works.

日志压缩可确保 Kafka 始终至少为单个 topic partition 的数据日志中的每个 message key 保留最新的已知值。这样的设计解决了应用程序崩溃、系统故障后恢复或者应用在运行维护过程中重启后重新加载缓存的场景。接下来让我们深入讨论这些在使用过程中的更多细节，阐述在这个过程中它是如何进行日志压缩的。

So far we have described only the simpler approach to data retention where old log data is discarded after a fixed period of time or when the log reaches some predetermined size. This works well for temporal event data such as logging where each record stands alone. However an important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a database table).

迄今为止，我们只介绍了简单的日志保留方法（当旧的数据保留时间超过指定时间、日志文件大小达到设置大小后就丢弃）。这样的策略非常适用于处理那些暂存的数据，例如记录每条消息之间相互独立的日志。然而在实际使用过程中还有一种非常重要的场景 -- 根据 key 进行数据变更（例如更改数据库表内容），使用以上的方式显然不行。

Let's discuss a concrete example of such a stream. Say we have a topic containing user email addresses; every time a user updates their email address we send a message to this topic using their user id as the primary key. Now say we send the following messages over some time period for a user with id 123, each message corresponding to a change in email address (messages for other ids are omitted):

让我们来讨论一个关于处理这样流式数据的具体的例子。假设我们有一个 topic，里面的内容包含用户的 email 地址；每次用户更新他们的 email 地址时，我们发送一条消息到这个 topic，这里使用用户 Id 作为消息的 key 值。现在，我们在一段时间内为 id 为 123 的用户发送一些消息，每个消息对应 email 地址的改变（其他 ID 消息省略）：

```
123 => bill@microsoft.com
.
.
.
123 => bill@gatesfoundation.org
.
.
.
123 => bill@gmail.com
```

Log compaction gives us a more granular retention mechanism so that we are guaranteed to retain at least the last update for each primary key (e.g. `bill@gmail.com`). By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

日志压缩为我们提供了更精细的保留机制，所以我们至少保留每个 key 的最后一次更新（例如：bill@gmail.com）。这样我们保证日志包含每一个 key 的最终值而不只是最近变更的完整快照。这意味着下游的消费者可以获得最终的状态而无需拿到所有的变化的消息信息。

Let's start by looking at a few use cases where this is useful, then we'll see how it can be used.

让我们先看几个有用的使用场景，然后再看看如何使用它。

1. *Database change subscription*. It is often necessary to have a data set in multiple data systems, and often one of these systems is a database of some kind (either a RDBMS or perhaps a new-fangled key-value store). For example you might have a database, a cache, a search cluster, and a Hadoop cluster. Each change to the database will need to be reflected in the cache, the search cluster, and eventually in Hadoop. In the case that one is only handling the real-time updates you only need recent log. But if you want to be able to reload the cache or restore a failed search node you may need a complete data set.
2. *Event sourcing*. This is a style of application design which co-locates query processing with application design and uses a log of changes as the primary store for the application.
3. *Journaling for high-availability*. A process that does local computation can be made fault-tolerant by logging out changes that it makes to it's local state so another process can reload these changes and carry on if it should fail. A concrete example of this is handling counts, aggregations, and other "group by"-like processing in a stream query system. Samza, a real-time stream-processing framework, **uses this feature** for exactly this purpose.

4. 数据库更改订阅。通常需要在多个数据系统设置拥有一个数据集，这些系统中通常有一个是某种类型的数据库（无论是 RDBMS 或者新流行的 key-value 数据库）。例如，你可能有一个数据库，缓存，搜索引擎集群或者 Hadoop 集群。每次变更数据库，也同时需要变更缓存、搜索引擎以及 hadoop 集群。在只需处理最新日志的实时更新的情况下，你只需要最近的日志。但是，如果你希望能够重新加载缓存或恢复搜索失败的节点，你可能需要一个完整的数据集

5. 事件源。这是一种应用程序设计风格，它将查询处理与应用程序设计相结合，并使用变更的日志作为应用程序的主要存储
6. 日志高可用。执行本地计算的进程可以通过注销对其本地状态所做的更改来实现容错，以便另一个进程可以重新加载这些更改并在出现故障时继续进行。一个具体的例子就是在流查询系统中进行计数，聚合和其他类似“group by”的操作。实时流处理框架 Samza，使用这个特性 正是出于这一原因

In each of these cases one needs primarily to handle the real-time feed of changes, but occasionally, when a machine crashes or data needs to be re-loaded or re-processed, one needs to do a full load. Log compaction allows feeding both of these use cases off the same backing topic. This style of usage of a log is described in more detail in **this blog post**.

在这些场景中，主要需要处理变化的实时 feed，但是偶尔当机器崩溃或需要重新加载或重新处理数据时，需要处理所有数据。日志压缩允许在同一 topic 下同时使用这两个用例。这种日志使用方式更详细的描述请看这篇博客。

The general idea is quite simple. If we had infinite log retention, and we logged each change in the above cases, then we would have captured the state of the system at each time from when it first began. Using this complete log, we could restore to any point in time by replaying the first N records in the log. This hypothetical complete log is not very practical for systems that update a single record many times as the log will grow without bound even for a stable dataset. The simple log retention mechanism which throws away old updates will bound space but the log is no longer a way to restore the current state—now restoring from the beginning of the log no longer recreates the current state as old updates may not be captured at all.

想法很简单，我们有无限的日志，以上每种情况记录变更日志，我们从一开始就捕获每一次变更。使用这个完整的日志，我们可以通过回放日志来恢复到任何一个时间点的状态。然而这种假设的情况下，完整的日志是不实际的，对于那些每一行记录会变更多次的系统，即使数据集很小，日志也会无限的增长下去。丢弃旧日志的简单操作可以限制空间的增长，但是无法重建状态——因为旧的日志被丢弃，可能一部分记录的状态会无法重建（这些记录所有的状态变更都在旧日志中）。

Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

日志压缩机制是更细粒度的、每个记录都保留的机制，而不是基于时间的粗粒度。这个理念是选择性删除那些有更新的变更的记录的日志。这样最终日志至少包含每个 key 的记录的最后一个状态。

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

这种保留策略可以针对每一个 topci 进行设置，遮掩一个集群中，可以让部分 topic 通过时间和大小保留日志，另一些可以通过压缩策略保留。

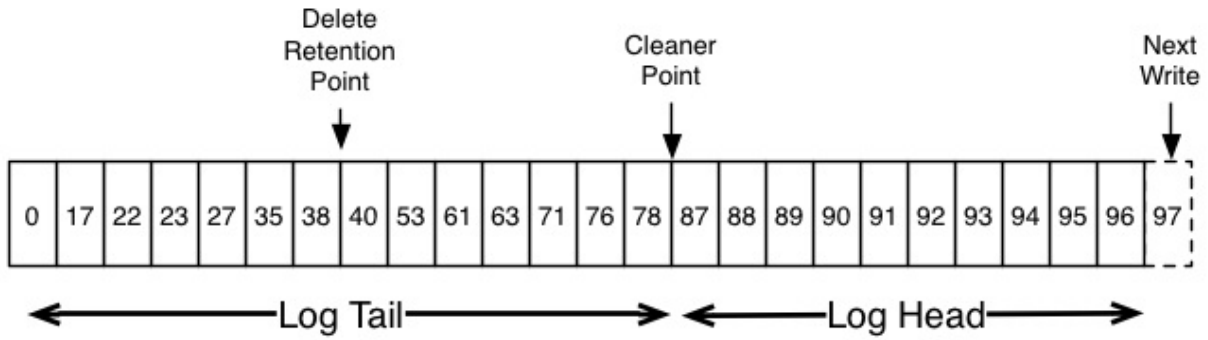
This functionality is inspired by one of LinkedIn's oldest and most successful pieces of infrastructure—a database changelog caching service called **Databus**. Unlike most log-structured storage systems Kafka is built for subscription and organizes data for fast linear reads and writes. Unlike Databus, Kafka acts as a source-of-truth store so it is useful even in situations where the upstream data source would not otherwise be replayable.

这个功能的灵感来自于 LinkedIn 的最古老且最成功的基础设置 -- 一个称为 **Databus** 的数据库变更日志缓存系统。不像大多数的日志存储系统，Kafka 是专门为订阅和快速线性的读和写组织数据而设计。和 Databus 不同，Kafka 作为真实的存储，压缩日志是非常有用的，这非常有利于上游数据源不能重放的情况。

日志压缩基础

Here is a high-level picture that shows the logical structure of a Kafka log with the offset for each message.

这是一个高级别的日志逻辑图，展示了 kafka 日志的每条消息的 offset 逻辑结构。



The head of the log is identical to a traditional Kafka log. It has dense, sequential offsets and retains all messages. Log compaction adds an option for handling the tail of the log. The picture above shows a log with a compacted tail. Note that the messages in the tail of the log retain the original offset assigned when they were first written—that never changes. Note also that all offsets remain valid positions in the log, even if the message with that offset has been compacted away; in this case this position is indistinguishable from the next highest offset that does appear in the log. For example, in the picture above the offsets 36, 37, and 38 are all equivalent positions and a read beginning at any of these offsets would return a message set beginning with 38.

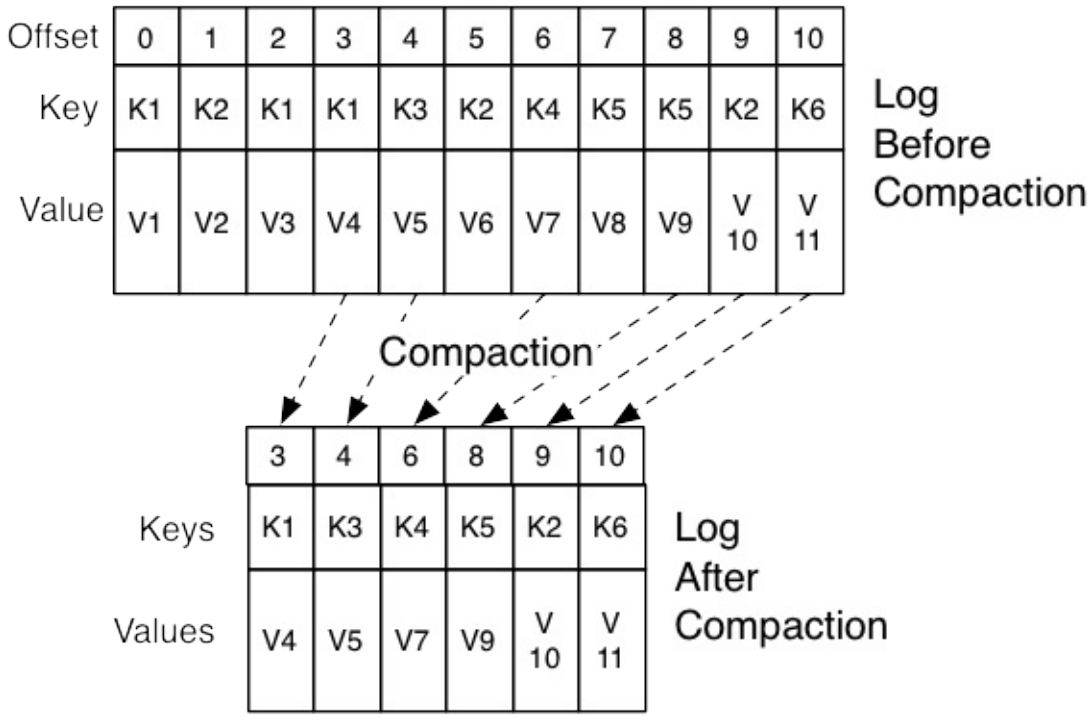
Log head 中包含传统的 Kafka 日志，它包含了连续的 offset 和所有的消息。日志压缩增加了处理 tail Log 的选项。上图展示了日志压缩的的 Log tail 的情况。tail 中的消息保存了初次写入时的 offset。即使该 offset 的消息被压缩，所有 offset 仍然在日志中是有效的。在这个场景中，无法区分和下一个出现的更高 offset 的位置。如上面的例子中，36、37、38 是属于相同位置的，从他们开始读取日志都将从 38 开始。

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "delete retention point" in the above diagram.

压缩也允许删除。通过消息的 key 和空负载（null payload）来标识该消息可从日志中删除。这个删除标记将会引起所有之前拥有相同 key 的消息被移除（包括拥有 key 相同的新消息）。但是删除标记比较特殊，它将在一定周期后被从日志中删除来释放空间。这个时间点被称为“delete retention point”，如上图。

The compaction is done in the background by periodically recopying log segments. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I/O throughput to avoid impacting producers and consumers. The actual process of compacting a log segment looks something like this:

压缩操作通过后台周期性的拷贝日志段来完成。清除操作不会阻塞读取，并且可以被配置不超过一定 IO 吞吐来避免影响 Producer 和 Consumer。实际的日志段压缩过程有点像这样：



What guarantees does log compaction provide?

Log compaction guarantees the following: 日志压缩的保障措施：

1. Any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets.

2. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.
3. The offset for a message never changes. It is the permanent identifier for a position in the log.
4. Any read progressing from offset 0 will see at least the final state of all records in the order they were written. All delete markers for deleted records will be seen provided the reader reaches the head of the log in a time period less than the topic's delete.retention.ms setting (the default is 24 hours). This is important as delete marker removal happens concurrently with read (and thus it is important that we not remove any delete marker prior to the reader seeing it).
5. Any consumer progressing from the start of the log will see at least the *final* state of all records in the order they were written. All delete markers for deleted records will be seen provided the consumer reaches the head of the log in a time period less than the topic's `delete.retention.ms` setting (the default is 24 hours). This is important as delete marker removal happens concurrently with read, and thus it is important that we do not remove any delete marker prior to the consumer seeing it.
6. 任何滞留在日志 head 中的所有消费者能看到写入的所有消息；这些消息都是有序的 offset。 topic 使用 min.compaction.lag.ms 来保障消息写入之前必须经过的最小时间长度，才能被压缩。 这限制了一条消息在 Log Head 中的最短存在时间。
7. 消息始终保持有序。压缩永远不会重新排序消息，只是删除了一些。
8. 消息的 Offset 永远不会变更。这是消息在日志中的永久标志。
9. 任何从头开始处理日志的 Consumer 至少会拿到每个 key 的最终状态。另外，只要 Consumer 在小于 Topic 的 `delete.retention.ms` 设置（默认 24 小时）的时间段内到达 Log head，将会看到所有删除记录的所有删除标记。换句话说，因为移除删除标记和读取是同时发生的，Consumer 可能会因为落后超过 delete.retention.ms 而导致错过删除标记。

Log 压缩细节

Log compaction is handled by the log cleaner, a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Each compactor thread works as follows:

日志压缩由 log cleaner 执行，log cleaner 是一个后台线程池，它会 recopy 日志段文件，移除那些 key 存在于 Log Head 中的记录。每个压缩线程工作的步骤如下：

1. It chooses the log that has the highest ratio of log head to log tail
2. It creates a succinct summary of the last offset for each key in the head of the log
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log. New, clean segments are swapped into the log immediately so the additional

disk space required is just one additional log segment (not a fully copy of the log).

4. The summary of the log head is essentially just a space-compact hash table. It uses exactly 24 bytes per entry. As a result with 8GB of cleaner buffer one cleaner iteration can clean around 366GB of log head (assuming 1k messages).
5. 选择 log head 与 log tail 比率最高的日志
6. 在 head log 中为每个 key 最后 offset 创建一个简单概要
7. 从日志的开始到结束，删除那些在日志中最新出现的 key 的旧值。新的、干净的日志会被立即提交到日志中，所以只需要一个额外的日志段空间（不是日志的完整副本）
8. 日志 head 的概念本质上是一个空间密集的 hash 表，每个条目使用 24 个字节。所以如果有 8G 的整理缓冲区，则能迭代处理大约 336G 的 log head （假设消息大小为 1k）

配置 Log Cleaner

The log cleaner is disabled by default. To enable it set the server config

log cleaner 默认是关闭的，可以通过以下服务端配置开启：

```
log.cleaner.enable=true
```

This will start the pool of cleaner threads. To enable log cleaning on a particular topic you can add the log-specific property

这会启动清理线程池。如果要开启特定 topic 的清理功能，需要开启特定的 log-specific 属性

```
log.cleanup.policy=compact
```

This can be done either at topic creation time or using the alter topic command.

这个可以通过创建 topic 时配置或者之后使用 topic 命令实现。

更多的关于 cleaner 的配置可以从这里找到。

Log Compaction Limitations

1. You cannot configure yet how much log is retained without compaction (the "head" of the log). Currently all segments are eligible except for the last segment, i.e. the one currently being written to.

4.9 配额

Starting in 0.9, the Kafka cluster has the ability to enforce quotas on produce and fetch requests. Quotas are basically byte-rate thresholds defined per client-id. A client-id logically identifies an application making a request. Hence a single client-id can span multiple producer and consumer instances and the quota will apply for all of them as a single entity i.e. if client-id="test-client" has a produce quota of 10MB/sec, this is shared across all instances with that same id.

Why are quotas necessary?

It is possible for producers and consumers to produce/consume very high volumes of data and thus monopolize broker resources, cause network saturation and generally DOS other clients and the brokers themselves. Having quotas protects against these issues and is all the more important in large multi-tenant clusters where a small set of badly behaved clients can degrade user experience for the well behaved ones. In fact, when running Kafka as a service this even makes it possible to enforce API limits according to an agreed upon contract.

producers 和 consumer 可能会产生和消费大量的消息从而导致独占 broker 资源，进而引起网络饱和，对其他 client 和 broker 造成 DOS 攻击。资源的配额保护可以有效的防止这些问题，大型的多租户集群中，因为一小部分表现不佳的客户端降低了良好的用户体验，这种情况下非常需要资源的配额保护。实际情况中，当把 Kafka 当做一种服务提供时，可以根据客户端和服务端的契约对 API 调用做限制。

Enforcement

By default, each unique client-id receives a fixed quota in bytes/sec as configured by the cluster (quota.producer.default, quota.consumer.default). This quota is defined on a per-broker basis. Each client can publish/fetch a maximum of X bytes/sec per broker before it gets throttled. We decided that defining these quotas per broker is much better than having a fixed cluster wide bandwidth per client because that would require a mechanism to share client quota usage among all the brokers. This can be harder to get right than the quota implementation itself!

默认情况下，每个唯一的客户端分组在集群上配置一个固定的限额，这个限额是基于每台服务器的 (quota.producer.default, quota.consumer.default)，每个客户端能发布或获取每台服务器都的最大速率，我们按服务器 (broker) 定义配置，而不是按整个集群定义，是因为如果是集群范围需要额外的机制来共享配额的使用情况，这会导致配额机制的实现比较难。

How does a broker react when it detects a quota violation? In our solution, the broker does not return an error rather it attempts to slow down a client exceeding its quota. It computes the amount of delay needed to bring a guilty client under it's quota and delays the response for that time. This approach keeps the quota violation transparent to clients (outside of client-

side metrics). This also keeps them from having to implement any special backoff and retry behavior which can get tricky. In fact, bad client behavior (retry without backoff) can exacerbate the very problem quotas are trying to solve.

当 broker 检测到超过配额时如何反应？在我们的解决方案中，broker 不会返回错误，相反他会尝试降低超过限额的客户端速度，它计算将超过限额客户端拉回到正常水平的时间，并相应的延迟响应时间。这个方法让超出配额的处理变得透明化。这个方法同样让客户端免于处理棘手的重试和特殊的补救措施。事实上，错误的补救措施可能加重限额这个问题。

Client byte rate is measured over multiple small windows (e.g. 30 windows of 1 second each) in order to detect and correct quota violations quickly. Typically, having large measurement windows (for e.g. 10 windows of 30 seconds each) leads to large bursts of traffic followed by long delays which is not great in terms of user experience.

客户端的字节限速使用多个小时间窗口（每秒 30 个窗口）来快速检测和更正配额越界。如果使用太大的配额窗口（例如 30 秒 10 个窗口），容易导致在较长时间内有巨大的流量突增，这个在实际中用户体验并不好。

Quota overrides

It is possible to override the default quota for client-ids that need a higher (or even lower) quota. The mechanism is similar to the per-topic log config overrides. Client-id overrides are written to ZooKeeper under***/config/clients***. These overrides are read by all brokers and are effective immediately. This lets us change quotas without having to do a rolling restart of the entire cluster. See **here** for details.

覆盖 client-ids 默认的配额是可行的。这个机制类似于每一个 topic 日志的配置覆盖。client-id 覆盖会被写到 ZooKeeper，这个覆盖会被所有的 broker 读取并且迅速加载生效。这样使得我们可以不需要重启集群中的机器而快速的改变配额。点击 这里 查看更多的信息。

4. Design

4.1 Motivation

We designed Kafka to be able to act as a unified platform for handling all the real-time data feeds **a large company might have**. To do this we had to think through a fairly broad set of use cases.

It would have to have high-throughput to support high volume event streams such as real-time log aggregation.

It would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.

It also meant the system would have to handle low-latency delivery to handle more traditional messaging use-cases.

We wanted to support partitioned, distributed, real-time processing of these feeds to create new, derived feeds. This motivated our partitioning and consumer model.

Finally in cases where the stream is fed into other data systems for serving, we knew the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

Supporting these uses led us to a design with a number of unique elements, more akin to a database log than a traditional messaging system. We will outline some elements of the design in the following sections.

4.2 Persistence

Don't fear the filesystem!

Kafka relies heavily on the filesystem for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a **JBOD** configuration with six 7200rpm SATA RAID-5 array is about 600MB\/sec but the performance of random writes is only about 100k\/sec—a difference of over 6000X.

These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this **ACM Queue article**; they actually find that**sequential disk access can in some cases be faster than random memory access!**

To compensate for this performance divergence, modern operating systems have become increasingly aggressive in their use of main memory for disk caching. A modern OS will happily divert *all* free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I\/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice.

Furthermore we are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

1. The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
2. Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure—we at least double the available cache by having automatic access to all free memory, and likely double again by storing a compact byte structure rather than individual objects. Doing so will result in a cache of up to 28-30GB on a 32GB machine without GC penalties. Furthermore this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory (which for a 10GB cache may take 10 minutes) or else it will need to start with a completely cold cache (which likely means terrible initial performance). This also greatly simplifies the code as all logic for maintaining coherency between the cache and filesystem is now in the OS, which tends to do so more efficiently and more correctly than one-off in-process attempts. If your disk usage favors linear reads then read-ahead is effectively pre-populating this cache with useful data on each disk read.

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.

This style of pagecache-centric design is described in an **article** on the design of Varnish here (along with a healthy dose of arrogance).

Constant Time Suffices

The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTree or other general-purpose random access data structures to maintain metadata about messages. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. They do come with a fairly high cost, though: Btree operations are O(log N). Normally O(log N) is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at 10 ms a pop, and each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead. Since storage systems mix very fast cached operations with very slow physical disk operations, the observed performance of tree structures is often superlinear as data increases with fixed cache--i.e. doubling your data makes things much worse then twice as slow.

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions. This structure has the advantage that all operations are O(1) and reads do not block writes or each other. This has obvious performance advantages since the performance is completely decoupled from the data size—one server can now take full advantage of a number of cheap, low-rotational speed 1+TB SATA drives. Though they have poor seek performance, these drives have acceptable performance for large reads and writes and come at 1⁄3 the price and 3x the capacity.

Having access to virtually unlimited disk space without any performance penalty means that we can provide some features not usually found in a messaging system. For example, in Kafka, instead of attempting to delete messages as soon as they are consumed, we can retain messages for a relatively long period (say a week). This leads to a great deal of flexibility for consumers, as we will describe.

4.3 Efficiency

We have put significant effort into efficiency. One of our primary use cases is handling web activity data, which is very high volume: each page view may generate dozens of writes. Furthermore we assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.

We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the

application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.

We discussed disk efficiency in the previous section. Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system: too many small I\/O operations, and excessive byte copying.

The small I\/O problem happens both between the client and the server and in the server's own persistent operations.

To avoid this, our protocol is built around a "message set" abstraction that naturally groups messages together. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time. The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.

This simple optimization produces orders of magnitude speed up. Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on, all of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.

The other inefficiency is in byte copying. At low message rates this is not an issue, but under load the impact is significant. To avoid this we employ a standardized binary message format that is shared by the producer, the broker, and the consumer (so data chunks can be transferred without modification between them).

The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer. Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the **sendfile system call**.

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space
2. The application reads the data from kernel space into a user-space buffer
3. The application writes the data back into kernel space into a socket buffer
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network

This is clearly inefficient, there are four copies and two system calls. Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.

We expect a common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to kernel space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.

This combination of pagecache and sendfile means that on a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.

For more background on the sendfile and zero-copy support in Java, see this **article**.

End-to-end Batch Compression

In some cases the bottleneck is actually not CPU or disk but network bandwidth. This is particularly true for a data pipeline that needs to send messages between data centers over a wide-area network. Of course the user can always compress its messages one at a time without any support needed from Kafka, but this can lead to very poor compression ratios as much of the redundancy is due to repetition between messages of the same type (e.g. field names in JSON or user agents in web logs or common string values). Efficient compression requires compressing multiple messages together rather than compressing each message individually.

Kafka supports this by allowing recursive message sets. A batch of messages can be clumped together compressed and sent to the server in this form. This batch of messages will be written in compressed form and will remain compressed in the log and will only be decompressed by the consumer.

Kafka supports GZIP, Snappy and LZ4 compression protocols. More details on compression can be found **here**.

4.4 The Producer

Load balancing

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this all Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriately direct its requests.

The client controls which partition it publishes messages to. This can be done at random, implementing a kind of random load balancing, or it can be done by some semantic partitioning function. We expose the interface for semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition (there is also an option to override the partition function if need be). For example if the key chosen was a user id then all data for a given user would be sent to the same partition. This in turn will allow consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in consumers.

Asynchronous send

Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say 64k or 10 ms). This allows the accumulation of more bytes to send, and few larger I\/O operations on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

Details on **configuration** and the **api** for the producer can be found elsewhere in the documentation.

4.5 The Consumer

The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

Push vs. pull

An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as **Scribe** and **Apache Flume**, follow a very different push-based path where data is pushed downstream. There are pros and cons to both approaches. However, a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal is generally for the consumer to be able to consume at the maximum possible rate; unfortunately, in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of

service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

Another advantage of a pull-based system is that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency, this will result in sending a single message at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the consumer always pulls all available messages after its current position in the log (or up to some configurable max size). So one gets optimal batching without introducing unnecessary latency.

The deficiency of a naive pull-based system is that if the broker has no data the consumer may end up polling in a tight loop, effectively busy-waiting for data to arrive. To avoid this we have parameters in our pull request that allow the consumer request to block in a "long poll" waiting until data arrives (and optionally waiting until a given number of bytes is available to ensure large transfer sizes).

You could imagine other possible designs which would be only pull, end-to-end. The producer would locally write to a local log, and brokers would pull from that with consumers pulling from them. A similar type of "store-and-forward" producer is often proposed. This is intriguing but we felt not very suitable for our target use cases which have thousands of producers. Our experience running persistent data systems at scale led us to feel that involving thousands of disks in the system across many applications would not actually make things more reliable and would be a nightmare to operate. And in practice we have found that we can run a pipeline with strong SLAs at large scale without a need for producer persistence.

Consumer Position

Keeping track of *what* has been consumed is, surprisingly, one of the key performance points of a messaging system.

Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally immediately or it may wait for acknowledgement from the consumer. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else this

state could go. Since the data structures used for storage in many messaging systems scale poorly, this is also a pragmatic choice--since the broker knows what is consumed it can immediately delete it, keeping the data size small.

What is perhaps not obvious is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as **consumed** immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as **sent** not **consumed** when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as**consumed**. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

Kafka handles this differently. Our topic is divided into a set of totally ordered partitions, each of which is consumed by one consumer at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.

There is a side benefit of this decision. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

Offline Data Load

Scalable persistence allows for the possibility of consumers that only periodically consume such as batch data loads that periodically bulk-load data into an offline system such as Hadoop or a relational data warehouse.

In the case of Hadoop we parallelize the data load by splitting the load over individual map tasks, one for each node\/topic\/partition combination, allowing full parallelism in the loading. Hadoop provides the task management, and tasks which fail can restart without danger of duplicate data—they simply restart from their original position.

4.6 Message Delivery Semantics

Now that we understand a little about how producers and consumers work, let's discuss the semantic guarantees Kafka provides between producer and consumer. Clearly there are multiple possible message delivery guarantees that could be provided:

- *At most once*—Messages may be lost but are never redelivered.
- *At least once*—Messages are never lost but may be redelivered.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

Many systems claim to provide "exactly once" delivery semantics, but it is important to read the fine print, most of these claims are misleading (i.e. they don't translate to the case where consumers or producers can fail, cases where there are multiple consumer processes, or cases where data written to disk can be lost).

Kafka's semantics are straight-forward. When publishing a message we have a notion of the message being "committed" to the log. Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive". The definition of alive as well as a description of which types of failures we attempt to handle will be described in more detail in the next section. For now let's assume a perfect, lossless broker and try to understand the guarantees to the producer and consumer. If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed. This is similar to the semantics of inserting into a database table with an autogenerated key.

These are not the strongest possible semantics for publishers. Although we cannot be sure of what happened in the case of a network error, it is possible to allow the producer to generate a sort of "primary key" that makes retrying the produce request idempotent. This feature is not trivial for a replicated system because of course it must work even (or especially) in the case of a server failure. With this feature it would suffice for the producer to retry until it receives acknowledgement of a successfully committed message at which point we would guarantee the message had been published exactly once. We hope to add this in a future Kafka version.

Not all use cases require such strong guarantees. For uses which are latency sensitive we allow the producer to specify the durability level it desires. If the producer specifies that it wants to wait on the message being committed this can take on the order of 10 ms. However

the producer can also specify that it wants to perform the send completely asynchronously or that it wants to wait only until the leader (but not necessarily the followers) have the message.

Now let's describe the semantics from the point-of-view of the consumer. All replicas have the exact same log with the same offsets. The consumer controls its position in this log. If the consumer never crashed it could just store this position in memory, but if the consumer fails and we want this topic partition to be taken over by another process the new process will need to choose an appropriate position from which to start processing. Let's say the consumer reads some messages -- it has several options for processing the messages and updating its position.

1. It can read the messages, then save its position in the log, and finally process the messages. In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing. In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed. This corresponds to "at-most-once" semantics as in the case of a consumer failure messages may not be processed.
2. It can read the messages, process the messages, and finally save its position. In this case there is a possibility that the consumer process crashes after processing messages but before saving its position. In this case when the new process takes over the first few messages it receives will already have been processed. This corresponds to the "at-least-once" semantics in the case of consumer failure. In many cases messages have a primary key and so the updates are idempotent (receiving the same message twice just overwrites a record with another copy of itself).
3. So what about exactly once semantics (i.e. the thing you actually want)? The limitation here is not actually a feature of the messaging system but rather the need to co-ordinate the consumer's position with what is actually stored as output. The classic way of achieving this would be to introduce a two-phase commit between the storage for the consumer position and the storage of the consumers output. But this can be handled more simply and generally by simply letting the consumer store its offset in the same place as its output. This is better because many of the output systems a consumer might want to write to will not support a two-phase commit. As an example of this, our Hadoop ETL that populates data in HDFS stores its offsets in HDFS with the data it reads so that it is guaranteed that either data and offsets are both updated or neither is. We follow similar patterns for many other data systems which require these stronger semantics and for which the messages do not have a primary key to allow for deduplication.

So effectively Kafka guarantees at-least-once delivery by default and allows the user to implement at most once delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages. Exactly-once delivery requires co-operation with the destination storage system but Kafka provides the offset which makes implementing this straight-forward.

4.7 Replication

Kafka replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: slaves are inactive, throughput is heavily impacted, it requires fiddly manual configuration, etc. Kafka is meant to be used with replication by default—in fact we implement un-replicated topics as replicated topics where the replication factor is one.

The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log—all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).

Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions

1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)
2. If it is a slave it must replicate the writes happening on the leader and not fall "too far" behind

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The

determination of stuck and lagging replicas is controlled by the replica.lag.time.max.ms configuration.

In distributed systems terminology we only attempt to handle a "fail\/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). Kafka does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).

A message is considered "committed" when all in sync replicas for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the acks setting that the producer uses.

The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.

Kafka will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

Replicated Logs: Quorums, ISRs, and State Machines (Oh my!)

At its heart a Kafka partition is a replicated log. The replicated log is one of the most basic primitives in distributed data systems, and there are many approaches for implementing one. A replicated log can be used by other systems as a primitive for implementing other distributed systems in the **state-machine style**.

A replicated log models the process of coming into consensus on the order of a series of values (generally numbering the log entries 0, 1, 2, ...). There are many ways to implement this, but the simplest and fastest is with a leader who chooses the ordering of values provided to it. As long as the leader remains alive, all followers need to only copy the values and ordering the leader chooses.

Of course if leaders didn't fail we wouldn't need followers! When the leader does die we need to choose a new leader from among the followers. But followers themselves may fall behind or crash so we must ensure we choose an up-to-date follower. The fundamental guarantee a log replication algorithm must provide is that if we tell the client a message is committed, and the leader fails, the new leader we elect must also have that message. This yields a tradeoff: if the leader waits for more followers to acknowledge a message before declaring it committed then there will be more potentially electable leaders.

If you choose the number of acknowledgements required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap, then this is called a Quorum.

A common approach to this tradeoff is to use a majority vote for both the commit decision and the leader election. This is not what Kafka does, but let's explore it anyway to understand the tradeoffs. Let's say we have 2*f*+1 replicas. If *f*+1 replicas must receive a message prior to a commit being declared by the leader, and if we elect a new leader by electing the follower with the most complete log from at least *f*+1 replicas, then, with no more than *f* failures, the leader is guaranteed to have all committed messages. This is because among any *f*+1 replicas, there must be at least one replica that contains all committed messages. That replica's log will be the most complete and therefore will be selected as the new leader. There are many remaining details that each algorithm must handle (such as precisely defined what makes a log more complete, ensuring log consistency during leader failure or changing the set of servers in the replica set) but we will ignore these for now.

This majority vote approach has a very nice property: the latency is dependent on only the fastest servers. That is, if the replication factor is three, the latency is determined by the faster slave not the slower one.

There are a rich variety of algorithms in this family including ZooKeeper's **Zab**, **Raft**, and **Viewstamped Replication**. The most similar academic publication we are aware of to Kafka's actual implementation is**PacificA** from Microsoft.

The downside of majority vote is that it doesn't take many failures to leave you with no electable leaders. To tolerate one failure requires three copies of the data, and to tolerate two failures requires five copies of the data. In our experience having only enough redundancy to tolerate a single failure is not enough for a practical system, but doing every write five times, with 5x the disk space requirements and 1\/5th the throughput, is not very practical for large volume data problems. This is likely why quorum algorithms more commonly appear for shared cluster configuration such as ZooKeeper but are less common for primary data storage. For example in HDFS the namenode's high-availability feature is built on a **majority-vote-based journal**, but this more expensive approach is not used for the data itself.

Kafka takes a slightly different approach to choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important. With this ISR model and *f+1* replicas, a Kafka topic can tolerate *f* failures without losing committed messages.

For most use cases we hope to handle, we think this tradeoff is a reasonable one. In practice, to tolerate _f_failures, both the majority vote and the ISR approach will wait for the same number of replicas to acknowledge before committing a message (e.g. to survive one failure a majority quorum needs three replicas and one acknowledgement and the ISR approach requires two replicas and one acknowledgement). The ability to commit without the slowest servers is an advantage of the majority vote approach. However, we think it is ameliorated by allowing the client to choose whether they block on the message commit or not, and the additional throughput and disk space due to the lower required replication factor is worth it.

Another important design distinction is that Kafka does not require that crashed nodes recover with all their data intact. It is not uncommon for replication algorithms in this space to depend on the existence of "stable storage" that cannot be lost in any failure-recovery scenario without potential consistency violations. There are two primary problems with this assumption. First, disk errors are the most common problem we observe in real operation of persistent data systems and they often do not leave data intact. Secondly, even if this were not a problem, we do not want to require the use of fsync on every write for our consistency guarantees as this can reduce performance by two to three orders of magnitude. Our protocol for allowing a replica to rejoin the ISR ensures that before rejoining, it must fully re-sync again even if it lost unflushed data in its crash.

Unclean leader election: What if they all die?

Note that Kafka's guarantee with respect to data loss is predicated on at least one replica remaining in sync. If all the nodes replicating a partition die, this guarantee no longer holds.

However a practical system needs to do something reasonable when all the replicas die. If you are unlucky enough to have this occur, it is important to consider what will happen. There are two behaviors that could be implemented:

1. Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).
2. Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

This is a simple tradeoff between availability and consistency. If we wait for replicas in the ISR, then we will remain unavailable as long as those replicas are down. If such replicas were destroyed or their data was lost, then we are permanently down. If, on the other hand, a non-in-sync replica comes back to life and we allow it to become leader, then its log becomes the source of truth even though it is not guaranteed to have every committed message. By default Kafka chooses the second strategy and favor choosing a potentially

inconsistent replica when all replicas in the ISR are dead. This behavior can be disabled using configuration property unclean.leader.election.enable, to support use cases where downtime is preferable to inconsistency.

This dilemma is not specific to Kafka. It exists in any quorum-based scheme. For example in a majority voting scheme, if a majority of servers suffer a permanent failure, then you must either choose to lose 100% of your data or violate consistency by taking what remains on an existing server as your new source of truth.

Availability and Durability Guarantees

When writing to Kafka, producers can choose whether they wait for the message to be acknowledged by 0,1 or all (-1) replicas. Note that "acknowledgement by all replicas" does not guarantee that the full set of assigned replicas have received the message. By default, when acks=all, acknowledgement happens as soon as all the current in-sync replicas have received the message. For example, if a topic is configured with only two replicas and one fails (i.e., only one in sync replica remains), then writes that specify acks=all will succeed. However, these writes could be lost if the remaining replica also fails. Although this ensures maximum availability of the partition, this behavior may be undesirable to some users who prefer durability over availability. Therefore, we provide two topic-level configurations that can be used to prefer message durability over availability:

1. Disable unclean leader election - if all replicas become unavailable, then the partition will remain unavailable until the most recent leader becomes available again. This effectively prefers unavailability over the risk of message loss. See the previous section on Unclean Leader Election for clarification.
2. Specify a minimum ISR size - the partition will only accept writes if the size of the ISR is above a certain minimum, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses acks=all and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between consistency and availability. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.

Replica Management

The above discussion on replicated logs really covers only a single log, i.e. one topic partition. However a Kafka cluster will manage hundreds or thousands of these partitions. We attempt to balance partitions within a cluster in a round-robin fashion to avoid clustering all partitions for high-volume topics on a small number of nodes. Likewise we try to balance leadership so that each node is the leader for a proportional share of its partitions.

It is also important to optimize the leadership election process as that is the critical window of unavailability. A naive implementation of leader election would end up running an election per partition for all partitions a node hosted when that node failed. Instead, we elect one of the brokers as the "controller". This controller detects failures at the broker level and is responsible for changing the leader of all affected partitions in a failed broker. The result is that we are able to batch together many of the required leadership change notifications which makes the election process far cheaper and faster for a large number of partitions. If the controller fails, one of the surviving brokers will become the new controller.

4.8 Log Compaction

Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance. Let's dive into these use cases in more detail and then describe how compaction works.

So far we have described only the simpler approach to data retention where old log data is discarded after a fixed period of time or when the log reaches some predetermined size. This works well for temporal event data such as logging where each record stands alone. However an important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a database table).

Let's discuss a concrete example of such a stream. Say we have a topic containing user email addresses; every time a user updates their email address we send a message to this topic using their user id as the primary key. Now say we send the following messages over some time period for a user with id 123, each message corresponding to a change in email address (messages for other ids are omitted):

```
123 => bill@microsoft.com
        .
        .
        .
123 => bill@gatesfoundation.org
        .
        .
        .
123 => bill@gmail.com
```

Log compaction gives us a more granular retention mechanism so that we are guaranteed to retain at least the last update for each primary key (e.g. `bill@gmail.com`). By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

Let's start by looking at a few use cases where this is useful, then we'll see how it can be used.

1. *Database change subscription*. It is often necessary to have a data set in multiple data systems, and often one of these systems is a database of some kind (either a RDBMS or perhaps a new-fangled key-value store). For example you might have a database, a cache, a search cluster, and a Hadoop cluster. Each change to the database will need to be reflected in the cache, the search cluster, and eventually in Hadoop. In the case that one is only handling the real-time updates you only need recent log. But if you want to be able to reload the cache or restore a failed search node you may need a complete data set.
2. *Event sourcing*. This is a style of application design which co-locates query processing with application design and uses a log of changes as the primary store for the application.
3. *Journaling for high-availability*. A process that does local computation can be made fault-tolerant by logging out changes that it makes to it's local state so another process can reload these changes and carry on if it should fail. A concrete example of this is handling counts, aggregations, and other "group by"-like processing in a stream query system. Samza, a real-time stream-processing framework, **uses this feature** for exactly this purpose.

In each of these cases one needs primarily to handle the real-time feed of changes, but occasionally, when a machine crashes or data needs to be re-loaded or re-processed, one needs to do a full load. Log compaction allows feeding both of these use cases off the same backing topic. This style of usage of a log is described in more detail in **this blog post**.

The general idea is quite simple. If we had infinite log retention, and we logged each change in the above cases, then we would have captured the state of the system at each time from when it first began. Using this complete log, we could restore to any point in time by replaying the first N records in the log. This hypothetical complete log is not very practical for systems that update a single record many times as the log will grow without bound even for a stable dataset. The simple log retention mechanism which throws away old updates will bound space but the log is no longer a way to restore the current state—now restoring from the beginning of the log no longer recreates the current state as old updates may not be captured at all.

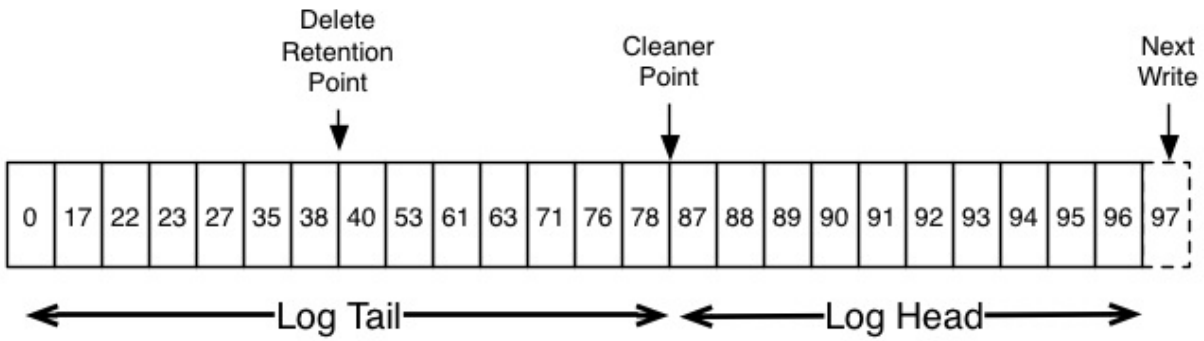
Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

This functionality is inspired by one of LinkedIn's oldest and most successful pieces of infrastructure—a database changelog caching service called **Databus**. Unlike most log-structured storage systems Kafka is built for subscription and organizes data for fast linear reads and writes. Unlike Databus, Kafka acts as a source-of-truth store so it is useful even in situations where the upstream data source would not otherwise be replayable.

Log Compaction Basics

Here is a high-level picture that shows the logical structure of a Kafka log with the offset for each message.

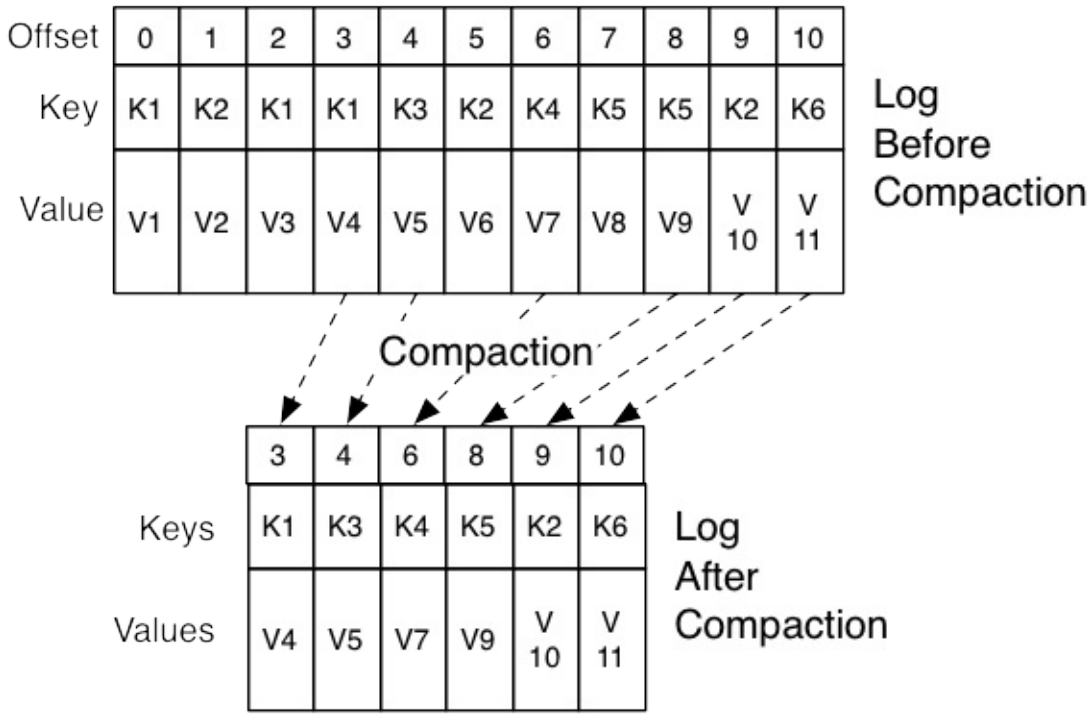


The head of the log is identical to a traditional Kafka log. It has dense, sequential offsets and retains all messages. Log compaction adds an option for handling the tail of the log. The picture above shows a log with a compacted tail. Note that the messages in the tail of the log retain the original offset assigned when they were first written—that never changes. Note also that all offsets remain valid positions in the log, even if the message with that offset has been compacted away; in this case this position is indistinguishable from the next highest offset that does appear in the log. For example, in the picture above the offsets 36, 37, and 38 are all equivalent positions and a read beginning at any of these offsets would return a message set beginning with 38.

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that

they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "delete retention point" in the above diagram.

The compaction is done in the background by periodically recopying log segments. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I\/O throughput to avoid impacting producers and consumers. The actual process of compacting a log segment looks something like this:



What guarantees does log compaction provide?

Log compaction guarantees the following:

1. Any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets.
2. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.
3. The offset for a message never changes. It is the permanent identifier for a position in the log.
4. Any read progressing from offset 0 will see at least the final state of all records in the order they were written. All delete markers for deleted records will be seen provided the reader reaches the head of the log in a time period less than the topic's delete.retention.ms setting (the default is 24 hours). This is important as delete marker removal happens concurrently with read (and thus it is important that we not remove any delete marker prior to the reader seeing it).

5. Any consumer progressing from the start of the log will see at least the *final* state of all records in the order they were written. All delete markers for deleted records will be seen provided the consumer reaches the head of the log in a time period less than the topic's `delete.retention.ms` setting (the default is 24 hours). This is important as delete marker removal happens concurrently with read, and thus it is important that we do not remove any delete marker prior to the consumer seeing it.

Log Compaction Details

Log compaction is handled by the log cleaner, a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Each compactor thread works as follows:

1. It chooses the log that has the highest ratio of log head to log tail
2. It creates a succinct summary of the last offset for each key in the head of the log
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log. New, clean segments are swapped into the log immediately so the additional disk space required is just one additional log segment (not a fully copy of the log).
4. The summary of the log head is essentially just a space-compact hash table. It uses exactly 24 bytes per entry. As a result with 8GB of cleaner buffer one cleaner iteration can clean around 366GB of log head (assuming 1k messages).

Configuring The Log Cleaner

The log cleaner is disabled by default. To enable it set the server config

```
log.cleaner.enable=true
```

This will start the pool of cleaner threads. To enable log cleaning on a particular topic you can add the log-specific property

```
log.cleanup.policy=compact
```

This can be done either at topic creation time or using the alter topic command.

Further cleaner configurations are described **here**.

Log Compaction Limitations

1. You cannot configure yet how much log is retained without compaction (the "head" of the log). Currently all segments are eligible except for the last segment, i.e. the one

currently being written to.

4.9 Quotas

Starting in 0.9, the Kafka cluster has the ability to enforce quotas on produce and fetch requests. Quotas are basically byte-rate thresholds defined per client-id. A client-id logically identifies an application making a request. Hence a single client-id can span multiple producer and consumer instances and the quota will apply for all of them as a single entity i.e. if client-id="test-client" has a produce quota of 10MB\/sec, this is shared across all instances with that same id.

Why are quotas necessary?

It is possible for producers and consumers to produce\/consume very high volumes of data and thus monopolize broker resources, cause network saturation and generally DOS other clients and the brokers themselves. Having quotas protects against these issues and is all the more important in large multi-tenant clusters where a small set of badly behaved clients can degrade user experience for the well behaved ones. In fact, when running Kafka as a service this even makes it possible to enforce API limits according to an agreed upon contract.

Enforcement

By default, each unique client-id receives a fixed quota in bytes\/sec as configured by the cluster (quota.producer.default, quota.consumer.default). This quota is defined on a per-broker basis. Each client can publish\/fetch a maximum of X bytes\/sec per broker before it gets throttled. We decided that defining these quotas per broker is much better than having a fixed cluster wide bandwidth per client because that would require a mechanism to share client quota usage among all the brokers. This can be harder to get right than the quota implementation itself!

How does a broker react when it detects a quota violation? In our solution, the broker does not return an error rather it attempts to slow down a client exceeding its quota. It computes the amount of delay needed to bring a guilty client under it's quota and delays the response for that time. This approach keeps the quota violation transparent to clients (outside of client-side metrics). This also keeps them from having to implement any special backoff and retry behavior which can get tricky. In fact, bad client behavior (retry without backoff) can exacerbate the very problem quotas are trying to solve.

Client byte rate is measured over multiple small windows (e.g. 30 windows of 1 second each) in order to detect and correct quota violations quickly. Typically, having large measurement windows (for e.g. 10 windows of 30 seconds each) leads to large bursts of traffic followed by long delays which is not great in terms of user experience.

Quota overrides

It is possible to override the default quota for client-ids that need a higher (or even lower) quota. The mechanism is similar to the per-topic log config overrides. Client-id overrides are written to ZooKeeper under***/config/clients***. These overrides are read by all brokers and are effective immediately. This lets us change quotas without having to do a rolling restart of the entire cluster. See **here** for details.

5. Implementation

5.1 API Design

Producer APIs

The Producer API that wraps the 2 low-level producers - `kafka.producer.SyncProducer` and `kafka.producer.async.AsyncProducer` .

```
class Producer {

  /* Sends the data, partitioned by key to the topic using either the */
  /* synchronous or the asynchronous producer */
  public void send(kafka.javaapi.producer.ProducerData<K,V> producerData);

  /* Sends a list of data, partitioned by key to the topic using either */
  /* the synchronous or the asynchronous producer */
  public void send(java.util.List<kafka.javaapi.producer.ProducerData<K,V>> producerDa
ta);

  /* Closes the producer and cleans up */
  public void close();

}
```

The goal is to expose all the producer functionality through a single API to the client. The new producer -

- can handle queueing\/buffering of multiple producer requests and asynchronous dispatch of the batched data - `kafka.producer.Producer` provides the ability to batch multiple produce requests (`producer.type=async`), before serializing and dispatching them to the appropriate kafka broker partition. The size of the batch can be controlled by a few config parameters. As events enter a queue, they are buffered in a queue, until either `queue.time` or `batch.size` is reached. A background thread (`kafka.producer.async.ProducerSendThread`) dequeues the batch of data and lets the `kafka.producer.EventHandler` serialize and send the data to the appropriate kafka broker partition. A custom event handler can be plugged in through the `event.handler` config parameter. At various stages of this producer queue pipeline, it is helpful to be able to inject callbacks, either for plugging in custom logging\/tracing code or custom monitoring logic. This is possible by implementing the `kafka.producer.async.CallbackHandler` interface and setting `callback.handler` config parameter to that class.

- handles the serialization of data through a user-specified `Encoder`:

```
interface Encoder<T> {
  public Message toMessage(T data);
}
```

  The default is the no-op `kafka.serializer.DefaultEncoder`

- provides software load balancing through an optionally user-specified `Partitioner`: The routing decision is influenced by the `kafka.producer.Partitioner`.

```
interface Partitioner<T> {
   int partition(T key, int numPartitions);
}
```

  The partition API uses the key and the number of available broker partitions to return a partition id. This id is used as an index into a sorted list of broker_ids and partitions to pick a broker partition for the producer request. The default partitioning strategy is `hash(key)%numPartitions`. If the key is null, then a random broker partition is picked. A custom partitioning strategy can also be plugged in using the `partitioner.class` config parameter.

Consumer APIs

We have 2 levels of consumer APIs. The low-level "simple" API maintains a connection to a single broker and has a close correspondence to the network requests sent to the server. This API is completely stateless, with the offset being passed in on every request, allowing the user to maintain this metadata however they choose.

The high-level API hides the details of brokers from the consumer and allows consuming off the cluster of machines without concern for the underlying topology. It also maintains the state of what has been consumed. The high-level API also provides the ability to subscribe to topics that match a filter expression (i.e., either a whitelist or a blacklist regular expression).

Low-level API

```
class SimpleConsumer {
  /* Send fetch request to a broker and get back a set of messages. */
  public ByteBufferMessageSet fetch(FetchRequest request);
  /* Send a list of fetch requests to a broker and get back a response set. */
  public MultiFetchResponse multifetch(List<FetchRequest> fetches);
  /**
   * Get a list of valid offsets (up to maxSize) before the given time.
   * The result is a list of offsets, in descending order.
   * @param time: time in millisecs,
   *              if set to OffsetRequest$.MODULE$.LATEST_TIME(), get from the latest
offset available.
   *              if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the earli
est offset available.
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOff
sets);
}
```

The low-level API is used to implement the high-level API as well as being used directly for some of our offline consumers which have particular requirements around maintaining state.

High-level API

```
/* create a connection to the cluster */
ConsumerConnector connector = Consumer.create(consumerConfig);

interface ConsumerConnector {

  /**
   * This method is used to get a list of KafkaStreams, which are iterators over
   * MessageAndMetadata objects from which you can obtain messages and their
   * associated metadata (currently only topic).
   *  Input: a map of <topic, #streams>
   *  Output: a map of <topic, list of message streams>
   */
  public Map<String,List<KafkaStream>> createMessageStreams(Map<String,Int> topicCount
Map);

  /**
   * You can also obtain a list of KafkaStreams, that iterate over messages
   * from topics that match a TopicFilter. (A TopicFilter encapsulates a
   * whitelist or a blacklist which is a standard Java regex.)
   */
  public List<KafkaStream> createMessageStreamsByFilter(
      TopicFilter topicFilter, int numStreams);

  /* Commit the offsets of all messages consumed so far. */
  public commitOffsets()

  /* Shut down the connector */
  public shutdown()
}
```

This API is centered around iterators, implemented by the KafkaStream class. Each KafkaStream represents the stream of messages from one or more partitions on one or more servers. Each stream is used for single threaded processing, so the client can provide the number of desired streams in the create call. Thus a stream may represent the merging of multiple server partitions (to correspond to the number of processing threads), but each partition only goes to one stream.

The createMessageStreams call registers the consumer for the topic, which results in rebalancing the consumer\/broker assignment. The API encourages creating many topic streams in a single call in order to minimize this rebalancing. The createMessageStreamsByFilter call (additionally) registers watchers to discover new topics that match its filter. Note that each stream that createMessageStreamsByFilter returns may iterate over messages from multiple topics (i.e., if multiple topics are allowed by the filter).

5.2 Network Layer

The network layer is a fairly straight-forward NIO server, and will not be described in great detail. The sendfile implementation is done by giving the `MessageSet` interface a `writeTo` method. This allows the file-backed message set to use the more efficient `transferTo` implementation instead of an in-process buffered write. The threading model is a single acceptor thread and *N* processor threads which handle a fixed number of connections each. This design has been pretty thoroughly tested **elsewhere** and found to be simple to implement and fast. The protocol is kept quite simple to allow for future implementation of clients in other languages.

5.3 Messages

Messages consist of a fixed-size header, a variable length opaque key byte array and a variable length opaque value byte array. The header contains the following fields:

- A CRC32 checksum to detect corruption or truncation.
-
- A format version.
- An attributes identifier
- A timestamp

Leaving the key and value opaque is the right decision: there is a great deal of progress being made on serialization libraries right now, and any particular choice is unlikely to be right for all uses. Needless to say a particular application using Kafka would likely mandate a particular serialization type as part of its usage. The `MessageSet` interface is simply an iterator over messages with specialized methods for bulk reading and writing to an NIO `Channel`.

5.4 Message Format

```
    /**
     * 1. 4 byte CRC32 of the message
     * 2. 1 byte "magic" identifier to allow format changes, value is 0 or 1
     * 3. 1 byte "attributes" identifier to allow annotations on the message independe
nt of the version
     *    bit 0 ~ 2 : Compression codec.
     *      0 : no compression
     *      1 : gzip
     *      2 : snappy
     *      3 : lz4
     *    bit 3 : Timestamp type
     *      0 : create time
     *      1 : log append time
     *    bit 4 ~ 7 : reserved
     * 4. (Optional) 8 byte timestamp only if "magic" identifier is greater than 0
     * 5. 4 byte key length, containing length K
     * 6. K byte key
     * 7. 4 byte payload length, containing length V
     * 8. V byte payload
     */
```

5.5 Log

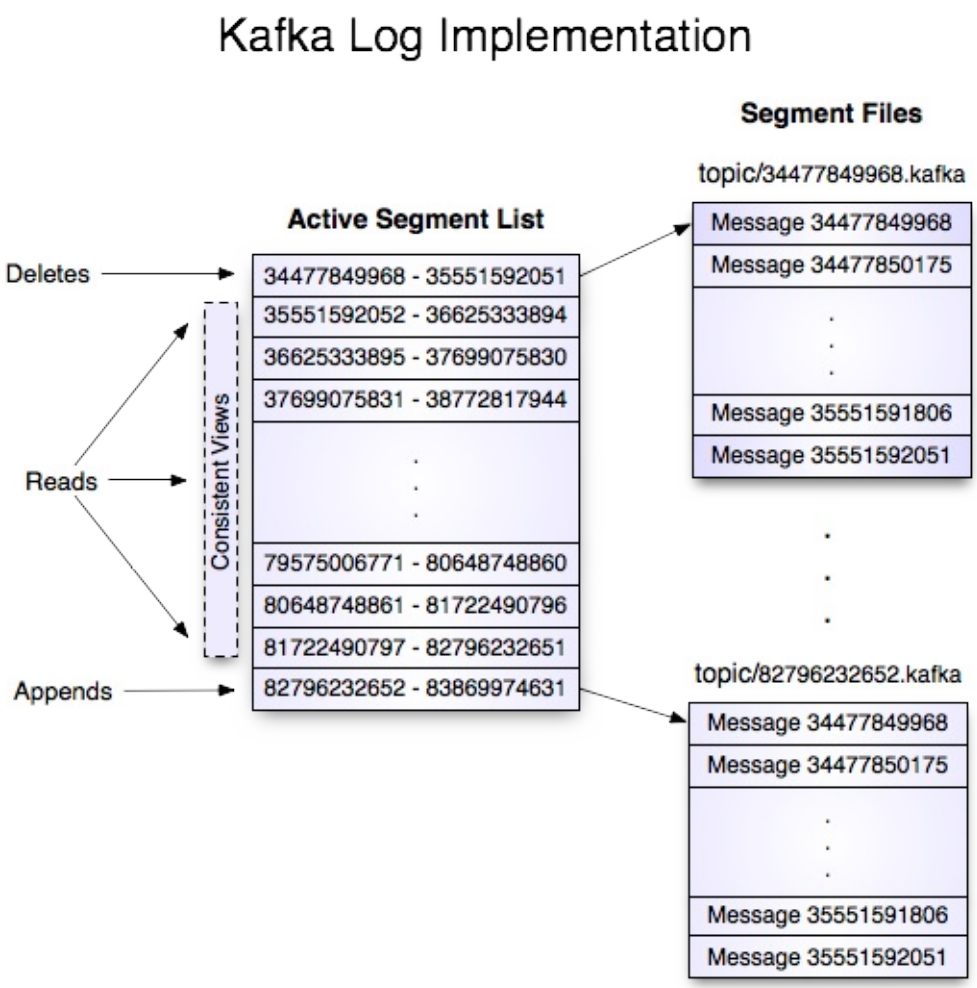
A log for a topic named "my_topic" with two partitions consists of two directories (namely `my_topic_0` and `my_topic_1`) populated with data files containing the messages for that topic. The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer *N* storing the message length which is followed by the *N* message bytes. Each message is uniquely identified by a 64-bit integer *offset* giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition. The on-disk format of each message is given below. Each log file is named with the offset of the first message it contains. So the first file created will be 00000000000.kafka, and each additional file will have an integer name roughly *S_bytes from the previous file where _S* is the max log file size given in the configuration.

The exact binary format for messages is versioned and maintained as a standard interface so message sets can be transferred between producer, broker, and client without recopying or conversion when desirable. This format is as follows:

```
On-disk format of a message

offset         : 8 bytes
message length : 4 bytes (value: 4 + 1 + 1 + 8(if magic value > 0) + 4 + K + 4 + V)
crc            : 4 bytes
magic value    : 1 byte
attributes     : 1 byte
timestamp      : 8 bytes (Only exists when magic value is greater than zero)
key length     : 4 bytes
key            : K bytes
value length   : 4 bytes
value          : V bytes
```

The use of the message offset as the message id is unusual. Our original idea was to use a GUID generated by the producer, and maintain a mapping from GUID to offset on each broker. But since a consumer must maintain an ID for each server, the global uniqueness of the GUID provides no value. Furthermore the complexity of maintaining the mapping from a random id to an offset requires a heavy weight index structure which must be synchronized with disk, essentially requiring a full persistent random-access data structure. Thus to simplify the lookup structure we decided to use a simple per-partition atomic counter which could be coupled with the partition id and node id to uniquely identify a message; this makes the lookup structure simpler, though multiple seeks per consumer request are still likely. However once we settled on a counter, the jump to directly using the offset seemed natural—both after all are monotonically increasing integers unique to a partition. Since the offset is hidden from the consumer API this decision is ultimately an implementation detail and we went with the more efficient approach.



Writes

The log allows serial appends which always go to the last file. This file is rolled over to a fresh file when it reaches a configurable size (say 1GB). The log takes two configuration parameters: *M*, which gives the number of messages to write before forcing the OS to flush the file to disk, and *S*, which gives a number of seconds after which a flush is forced. This gives a durability guarantee of losing at most *M* messages or *S* seconds of data in the event of a system crash.

Reads

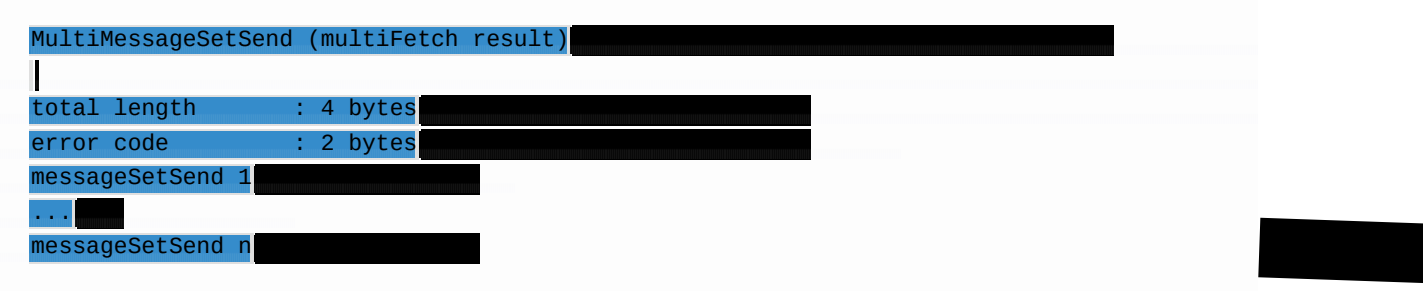
Reads are done by giving the 64-bit logical offset of a message and an *S*-byte max chunk size. This will return an iterator over the messages contained in the *S*-byte buffer. *S* is intended to be larger than any single message, but in the event of an abnormally large message, the read can be retried multiple times, each time doubling the buffer size, until the message is read successfully. A maximum message and buffer size can be specified to

make the server reject messages larger than some size, and to give a bound to the client on the maximum it needs to ever read to get a complete message. It is likely that the read buffer ends with a partial message, this is easily detected by the size delimiting.

The actual process of reading from an offset requires first locating the log segment file in which the data is stored, calculating the file-specific offset from the global offset value, and then reading from that file offset. The search is done as a simple binary search variation against an in-memory range maintained for each file.

The log provides the capability of getting the most recently written message to allow clients to start subscribing as of "right now". This is also useful in the case the consumer fails to consume its data within its SLA-specified number of days. In this case when the client attempts to consume a non-existent offset it is given an OutOfRangeException and can either reset itself or fail as appropriate to the use case.

The following is the format of the results sent to the consumer.



Deletes

Data is deleted one log segment at a time. The log manager allows pluggable delete policies to choose which files are eligible for deletion. The current policy deletes any log with a modification time of more than *N* days ago, though a policy which retained the last *N* GB could also be useful. To avoid locking reads while still allowing deletes that modify the segment list we use a copy-on-write style segment list implementation that provides consistent views to allow a binary search to proceed on an immutable static snapshot view of the log segments while deletes are progressing.

Guarantees

The log provides a configuration parameter *M* which controls the maximum number of messages that are written before forcing a flush to disk. On startup a log recovery process is run that iterates over all messages in the newest log segment and verifies that each message entry is valid. A message entry is valid if the sum of its size and offset are less than the length of the file AND the CRC32 of the message payload matches the CRC stored with the message. In the event corruption is detected the log is truncated to the last valid offset.

Note that two kinds of corruption must be handled: truncation in which an unwritten block is lost due to a crash, and corruption in which a nonsense block is ADDED to the file. The reason for this is that in general the OS makes no guarantee of the write order between the file inode and the actual block data so in addition to losing written data the file can gain nonsense data if the inode is updated with a new size but a crash occurs before the block containing that data is written. The CRC detects this corner case, and prevents it from corrupting the log (though the unwritten messages are, of course, lost).

5.6 Distribution

Consumer Offset Tracking

The high-level consumer tracks the maximum offset it has consumed in each partition and periodically commits its offset vector so that it can resume from those offsets in the event of a restart. Kafka provides the option to store all the offsets for a given consumer group in a designated broker (for that group) called the*offset manager*. i.e., any consumer instance in that consumer group should send its offset commits and fetches to that offset manager (broker). The high-level consumer handles this automatically. If you use the simple consumer you will need to manage offsets manually. This is currently unsupported in the Java simple consumer which can only commit or fetch offsets in ZooKeeper. If you use the Scala simple consumer you can discover the offset manager and explicitly commit or fetch offsets to the offset manager. A consumer can look up its offset manager by issuing a GroupCoordinatorRequest to any Kafka broker and reading the GroupCoordinatorResponse which will contain the offset manager. The consumer can then proceed to commit or fetch offsets from the offsets manager broker. In case the offset manager moves, the consumer will need to rediscover the offset manager. If you wish to manage your offsets manually, you can take a look at these **code samples that explain how to issue OffsetCommitRequest and OffsetFetchRequest**.

When the offset manager receives an OffsetCommitRequest, it appends the request to a special **compacted**Kafka topic named _consumer_offsets_. The offset manager sends a successful offset commit response to the consumer only after all the replicas of the offsets topic receive the offsets. In case the offsets fail to replicate within a configurable timeout, the offset commit will fail and the consumer may retry the commit after backing off. (This is done

automatically by the high-level consumer.) The brokers periodically compact the offsets topic since it only needs to maintain the most recent offset commit per partition. The offset manager also caches the offsets in an in-memory table in order to serve offset fetches quickly.

When the offset manager receives an offset fetch request, it simply returns the last committed offset vector from the offsets cache. In case the offset manager was just started or if it just became the offset manager for a new set of consumer groups (by becoming a leader for a partition of the offsets topic), it may need to load the offsets topic partition into the cache. In this case, the offset fetch will fail with an OffsetsLoadInProgress exception and the consumer may retry the OffsetFetchRequest after backing off. (This is done automatically by the high-level consumer.)

Migrating offsets from ZooKeeper to Kafka

Kafka consumers in earlier releases store their offsets by default in ZooKeeper. It is possible to migrate these consumers to commit offsets into Kafka by following these steps:

1. Set `offsets.storage=kafka` and `dual.commit.enabled=true` in your consumer config.
2. Do a rolling bounce of your consumers and then verify that your consumers are healthy.
3. Set `dual.commit.enabled=false` in your consumer config.
4. Do a rolling bounce of your consumers and then verify that your consumers are healthy.

A roll-back (i.e., migrating from Kafka back to ZooKeeper) can also be performed using the above steps if you set `offsets.storage=zookeeper`.

ZooKeeper Directories

The following gives the ZooKeeper structures and algorithms used for co-ordination between consumers and brokers.

Notation

When an element in a path is denoted [xyz], that means that the value of xyz is not fixed and there is in fact a ZooKeeper znode for each possible value of xyz. For example \/topics\/[topic] would be a directory named \/topics containing a sub-directory for each topic name. Numerical ranges are also given such as [0...5] to indicate the subdirectories 0, 1, 2, 3, 4. An arrow -> is used to indicate the contents of a znode. For example \/hello -> world would indicate a znode \/hello containing the value "world".

Broker Node Registry

```
/brokers/ids/[0...N] --> {"jmx_port":...,"timestamp":...,"endpoints":[...],"host":...,
"version":...,"port":...} (ephemeral node)
```

This is a list of all present broker nodes, each of which provides a unique logical broker id which identifies it to consumers (which must be given as part of its configuration). On startup, a broker node registers itself by creating a znode with the logical broker id under \/brokers\/ids. The purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) results in an error.

Since the broker registers itself in ZooKeeper using ephemeral znodes, this registration is dynamic and will disappear if the broker is shutdown or dies (thus notifying consumers it is no longer available).

Broker Topic Registry

```
/brokers/topics/[topic]/partitions/[0...N]/state --> {"controller_epoch":...,"leader":
...,"version":...,"leader_epoch":...,"isr":[...]} (ephemeral node)
```

Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

Consumers and Consumer Groups

Consumers of topics also register themselves in ZooKeeper, in order to coordinate with each other and balance the consumption of data. Consumers can also store their offsets in ZooKeeper by setting `offsets.storage=zookeeper`. However, this offset storage mechanism will be deprecated in a future release. Therefore, it is recommended to **migrate offsets storage to Kafka**.

Multiple consumers can form a group and jointly consume a single topic. Each consumer in the same group is given a shared group_id. For example if one consumer is your foobar process, which is run across three machines, then you might assign this group of consumers the id "foobar". This group id is provided in the configuration of the consumer, and is your way to tell the consumer which group it belongs to.

The consumers in a group divide up the partitions as fairly as possible, each partition is consumed by exactly one consumer in a consumer group.

Consumer Id Registry

In addition to the group_id which is shared by all consumers in a group, each consumer is given a transient, unique consumer_id (of the form hostname:uuid) for identification purposes. Consumer ids are registered in the following directory.

```
/consumers/[group_id]/ids/[consumer_id] --> {"version":...,"subscription":{...:...},"p
attern":...,"timestamp":...} (ephemeral node)
```

Each of the consumers in the group registers under its group and creates a znode with its consumer_id. The value of the znode contains a map of <topic, #streams>. This id is simply used to identify each of the consumers which is currently active within a group. This is an ephemeral node so it will disappear if the consumer process dies.

Consumer Offsets

Consumers track the maximum offset they have consumed in each partition. This value is stored in a ZooKeeper directory if `offsets.storage=zookeeper` .

```
/consumers/[group_id]/offsets/[topic]/[partition_id] --> offset_counter_value ((persis
tent node)
```

Partition Owner registry

Each broker partition is consumed by a single consumer within a given consumer group. The consumer must establish its ownership of a given partition before any consumption can begin. To establish its ownership, a consumer writes its own id in an ephemeral node under the particular broker partition it is claiming.

```
/consumers/[group_id]/owners/[topic]/[partition_id] --> consumer_node_id (ephemeral no
de)
```

Broker node registration

The broker nodes are basically independent, so they only publish information about what they have. When a broker joins, it registers itself under the broker node registry directory and writes information about its host name and port. The broker also register the list of existing topics and their logical partitions in the broker topic registry. New topics are registered dynamically when they are created on the broker.

Consumer registration algorithm

When a consumer starts, it does the following:

1. Register itself in the consumer id registry under its group.
2. Register a watch on changes (new consumers joining or any existing consumers leaving) under the consumer id registry. (Each change triggers rebalancing among all consumers within the group to which the changed consumer belongs.)
3. Register a watch on changes (new brokers joining or any existing brokers leaving) under the broker id registry. (Each change triggers rebalancing among all consumers in all consumer groups.)
4. If the consumer creates a message stream using a topic filter, it also registers a watch on changes (new topics being added) under the broker topic registry. (Each change will trigger re-evaluation of the available topics to determine which topics are allowed by the topic filter. A new allowed topic will trigger rebalancing among all consumers within the consumer group.)
5. Force itself to rebalance within in its consumer group.

Consumer rebalancing algorithm

The consumer rebalancing algorithms allows all the consumers in a group to come into consensus on which consumer is consuming which partitions. Consumer rebalancing is triggered on each addition or removal of both broker nodes and other consumers within the same group. For a given topic and a given consumer group, broker partitions are divided evenly among consumers within the group. A partition is always consumed by a single consumer. This design simplifies the implementation. Had we allowed a partition to be concurrently consumed by multiple consumers, there would be contention on the partition and some kind of locking would be required. If there are more consumers than partitions, some consumers won't get any data at all. During rebalancing, we try to assign partitions to consumers in such a way that reduces the number of broker nodes each consumer has to connect to.

Each consumer does the following during rebalancing:

```
1. For each topic T that Ci subscribes to
2.   let PT be all partitions producing topic T
3.   let CG be all consumers in the same group as Ci that consume topic T
4.   sort PT (so partitions on the same broker are clustered together)
5.   sort CG
6.   let i be the index position of Ci in CG and let N = size(PT)/size(CG)
7.   assign partitions from i*N to (i+1)*N - 1 to consumer Ci
8.   remove current entries owned by Ci from the partition owner registry
9.   add newly assigned partitions to the partition owner registry
       (we may need to re-try this until the original partition owner releases its ow
nership)
```

When rebalancing is triggered at one consumer, rebalancing should be triggered in other consumers within the same group about the same time.

6. Operations

Here is some information on actually running Kafka as a production system based on usage and experience at LinkedIn. Please send us any additional tips you know of.

6.1 Basic Kafka Operations

This section will review the most common operations you will perform on your Kafka cluster. All of the tools reviewed in this section are available under the `bin/` directory of the Kafka distribution and each tool will print details on all possible commandline options if it is run with no arguments.

Adding and removing topics

You have the option of either adding topics manually or having them be created automatically when data is first published to a non-existent topic. If topics are auto-created then you may want to tune the default **topic configurations** used for auto-created topics.

Topics are added and modified using the topic tool:

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --create --topic my_topic_name
      --partitions 20 --replication-factor 3 --config x=y
```

The replication factor controls how many servers will replicate each message that is written. If you have a replication factor of 3 then up to 2 servers can fail before you will lose access to your data. We recommend you use a replication factor of 2 or 3 so that you can transparently bounce machines without interrupting data consumption.

The partition count controls how many logs the topic will be sharded into. There are several impacts of the partition count. First each partition must fit entirely on a single server. So if you have 20 partitions the full data set (and read and write load) will be handled by no more than 20 servers (no counting replicas). Finally the partition count impacts the maximum parallelism of your consumers. This is discussed in greater detail in the**concepts section**.

Each sharded partition log is placed into its own folder under the Kafka log directory. The name of such folders consists of the topic name, appended by a dash (-) and the partition id. Since a typical folder name can not be over 255 characters long, there will be a limitation on the length of topic names. We assume the number of partitions will not ever be above 100,000. Therefore, topic names cannot be longer than 249 characters. This leaves just enough room in the folder name for a dash and a potentially 5 digit long partition id.

The configurations added on the command line override the default settings the server has for things like the length of time data should be retained. The complete set of per-topic configurations is documented **here**.

Modifying topics

You can change the configuration or partitioning of a topic using the same topic tool.

To add partitions you can do

```
 > bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name
       --partitions 40
```

Be aware that one use case for partitions is to semantically partition data, and adding partitions doesn't change the partitioning of existing data so this may disturb consumers if they rely on that partition. That is if data is partitioned by `hash(key) % number_of_partitions` then this partitioning will potentially be shuffled by adding partitions but Kafka will not attempt to automatically redistribute data in any way.

To add configs:

```
 > bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name -
-config x=y
```

To remove a config:

```
 > bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name -
-delete-config x
```

And finally deleting a topic:

```
 > bin/kafka-topics.sh --zookeeper zk_host:port/chroot --delete --topic my_topic_name
```

Topic deletion option is disabled by default. To enable it set the server config

```
delete.topic.enable=true
```

Kafka does not currently support reducing the number of partitions for a topic.

Instructions for changing the replication factor of a topic can be found **here**.

Graceful shutdown

The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine. This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes. For the latter cases Kafka supports a more graceful mechanism for stopping a server than just killing it. When a server is stopped gracefully it has two optimizations it will take advantage of:

1. It will sync all its logs to disk to avoid needing to do any log recovery when it restarts (i.e. validating the checksum for all messages in the tail of the log). Log recovery takes time so this speeds up intentional restarts.
2. It will migrate any partitions the server is the leader for to other replicas prior to shutting down. This will make the leadership transfer faster and minimize the time each partition is unavailable to a few milliseconds.

Syncing the logs will happen automatically whenever the server is stopped other than by a hard kill, but the controlled leadership migration requires using a special setting:

```
    controlled.shutdown.enable=true
```

Note that controlled shutdown will only succeed if *all* the partitions hosted on the broker have replicas (i.e. the replication factor is greater than 1 *and* at least one of these replicas is alive). This is generally what you want since shutting down the last replica would make that topic partition unavailable.

Balancing leadership

Whenever a broker stops or crashes leadership for that broker's partitions transfers to other replicas. This means that by default when the broker is restarted it will only be a follower for all its partitions, meaning it will not be used for client reads and writes.

To avoid this imbalance, Kafka has a notion of preferred replicas. If the list of replicas for a partition is 1,5,9 then node 1 is preferred as the leader to either node 5 or 9 because it is earlier in the replica list. You can have the Kafka cluster try to restore leadership to the restored replicas by running the command:

```
> bin/kafka-preferred-replica-election.sh --zookeeper zk_host:port/chroot
```

Since running this command can be tedious you can also configure Kafka to do this automatically by setting the following configuration:

```
    auto.leader.rebalance.enable=true
```

Balancing Replicas Across Racks

The rack awareness feature spreads replicas of the same partition across different racks. This extends the guarantees Kafka provides for broker-failure to cover rack-failure, limiting the risk of data loss should all the brokers on a rack fail at once. The feature can also be applied to other broker groupings such as availability zones in EC2.

You can specify that a broker belongs to a particular rack by adding a property to the broker config:

```
broker.rack=my-rack-id
```

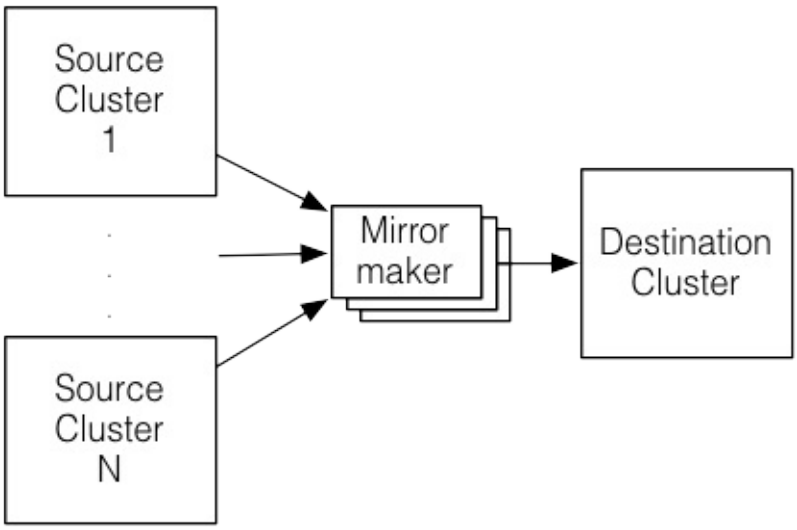
When a topic is **created**, **modified** or replicas are **redistributed**, the rack constraint will be honoured, ensuring replicas span as many racks as they can (a partition will span min(#racks, replication-factor) different racks).

The algorithm used to assign replicas to brokers ensures that the number of leaders per broker will be constant, regardless of how brokers are distributed across racks. This ensures balanced throughput.

However if racks are assigned different numbers of brokers, the assignment of replicas will not be even. Racks with fewer brokers will get more replicas, meaning they will use more storage and put more resources into replication. Hence it is sensible to configure an equal number of brokers per rack.

Mirroring data between clusters

We refer to the process of replicating data *between* Kafka clusters "mirroring" to avoid confusion with the replication that happens amongst the nodes in a single cluster. Kafka comes with a tool for mirroring data between Kafka clusters. The tool reads from a source cluster and writes to a destination cluster, like this:



A common use case for this kind of mirroring is to provide a replica in another datacenter. This scenario will be discussed in more detail in the next section.

You can run many such mirroring processes to increase throughput and for fault-tolerance (if one process dies, the others will take overs the additional load).

Data will be read from topics in the source cluster and written to a topic with the same name in the destination cluster. In fact the mirror maker is little more than a Kafka consumer and producer hooked together.

The source and destination clusters are completely independent entities: they can have different numbers of partitions and the offsets will not be the same. For this reason the mirror cluster is not really intended as a fault-tolerance mechanism (as the consumer position will be different); for that we recommend using normal in-cluster replication. The mirror maker process will, however, retain and use the message key for partitioning so order is preserved on a per-key basis.

Here is an example showing how to mirror a single topic (named *my-topic*) from two input clusters:

```
 > bin/kafka-mirror-maker.sh [redacted]
       --consumer.config consumer-1.properties --consumer.config consumer-2.properties [redacted]
       --producer.config producer.properties --whitelist my-topic [redacted]
```

Note that we specify the list of topics with the `--whitelist` option. This option allows any regular expression using **Java-style regular expressions**. So you could mirror two topics named *A* and *B* using `--whitelist 'A|B'`. Or you could mirror *all* topics using `--whitelist '*'`. Make sure to quote any regular expression to ensure the shell doesn't try to expand it as a file path. For convenience we allow the use of ',' instead of '|' to specify a list of topics.

Sometimes it is easier to say what it is that you *don't* want. Instead of using `--whitelist` to say what you want to mirror you can use `--blacklist` to say what to exclude. This also takes a regular expression argument. However, `--blacklist` is not supported when using `--new.consumer`.

Combining mirroring with the configuration `auto.create.topics.enable=true` makes it possible to have a replica cluster that will automatically create and replicate all data in a source cluster even as new topics are added.

Checking consumer position

Sometimes it's useful to see the position of your consumers. We have a tool that will show the position of all consumers in a consumer group as well as how far behind the end of the log they are. To run this tool on a consumer group named *my-group* consuming a topic named *my-topic* would look like this:

```
 > bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --zookeeper localhost:2181
 --group test
Group           Topic                          Pid Offset          logSize         Lag
             Owner
my-group        my-topic                       0   0               0               0
             test_jkreps-mn-1394154511599-60744496-0
my-group        my-topic                       1   0               0               0
             test_jkreps-mn-1394154521217-1a0be913-0
```

Note, however, after 0.9.0, the kafka.tools.ConsumerOffsetChecker tool is deprecated and you should use the kafka.admin.ConsumerGroupCommand (or the bin\/kafka-consumer-groups.sh script) to manage consumer groups, including consumers created with the **new consumer API**.

Managing Consumer Groups

With the ConsumerGroupCommand tool, we can list, delete, or describe consumer groups. For example, to list all consumer groups across all topics:

```
 > bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --list

test-consumer-group
```

To view offsets as in the previous example with the ConsumerOffsetChecker, we "describe" the consumer group like this:

```
 > bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --describe --group test-con
sumer-group

GROUP                          TOPIC                          PARTITION  CURRENT-OFFSE
T  LOG-END-OFFSET  LAG             OWNER
test-consumer-group            test-foo                       0          1
   3               2               test-consumer-group_postamac.local-1456198719410-29
ccd54f-0
```

When you're using the **new consumer API** where the broker handles coordination of partition handling and rebalance, you can manage the groups with the "--new-consumer" flags:

```
 > bin/kafka-consumer-groups.sh --new-consumer --bootstrap-server broker1:9092 --list
```

Expanding your cluster

Adding servers to a Kafka cluster is easy, just assign them a unique broker id and start up Kafka on your new servers. However these new servers will not automatically be assigned any data partitions, so unless partitions are moved to them they won't be doing any work until new topics are created. So usually when you add machines to your cluster you will want to migrate some existing data to these machines.

The process of migrating data is manually initiated but fully automated. Under the covers what happens is that Kafka will add the new server as a follower of the partition it is migrating and allow it to fully replicate the existing data in that partition. When the new server has fully replicated the contents of this partition and joined the in-sync replica one of the existing replicas will delete their partition's data.

The partition reassignment tool can be used to move partitions across brokers. An ideal partition distribution would ensure even data load and partition sizes across all brokers. The partition reassignment tool does not have the capability to automatically study the data distribution in a Kafka cluster and move partitions around to attain an even load distribution. As such, the admin has to figure out which topics or partitions should be moved around.

The partition reassignment tool can run in 3 mutually exclusive modes -

- --generate: In this mode, given a list of topics and a list of brokers, the tool generates a candidate reassignment to move all partitions of the specified topics to the new brokers. This option merely provides a convenient way to generate a partition reassignment plan given a list of topics and target brokers.
- --execute: In this mode, the tool kicks off the reassignment of partitions based on the user provided reassignment plan. (using the --reassignment-json-file option). This can

either be a custom reassignment plan hand crafted by the admin or provided by using the --generate option

- --verify: In this mode, the tool verifies the status of the reassignment for all partitions listed during the last --execute. The status can be either of successfully completed, failed or in progress

Automatically migrating data to new machines

The partition reassignment tool can be used to move some topics off of the current set of brokers to the newly added brokers. This is typically useful while expanding an existing cluster since it is easier to move entire topics to the new set of brokers, than moving one partition at a time. When used to do this, the user should provide a list of topics that should be moved to the new set of brokers and a target list of new brokers. The tool then evenly distributes all partitions for the given list of topics across the new set of brokers. During this move, the replication factor of the topic is kept constant. Effectively the replicas for all partitions for the input list of topics are moved from the old set of brokers to the newly added brokers.

For instance, the following example will move all partitions for topics foo1,foo2 to the new set of brokers 5,6. At the end of this move, all partitions for topics foo1 and foo2 will *only* exist on brokers 5,6.

Since the tool accepts the input list of topics as a json file, you first need to identify the topics you want to move and create the json file as follows:

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"},
            {"topic": "foo2"}],
 "version":1
}
```

Once the json file is ready, use the partition reassignment tool to generate a candidate assignment:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-fi
le topics-to-move.json --broker-list "5,6" --generate
Current partition replica assignment

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

Proposed partition reassignment configuration

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
               {"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":2,"replicas":[5,6]},
               {"topic":"foo2","partition":0,"replicas":[5,6]},
               {"topic":"foo1","partition":1,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

The tool generates a candidate assignment that will move all partitions from topics foo1,foo2 to brokers 5,6. Note, however, that at this point, the partition movement has not started, it merely tells you the current assignment and the proposed new assignment. The current assignment should be saved in case you want to rollback to it. The new assignment should be saved in a json file (e.g. expand-cluster-reassignment.json) to be input to the tool with the --execute option as follows:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
 expand-cluster-reassignment.json --execute
Current partition replica assignment

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

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

Finally, the --verify option can be used with the tool to check the status of the partition reassignment. Note that the same expand-cluster-reassignment.json (used with the --execute option) should be used with the --verify option:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
 expand-cluster-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo1,1] is in progress
Reassignment of partition [foo1,2] is in progress
Reassignment of partition [foo2,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
Reassignment of partition [foo2,2] completed successfully
```

Custom partition assignment and migration

The partition reassignment tool can also be used to selectively move replicas of a partition to a specific set of brokers. When used in this manner, it is assumed that the user knows the reassignment plan and does not require the tool to generate a candidate reassignment, effectively skipping the --generate step and moving straight to the --execute step

For instance, the following example moves partition 0 of topic foo1 to brokers 5,6 and partition 1 of topic foo2 to brokers 2,3:

The first step is to hand craft the custom reassignment plan in a json file:

```
> cat custom-reassignment.json
{"version":1,"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},{"topic":"f
oo2","partition":1,"replicas":[2,3]}]}
```

Then, use the json file with the --execute option to start the reassignment process:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
 custom-reassignment.json --execute
Current partition replica assignment

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

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
}
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same expand-cluster-reassignment.json (used with the --execute option) should be used with the --verify option:

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file c
ustom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
```

Decommissioning brokers

The partition reassignment tool does not have the ability to automatically generate a reassignment plan for decommissioning brokers yet. As such, the admin has to come up with a reassignment plan to move the replica for all partitions hosted on the broker to be decommissioned, to the rest of the brokers. This can be relatively tedious as the reassignment needs to ensure that all the replicas are not moved from the decommissioned broker to only one other broker. To make this process effortless, we plan to add tooling support for decommissioning brokers in the future.

Increasing replication factor

Increasing the replication factor of an existing partition is easy. Just specify the extra replicas in the custom reassignment json file and use it with the --execute option to increase the replication factor of the specified partitions.

For instance, the following example increases the replication factor of partition 0 of topic foo from 1 to 3. Before increasing the replication factor, the partition's only replica existed on broker 5. As part of increasing the replication factor, we will add more replicas on brokers 6 and 7.

The first step is to hand craft the custom reassignment plan in a json file:

```
> cat increase-replication-factor.json
{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

Then, use the json file with the --execute option to start the reassignment process:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
 increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same increase-replication-factor.json (used with the --execute option) should be used with the --verify option:

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file i
ncrease-replication-factor.json --verify
Status of partition reassignment:
Reassignment of partition [foo,0] completed successfully
```

You can also verify the increase in replication factor with the kafka-topics tool:

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --topic foo --describe
Topic:foo    PartitionCount:1    ReplicationFactor:3    Configs:
    Topic: foo    Partition: 0    Leader: 5    Replicas: 5,6,7    Isr: 5,6,7
```

Setting quotas

It is possible to set default quotas that apply to all client-ids by setting these configs on the brokers. By default, each client-id receives an unlimited quota. The following sets the default quota per producer and consumer client-id to 10MB\/sec.

```
  quota.producer.default=10485760
  quota.consumer.default=10485760
```

It is also possible to set custom quotas for each client.

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte
_rate=1024,consumer_byte_rate=2048' --entity-name clientA --entity-type clients
Updated config for clientId: "clientA".
```

Here's how to describe the quota for a given client.

```
> ./kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-name clientA --en
tity-type clients
Configs for clients:clientA are producer_byte_rate=1024,consumer_byte_rate=2048
```

6.2 Datacenters

Some deployments will need to manage a data pipeline that spans multiple datacenters. Our recommended approach to this is to deploy a local Kafka cluster in each datacenter with application instances in each datacenter interacting only with their local cluster and mirroring between clusters (see the documentation on the **mirror maker tool** for how to do this).

This deployment pattern allows datacenters to act as independent entities and allows us to manage and tune inter-datacenter replication centrally. This allows each facility to stand alone and operate even if the inter-datacenter links are unavailable: when this occurs the mirroring falls behind until the link is restored at which time it catches up.

For applications that need a global view of all data you can use mirroring to provide clusters which have aggregate data mirrored from the local clusters in *all* datacenters. These aggregate clusters are used for reads by applications that require the full data set.

This is not the only possible deployment pattern. It is possible to read from or write to a remote Kafka cluster over the WAN, though obviously this will add whatever latency is required to get the cluster.

Kafka naturally batches data in both the producer and consumer so it can achieve high-throughput even over a high-latency connection. To allow this though it may be necessary to increase the TCP socket buffer sizes for the producer, consumer, and broker using the

`socket.send.buffer.bytes` and `socket.receive.buffer.bytes` configurations. The appropriate way to set this is documented **here**.

It is generally *not* advisable to run a *single* Kafka cluster that spans multiple datacenters over a high-latency link. This will incur very high replication latency both for Kafka writes and ZooKeeper writes, and neither Kafka nor ZooKeeper will remain available in all locations if the network between locations is unavailable.

6.3 Kafka Configuration

Important Client Configurations

The most important producer configurations control

- compression
- sync vs async production
- batch size (for async producers)

The most important consumer configuration is the fetch size.

All configurations are documented in the **configuration** section.

A Production Server Config

Here is our production server configuration:

```
# Replication configurations
num.replica.fetchers=4
replica.fetch.max.bytes=1048576
replica.fetch.wait.max.ms=500
replica.high.watermark.checkpoint.interval.ms=5000
replica.socket.timeout.ms=30000
replica.socket.receive.buffer.bytes=65536
replica.lag.time.max.ms=10000

controller.socket.timeout.ms=30000
controller.message.queue.size=10

# Log configuration
num.partitions=8
message.max.bytes=1000000
auto.create.topics.enable=true
log.index.interval.bytes=4096
log.index.size.max.bytes=10485760
log.retention.hours=168
log.flush.interval.ms=10000
log.flush.interval.messages=20000
log.flush.scheduler.interval.ms=2000
log.roll.hours=168
log.retention.check.interval.ms=300000
log.segment.bytes=1073741824

# ZK configuration
zookeeper.connection.timeout.ms=6000
zookeeper.sync.time.ms=2000

# Socket server configuration
num.io.threads=8
num.network.threads=8
socket.request.max.bytes=104857600
socket.receive.buffer.bytes=1048576
socket.send.buffer.bytes=1048576
queued.max.requests=16
fetch.purgatory.purge.interval.requests=100
producer.purgatory.purge.interval.requests=100
```

Our client configuration varies a fair amount between different use cases.

Java Version

From a security perspective, we recommend you use the latest released version of JDK 1.8 as older freely available versions have disclosed security vulnerabilities. LinkedIn is currently running JDK 1.8 u5 (looking to upgrade to a newer version) with the G1 collector. If you decide to use the G1 collector (the current default) and you are still on JDK 1.7, make sure you are on u51 or newer. LinkedIn tried out u21 in testing, but they had a number of problems with the GC implementation in that version. LinkedIn's tuning looks like this:

```
-Xmx6g -Xms6g -XX:MetaspaceSize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

For reference, here are the stats on one of LinkedIn's busiest clusters (at peak):

- 60 brokers
- 50k partitions (replication factor 2)
- 800k messages\/sec in
- 300 MB\/sec inbound, 1 GB\/sec+ outbound

The tuning looks fairly aggressive, but all of the brokers in that cluster have a 90% GC pause time of about 21ms, and they're doing less than 1 young GC per second.

6.4 Hardware and OS

We are using dual quad-core Intel Xeon machines with 24GB of memory.

You need sufficient memory to buffer active readers and writers. You can do a back-of-the-envelope estimate of memory needs by assuming you want to be able to buffer for 30 seconds and compute your memory need as write_throughput*30.

The disk throughput is important. We have 8x7200 rpm SATA drives. In general disk throughput is the performance bottleneck, and more disks is better. Depending on how you configure flush behavior you may or may not benefit from more expensive disks (if you force flush often then higher RPM SAS drives may be better).

OS

Kafka should run well on any unix system and has been tested on Linux and Solaris.

We have seen a few issues running on Windows and Windows is not currently a well supported platform though we would be happy to change that.

It is unlikely to require much OS-level tuning, but there are two potentially important OS-level configurations:

- File descriptor limits: Kafka uses file descriptors for log segments and open connections. If a broker hosts many partitions, consider that the broker needs at least (number_of_partitions)*(partition_size\/segment_size) to track all log segments in addition to the number of connections the broker makes. We recommend at least 100000 allowed file descriptors for the broker processes as a starting point.
- Max socket buffer size: can be increased to enable high-performance data transfer between data centers as **described here**.

Disks and Filesystem

We recommend using multiple drives to get good throughput and not sharing the same drives used for Kafka data with application logs or other OS filesystem activity to ensure good latency. You can either RAID these drives together into a single volume or format and mount each drive as its own directory. Since Kafka has replication the redundancy provided by RAID can also be provided at the application level. This choice has several tradeoffs.

If you configure multiple data directories partitions will be assigned round-robin to data directories. Each partition will be entirely in one of the data directories. If data is not well balanced among partitions this can lead to load imbalance between disks.

RAID can potentially do better at balancing load between disks (although it doesn't always seem to) because it balances load at a lower level. The primary downside of RAID is that it is usually a big performance hit for write throughput and reduces the available disk space.

Another potential benefit of RAID is the ability to tolerate disk failures. However our experience has been that rebuilding the RAID array is so I\/O intensive that it effectively disables the server, so this does not provide much real availability improvement.

Application vs. OS Flush Management

Kafka always immediately writes all data to the filesystem and supports the ability to configure the flush policy that controls when data is forced out of the OS cache and onto disk using the flush. This flush policy can be controlled to force data to disk after a period of time or after a certain number of messages has been written. There are several choices in this configuration.

Kafka must eventually call fsync to know that data was flushed. When recovering from a crash for any log segment not known to be fsync'd Kafka will check the integrity of each message by checking its CRC and also rebuild the accompanying offset index file as part of the recovery process executed on startup.

Note that durability in Kafka does not require syncing data to disk, as a failed node will always recover from its replicas.

We recommend using the default flush settings which disable application fsync entirely. This means relying on the background flush done by the OS and Kafka's own background flush. This provides the best of all worlds for most uses: no knobs to tune, great throughput and latency, and full recovery guarantees. We generally feel that the guarantees provided by replication are stronger than sync to local disk, however the paranoid still may prefer having both and application level fsync policies are still supported.

The drawback of using application level flush settings is that it is less efficient in it's disk usage pattern (it gives the OS less leeway to re-order writes) and it can introduce latency as fsync in most Linux filesystems blocks writes to the file whereas the background flushing does much more granular page-level locking.

In general you don't need to do any low-level tuning of the filesystem, but in the next few sections we will go over some of this in case it is useful.

Understanding Linux OS Flush Behavior

In Linux, data written to the filesystem is maintained in **pagecache** until it must be written out to disk (due to an application-level fsync or the OS's own flush policy). The flushing of data is done by a set of background threads called pdflush (or in post 2.6.32 kernels "flusher threads").

Pdflush has a configurable policy that controls how much dirty data can be maintained in cache and for how long before it must be written back to disk. This policy is described **here**. When Pdflush cannot keep up with the rate of data being written it will eventually cause the writing process to block incurring latency in the writes to slow down the accumulation of data.

You can see the current state of OS memory usage by doing

```
> cat /proc/meminfo
```

The meaning of these values are described in the link above.

Using pagecache has several advantages over an in-process cache for storing data that will be written out to disk:

- The I\/O scheduler will batch together consecutive small writes into bigger physical writes which improves throughput.
- The I\/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves throughput.
- It automatically uses all the free memory on the machine

Filesystem Selection

Kafka uses regular files on disk, and as such it has no hard dependency on a specific filesystem. The two filesystems which have the most usage, however, are EXT4 and XFS. Historically, EXT4 has had more usage, but recent improvements to the XFS filesystem have shown it to have better performance characteristics for Kafka's workload with no compromise in stability.

Comparison testing was performed on a cluster with significant message loads, using a variety of filesystem creation and mount options. The primary metric in Kafka that was monitored was the "Request Local Time", indicating the amount of time append operations were taking. XFS resulted in much better local times (160ms vs. 250ms+ for the best EXT4 configuration), as well as lower average wait times. The XFS performance also showed less variability in disk performance.

General Filesystem Notes

For any filesystem used for data directories, on Linux systems, the following options are recommended to be used at mount time:

- noatime: This option disables updating of a file's atime (last access time) attribute when the file is read. This can eliminate a significant number of filesystem writes, especially in the case of bootstrapping consumers. Kafka does not rely on the atime attributes at all, so it is safe to disable this.

XFS Notes

The XFS filesystem has a significant amount of auto-tuning in place, so it does not require any change in the default settings, either at filesystem creation time or at mount. The only tuning parameters worth considering are:

- largeio: This affects the preferred I\/O size reported by the stat call. While this can allow for higher performance on larger disk writes, in practice it had minimal or no effect on performance.
- nobarrier: For underlying devices that have battery-backed cache, this option can provide a little more performance by disabling periodic write flushes. However, if the underlying device is well-behaved, it will report to the filesystem that it does not require flushes, and this option will have no effect.

EXT4 Notes

EXT4 is a serviceable choice of filesystem for the Kafka data directories, however getting the most performance out of it will require adjusting several mount options. In addition, these options are generally unsafe in a failure scenario, and will result in much more data loss and corruption. For a single broker failure, this is not much of a concern as the disk can be wiped and the replicas rebuilt from the cluster. In a multiple-failure scenario, such as a power outage, this can mean underlying filesystem (and therefore data) corruption that is not easily recoverable. The following options can be adjusted:

- data=writeback: Ext4 defaults to data=ordered which puts a strong order on some writes. Kafka does not require this ordering as it does very paranoid data recovery on all unflushed log. This setting removes the ordering constraint and seems to significantly

reduce latency.

- Disabling journaling: Journaling is a tradeoff: it makes reboots faster after server crashes but it introduces a great deal of additional locking which adds variance to write performance. Those who don't care about reboot time and want to reduce a major source of write latency spikes can turn off journaling entirely.
- commit=num_secs: This tunes the frequency with which ext4 commits to its metadata journal. Setting this to a lower value reduces the loss of unflushed data during a crash. Setting this to a higher value will improve throughput.
- nobh: This setting controls additional ordering guarantees when using data=writeback mode. This should be safe with Kafka as we do not depend on write ordering and improves throughput and latency.
- delalloc: Delayed allocation means that the filesystem avoid allocating any blocks until the physical write occurs. This allows ext4 to allocate a large extent instead of smaller pages and helps ensure the data is written sequentially. This feature is great for throughput. It does seem to involve some locking in the filesystem which adds a bit of latency variance.

6.6 Monitoring

Kafka uses Yammer Metrics for metrics reporting in both the server and the client. This can be configured to report stats using pluggable stats reporters to hook up to your monitoring system.

The easiest way to see the available metrics is to fire up jconsole and point it at a running kafka client or server; this will allow browsing all metrics with JMX.

We do graphing and alerting on the following metrics:

DescriptionMbean nameNormal value	
Message in rate	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
Byte in rate	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
Request rate	kafka.network:type=RequestMetrics,name=RequestsPerSec,reques
Byte out rate	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
Log flush rate and time	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs
# of under replicated partitions (\	ISR\
Is controller active	

Is controller active on broker	kafka.controller:type=KafkaController,name=ActiveControllerCount
Leader election rate	kafka.controller:type=ControllerStats,name=LeaderElectionRateAnd
Unclean leader election rate	kafka.controller:type=ControllerStats,name=UncleanLeaderElection
Partition counts	kafka.server:type=ReplicaManager,name=PartitionCount
Leader replica counts	kafka.server:type=ReplicaManager,name=LeaderCount
ISR shrink rate	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec
ISR expansion rate	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec
Max lag in messages btw follower and leader replicas	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=
Lag in messages per follower replica	kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId= ([-.\w]+),partition=([0-9]+)
Requests waiting in the producer purgatory	kafka.server:type=ProducerRequestPurgatory,name=PurgatorySize
Requests waiting in the fetch purgatory	kafka.server:type=FetchRequestPurgatory,name=PurgatorySize

Request total time	kafka.network:type=RequestMetrics,name=TotalTimeMs,request={F
Time the request waiting in the request queue	kafka.network:type=RequestMetrics,name=QueueTimeMs,request=
Time the request being processed at the leader	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={
Time the request waits for the follower	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request
Time to send the response	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs, {Produce\
Number of messages the consumer lags behind the producer by	kafka.consumer:type=ConsumerFetcherManager,name=MaxLag,cli
The average fraction of time the network processors are idle	kafka.network:type=SocketServer,name=NetworkProcessorAvgIdle
The average fraction of time the request handler threads are idle	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandl
Quota metrics per client-id	kafka.server:type={Produce\

New producer monitoring

The following metrics are available on new producer instances.

Metric\/Attribute nameDescriptionMbean name		
waiting-threads	The number of user threads blocked waiting for buffer memory to enqueue their records.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-total-bytes	The maximum amount of buffer memory the client can use (whether or not it is currently used).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-available-bytes	The total amount of buffer memory that is not being used (either unallocated or in the free list).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
bufferpool-wait-time	The fraction of time an appender waits for space allocation.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
batch-size-avg	The average number of bytes sent per partition per-request.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
batch-size-max	The max number of bytes sent per partition per-request.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
compression-rate-avg	The average compression rate of record batches.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-queue-time-avg	The average time in ms record batches spent in the record accumulator.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-queue-time-max	The maximum time in ms record batches spent in the record accumulator.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
request-latency-avg	The average request latency in ms.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
	The maximum request	kafka.producer:type=producer-

record-send-rate	The average number of records sent per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
records-per-request-avg	The average number of records per request.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-retry-rate	The average per-second number of retried record sends.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-error-rate	The average per-second number of record sends that resulted in errors.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-size-max	The maximum record size.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-size-avg	The average record size.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
requests-in-flight	The current number of in-flight requests awaiting a response.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
metadata-age	The age in seconds of the current producer metadata being used.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
connection-close-rate	Connections closed per second in the window.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
connection-creation-rate	New connections established per second in the window.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
network-io-rate	The average number of network operations (reads or writes) on all connections per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second to all servers.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
request-rate	The average number of requests sent per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
request-size-avg	The average size of all requests in the window.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
request-size-max	The maximum size of any request sent in the window.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)

request-size-max	any request sent in the window.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
incoming-byte-rate	Bytes\/second read off all sockets.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
response-rate	Responses received sent per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
select-rate	Number of times the I\/O layer checked for new I\/O to perform per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
io-wait-time-ns-avg	The average length of time the I\/O thread spent waiting for a socket ready for reads or writes in nanoseconds.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
io-wait-ratio	The fraction of time the I\/O thread spent waiting.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
io-time-ns-avg	The average length of time for I\/O per select call in nanoseconds.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
io-ratio	The fraction of time the I\/O thread spent doing I\/O.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
connection-count	The current number of active connections.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-rate	The average number of requests sent per second for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-avg	The average size of all requests in the window for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-max	The maximum size of any request sent in the window for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
incoming-byte-rate	The average number of responses received per second for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
	The average request	kafka.producer:type=producer-

request-latency-max	The maximum request latency in ms for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
response-rate	Responses received sent per second for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
record-send-rate	The average number of records sent per second for a topic.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+)
byte-rate	The average number of bytes sent per second for a topic.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+)
compression-rate	The average compression rate of record batches for a topic.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+)
record-retry-rate	The average per-second number of retried record sends for a topic.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+)
record-error-rate	The average per-second number of record sends that resulted in errors for a topic.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+)
produce-throttle-time-max	The maximum time in ms a request was throttled by a broker.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+)
produce-throttle-time-avg	The average time in ms a request was throttled by a broker.	kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+)

We recommend monitoring GC time and other stats and various server stats such as CPU utilization, I\/O service time, etc. On the client side, we recommend monitoring the message\/byte rate (global and per topic), request rate\/size\/time, and on the consumer side, max lag in messages among all partitions and min fetch request rate. For a consumer to keep up, max lag needs to be less than a threshold and min fetch rate needs to be larger than 0.

Audit

The final alerting we do is on the correctness of the data delivery. We audit that every message that is sent is consumed by all consumers and measure the lag for this to occur. For important topics we alert if a certain completeness is not achieved in a certain time period. The details of this are discussed in KAFKA-260.

6.7 ZooKeeper

Stable version

The current stable branch is 3.4 and the latest release of that branch is 3.4.6, which is the one ZkClient 0.7 uses. ZkClient is the client layer Kafka uses to interact with ZooKeeper.

Operationalizing ZooKeeper

Operationally, we do the following for a healthy ZooKeeper installation:

- Redundancy in the physical\/hardware\/network layout: try not to put them all in the same rack, decent (but don't go nuts) hardware, try to keep redundant power and network paths, etc. A typical ZooKeeper ensemble has 5 or 7 servers, which tolerates 2 and 3 servers down, respectively. If you have a small deployment, then using 3 servers is acceptable, but keep in mind that you'll only be able to tolerate 1 server down in this case.
- I\/O segregation: if you do a lot of write type traffic you'll almost definitely want the transaction logs on a dedicated disk group. Writes to the transaction log are synchronous (but batched for performance), and consequently, concurrent writes can significantly affect performance. ZooKeeper snapshots can be one such a source of concurrent writes, and ideally should be written on a disk group separate from the transaction log. Snapshots are writtent to disk asynchronously, so it is typically ok to share with the operating system and message log files. You can configure a server to use a separate disk group with the dataLogDir parameter.
- Application segregation: Unless you really understand the application patterns of other apps that you want to install on the same box, it can be a good idea to run ZooKeeper in isolation (though this can be a balancing act with the capabilities of the hardware).
- Use care with virtualization: It can work, depending on your cluster layout and read\/write patterns and SLAs, but the tiny overheads introduced by the virtualization layer can add up and throw off ZooKeeper, as it can be very time sensitive
- ZooKeeper configuration: It's java, make sure you give it 'enough' heap space (We usually run them with 3-5G, but that's mostly due to the data set size we have here). Unfortunately we don't have a good formula for it, but keep in mind that allowing for more ZooKeeper state means that snapshots can become large, and large snapshots affect recovery time. In fact, if the snapshot becomes too large (a few gigabytes), then

you may need to increase the initLimit parameter to give enough time for servers to recover and join the ensemble.

- Monitoring: Both JMX and the 4 letter words (4lw) commands are very useful, they do overlap in some cases (and in those cases we prefer the 4 letter commands, they seem more predictable, or at the very least, they work better with the LI monitoring infrastructure)
- Don't overbuild the cluster: large clusters, especially in a write heavy usage pattern, means a lot of intracluster communication (quorums on the writes and subsequent cluster member updates), but don't underbuild it (and risk swamping the cluster). Having more servers adds to your read capacity.

Overall, we try to keep the ZooKeeper system as small as will handle the load (plus standard growth capacity planning) and as simple as possible. We try not to do anything fancy with the configuration or application layout as compared to the official release as well as keep it as self contained as possible. For these reasons, we tend to skip the OS packaged versions, since it has a tendency to try to put things in the OS standard hierarchy, which can be 'messy', for want of a better way to word it.

7. Security

7.1 Security Overview

In release 0.9.0.0, the Kafka community added a number of features that, used either separately or together, increases security in a Kafka cluster. These features are considered to be of beta quality. The following security measures are currently supported:

1. Authentication of connections to brokers from clients (producers and consumers), other brokers and tools, using either SSL or SASL (Kerberos). SASL/PLAIN can also be used from release 0.10.0.0 onwards.
2. Authentication of connections from brokers to ZooKeeper
3. Encryption of data transferred between brokers and clients, between brokers, or between brokers and tools using SSL (Note that there is a performance degradation when SSL is enabled, the magnitude of which depends on the CPU type and the JVM implementation.)
4. Authorization of read / write operations by clients
5. Authorization is pluggable and integration with external authorization services is supported

It's worth noting that security is optional - non-secured clusters are supported, as well as a mix of authenticated, unauthenticated, encrypted and non-encrypted clients. The guides below explain how to configure and use the security features in both clients and brokers.

7.2 Encryption and Authentication using SSL

Apache Kafka allows clients to connect over SSL. By default SSL is disabled but can be turned on as needed.

1. Generate SSL key and certificate for each Kafka broker

The first step of deploying HTTPS is to generate the key and the certificate for each machine in the cluster. You can use Java's keytool utility to accomplish this task. We will generate the key into a temporary keystore initially so that we can export and sign it later with CA.

```
      keytool -keystore server.keystore.jks -alias localhost -validity {validity}
-genkey
```

You need to specify two parameters in the above command:

i. keystore: the keystore file that stores the certificate. The keystore file contains the private key of the certificate; therefore, it needs to be kept safely.
ii. validity: the valid time of the certificate in days.

Note: By default the property `ssl.endpoint.identification.algorithm` is not defined, so hostname verification is not performed. In order to enable hostname verification, set the following property:

```
ssl.endpoint.identification.algorithm=HTTPS
```

Once enabled, clients will verify the server's fully qualified domain name (FQDN) against one of the following two fields:

1. Common Name (CN)
2. Subject Alternative Name (SAN)

Both fields are valid, RFC-2818 recommends the use of SAN however. SAN is also more flexible, allowing for multiple DNS entries to be declared. Another advantage is that the CN can be set to a more meaningful value for authorization purposes. To add a SAN field append the following argument `-ext SAN=DNS:{FQDN}` to the keytool command:

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey -ext SAN=DNS:{FQDN}
```

The following command can be run afterwards to verify the contents of the generated certificate:

```
keytool -list -v -keystore server.keystore.jks
```

1. ## Creating your own CA

   After the first step, each machine in the cluster has a public-private key pair, and a certificate to identify the machine. The certificate, however, is unsigned, which means that an attacker can create such a certificate to pretend to be any machine. Therefore, it is important to prevent forged certificates by signing them for each machine in the cluster. A certificate authority (CA) is responsible for signing certificates. CA works likes a government that issues passports—the government stamps (signs) each passport so that the passport becomes difficult to forge. Other governments verify the stamps to ensure the passport is authentic. Similarly, the CA signs the certificates, and the

cryptography guarantees that a signed certificate is computationally difficult to forge. Thus, as long as the CA is a genuine and trusted authority, the clients have high assurance that they are connecting to the authentic machines.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

The generated CA is simply a public-private key pair and certificate, and it is intended to sign other certificates.

The next step is to add the generated CA to the **clients' truststore** so that the clients can trust this CA:

```
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

Note: If you configure the Kafka brokers to require client authentication by setting ssl.client.auth to be "requested" or "required" on the **Kafka brokers config** then you must provide a truststore for the Kafka brokers as well and it should have all the CA certificates that clients keys were signed by.

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

In contrast to the keystore in step 1 that stores each machine's own identity, the truststore of a client stores all the certificates that the client should trust. Importing a certificate into one's truststore also means trusting all certificates that are signed by that certificate. As the analogy above, trusting the government (CA) also means trusting all passports (certificates) that it has issued. This attribute is called the chain of trust, and it is particularly useful when deploying SSL on a large Kafka cluster. You can sign all certificates in the cluster with a single CA, and have all machines share the same truststore that trusts the CA. That way all machines can authenticate all other machines.

2. Signing the certificate

The next step is to sign all certificates generated by step 1 with the CA generated in step 2. First, you need to export the certificate from the keystore:

```
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-
file
```

Then sign it with the CA:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed
-days {validity} -CAcreateserial -passin pass:{ca-password}
```

Finally, you need to import both the certificate of the CA and the signed certificate into the keystore:

```
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore server.keystore.jks -alias localhost -import -file cert-s
igned
```

The definitions of the parameters are the following:

i. keystore: the location of the keystore
ii. ca-cert: the certificate of the CA
iii. ca-key: the private key of the CA
iv. ca-password: the passphrase of the CA
v. cert-file: the exported, unsigned certificate of the server
vi. cert-signed: the signed certificate of the server

Here is an example of a bash script with all above steps. Note that one of the commands assumes a password of `test1234`, so either use that password or edit the command before running it.

```
#!/bin/bash
#Step 1
keytool -keystore server.keystore.jks -alias localhost -validity 365 -keyal
g RSA -genkey
#Step 2
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
#Step 3
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-
file
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed
-days 365 -CAcreateserial -passin pass:test1234
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore server.keystore.jks -alias localhost -import -file cert-s
igned
```

3. Configuring Kafka Brokers

Kafka Brokers support listening for connections on multiple ports. We need to configure the following property in server.properties, which must have one or more comma-separated values:

```
listeners
```

If SSL is not enabled for inter-broker communication (see below for how to enable it), both PLAINTEXT and SSL ports will be necessary.

```
    listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

Following SSL configs are needed on the broker side

```
    ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
    ssl.keystore.password=test1234
    ssl.key.password=test1234
    ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
    ssl.truststore.password=test1234
```

Optional settings that are worth considering:

i. ssl.client.auth=none ("required" => client authentication is required, "requested" => client authentication is requested and client without certs can still connect. The usage of "requested" is discouraged as it provides a false sense of security and misconfigured clients will still connect successfully.)
ii. ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. (Default is an empty list)
iii. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1 (list out the SSL protocols that you are going to accept from clients. Do note that SSL is deprecated in favor of TLS and using SSL in production is not recommended)
iv. ssl.keystore.type=JKS
v. ssl.truststore.type=JKS

If you want to enable SSL for inter-broker communication, add the following to the broker properties file (it defaults to PLAINTEXT)

```
    security.inter.broker.protocol=SSL
```

Due to import regulations in some countries, the Oracle implementation limits the strength of cryptographic algorithms available by default. If stronger algorithms are needed (for example, AES with 256-bit keys), the **JCE Unlimited Strength Jurisdiction Policy Files** must be obtained and installed in the JDK/JRE. See the **JCA Providers Documentation** for more information.

Once you start the broker you should be able to see in the server.log

```
        with addresses: PLAINTEXT -> EndPoint(192.168.64.1,9092,PLAINTEXT),SSL -> E
ndPoint(192.168.64.1,9093,SSL)
```

To check quickly if the server keystore and truststore are setup properly you can run the following command

```
openssl s_client -debug -connect localhost:9093 -tls1
```

(Note: TLSv1 should be listed under ssl.enabled.protocols)

In the output of this command you should see server's certificate:

```
        -----BEGIN CERTIFICATE-----
        {variable sized random bytes}
        -----END CERTIFICATE-----
        subject=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=Sriharsha Chintalapani
        issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddress=test@te
st.com
```

If the certificate does not show up or if there are any other error messages then your keystore is not setup properly.

4. Configuring Kafka Clients

SSL is supported only for the new Kafka Producer and Consumer, the older API is not supported. The configs for SSL will be the same for both producer and consumer.

If client authentication is not required in the broker, then the following is a minimal configuration example:

```
        security.protocol=SSL
        ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
        ssl.truststore.password=test1234
```

If client authentication is required, then a keystore must be created like in step 1 and the following must also be configured:

```
        ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks
        ssl.keystore.password=test1234
        ssl.key.password=test1234
```

Other configuration settings that may also be needed depending on our requirements and the broker configuration:

i. ssl.provider (Optional). The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
ii. ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.
iii. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. It should list at least one of the protocols configured on the broker side
iv. ssl.truststore.type=JKS
v. ssl.keystore.type=JKS

Examples using console-producer and console-consumer:

```
kafka-console-producer.sh --broker-list localhost:9093 --topic test --produc
er.config client-ssl.properties
kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic test --n
ew-consumer --consumer.config client-ssl.properties
```

7.3 Authentication using SASL

1. SASL configuration for Kafka brokers

i. Select one or more supported mechanisms to enable in the broker. GSSAPI and PLAIN are the mechanisms currently supported in Kafka.
ii. Add a JAAS config file for the selected mechanisms as described in the examples for setting up**GSSAPI (Kerberos)** or **PLAIN**.
iii. Pass the JAAS config file location as JVM parameter to each Kafka broker. For example:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

iv. Configure a SASL port in server.properties, by adding at least one of SASL*PLAINTEXT or SASL_SSL to the _listeners* parameter, which contains one or more comma-separated values:

```
listeners=SASL_PLAINTEXT://host.name:port
```

If SASL_SSL is used, then **SSL must also be configured**. If you are only configuring a SASL port (or if you want the Kafka brokers to authenticate each other using SASL) then make sure you set the same SASL protocol for inter-broker communication:

```
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
```

v. Enable one or more SASL mechanisms in server.properties:

```
sasl.enabled.mechanisms=GSSAPI (,PLAIN)
```

vi. Configure the SASL mechanism for inter-broker communication in server.properties if using SASL for inter-broker communication:

```
sasl.mechanism.inter.broker.protocol=GSSAPI (or PLAIN)
```

vii. Follow the steps in **GSSAPI (Kerberos)** or **PLAIN** to configure SASL for the enabled mechanisms. To enable multiple mechanisms in the broker, follow the steps **here**.

viii. **Important notes:**
   i. KafkaServer is the section name in the JAAS file used by each KafkaServer/Broker. This section provides SASL configuration options for the broker including any SASL client connections made by the broker for inter-broker communication.
   ii. Client section is used to authenticate a SASL connection with zookeeper. It also allows the brokers to set SASL ACL on zookeeper nodes which locks these nodes down so that only the brokers can modify it. It is necessary to have the same principal name across all brokers. If you want to use a section name other than Client, set the system propertyzookeeper.sasl.client to the appropriate name (*e.g.*, -Dzookeeper.sasl.client=ZkClient).
   iii. ZooKeeper uses "zookeeper" as the service name by default. If you want to change this, set the system property zookeeper.sasl.client.username to the appropriate name (*e.g.*, -Dzookeeper.sasl.client.username=zk).

1. SASL configuration for Kafka clients

SASL authentication is only supported for the new Java Kafka producer and consumer, the older API is not supported. To configure SASL authentication on the clients:

i. Select a SASL mechanism for authentication.

ii. Add a JAAS config file for the selected mechanism as described in the examples for setting up**GSSAPI (Kerberos)** or **PLAIN**. KafkaClient is the section name in the JAAS file used by Kafka clients.

iii. Pass the JAAS config file location as JVM parameter to each client JVM. For example:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
```

iv. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI (or PLAIN)
```

v. Follow the steps in **GSSAPI (Kerberos)** or **PLAIN** to configure SASL for the selected mechanism.

2. Authentication using SASL/Kerberos

i. Prerequisites

i. Kerberos

If your organization is already using a Kerberos server (for example, by using Active Directory), there is no need to install a new server just for Kafka. Otherwise you will need to install one, your Linux vendor likely has packages for Kerberos and a short guide on how to install and configure it (**Ubuntu**, **Redhat**). Note that if you are using Oracle Java, you will need to download JCE policy files for your Java version and copy them to $JAVA_HOME/jre/lib/security.

i. Create Kerberos Principals

If you are using the organization's Kerberos or Active Directory server, ask your Kerberos administrator for a principal for each Kafka broker in your cluster and for every operating system user that will access Kafka with Kerberos authentication (via clients and tools).

If you have installed your own Kerberos, you will need to create these principals yourself using the following commands:

```
sudo /usr/sbin/kadmin.local -q 'addprinc -randkey kafka/{hostname}@{REALM}'
sudo /usr/sbin/kadmin.local -q "ktadd -k /etc/security/keytabs/{keytabname}.keytab kafka/{hostname}@{REALM}"
```

i. **Make sure all hosts can be reachable using hostnames** - it is a Kerberos requirement that all your hosts can be resolved with their FQDNs.

ii. **Configuring Kafka Brokers**

i. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka_server_jaas.conf for this example (note that each broker should have its own keytab):

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};

// Zookeeper client authentication
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};
```

ii. KafkaServer section in the JAAS file tells the broker which principal to use and the location of the keytab where this principal is stored. It allows the broker to login using the keytab specified in this section. See **notes** for more details on Zookeeper SASL configuration.

iii. Pass the JAAS and optionally the krb5 file locations as JVM parameters to each Kafka broker (see **here** for more details):

```
-Djava.security.krb5.conf=/etc/kafka/krb5.conf
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

iv. Make sure the keytabs configured in the JAAS file are readable by the operating system user who is starting kafka broker.

v. Configure SASL port and SASL mechanisms in server.properties as described **here**. For example: ``` listeners=SASL_PLAINTEXT://host.name:port
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.enabled.mechanisms=GSSAPI

```
6. We must also configure the service name in server.properties, which should
 match the principal name of the kafka brokers. In the above example, princip
al is "kafka/kafka1.hostname.com@EXAMPLE.com", so:

sasl.kerberos.service.name=kafka
```

```

1. **Configuring Kafka Clients**

   To configure SASL authentication on the clients:

   i. Clients (producers, consumers, connect workers, etc) will authenticate to the cluster with their own principal (usually with the same name as the user running the client), so obtain or create these principals as needed. Then create a JAAS file for each principal. The KafkaClient section describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client using a keytab (recommended for long-running processes):

   ```
   KafkaClient {
       com.sun.security.auth.module.Krb5LoginModule required
       useKeyTab=true
       storeKey=true
       keyTab="/etc/security/keytabs/kafka_client.keytab"
       principal="kafka-client-1@EXAMPLE.COM";
   };
   ```

   For command-line utilities like kafka-console-consumer or kafka-console-producer, kinit can be used along with "useTicketCache=true" as in:

   ```
   KafkaClient {
       com.sun.security.auth.module.Krb5LoginModule required
       useTicketCache=true;
   };
   ```

   ii. Pass the JAAS and optionally krb5 file locations as JVM parameters to each client JVM (see**here** for more details):

   ```
   -Djava.security.krb5.conf=/etc/kafka/krb5.conf
   -Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
   ```
   ```

iii. Make sure the keytabs configured in the kafka_client_jaas.conf are readable by the operating system user who is starting kafka client.

iv. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka
```

1. Authentication using SASL/PLAIN

SASL/PLAIN is a simple username/password authentication mechanism that is typically used with TLS for encryption to implement secure authentication. Kafka supports a default implementation for SASL/PLAIN which can be extended for production use as described **here**.

The username is used as the authenticated `Principal` for configuration of ACLs etc.

i. Configuring Kafka Brokers

i. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka_server_jaas.conf for this example:

```
KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret"
    user_alice="alice-secret";
};
```

This configuration defines two users (*admin* and *alice*). The properties username andpassword in the KafkaServer section are used by the broker to initiate connections to other brokers. In this example, *admin* is the user for inter-broker communication. The set of properties user_*userName* defines the passwords for all users that connect to the broker and the broker validates all client connections including those from other brokers using these properties.

i. Pass the JAAS config file location as JVM parameter to each Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

ii. Configure SASL port and SASL mechanisms in server.properties as described **here**. For example:

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```

1. **Configuring Kafka Clients**

   To configure SASL authentication on the clients:

   i. The KafkaClient section describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client for the PLAIN mechanism:

   ```
   KafkaClient {
       org.apache.kafka.common.security.plain.PlainLoginModule required
       username="alice"
       password="alice-secret";
   };
   ```

   The properties username and password in the KafkaClient section are used by clients to configure the user for client connections. In this example, clients connect to the broker as user *alice*.

   ii. Pass the JAAS config file location as JVM parameter to each client JVM:

   ```
   -Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
   ```

   iii. Configure the following properties in producer.properties or consumer.properties:

   ```
   security.protocol=SASL_SSL
   sasl.mechanism=PLAIN
   ```

1. **Use of SASL/PLAIN in production**

   - SASL/PLAIN should be used only with SSL as transport layer to ensure that clear passwords are not transmitted on the wire without encryption.
   - The default implementation of SASL/PLAIN in Kafka specifies usernames and passwords in the JAAS configuration file as shown **here**. To avoid storing passwords on disk, you can plugin your own implementation of `javax.security.auth.spi.LoginModule` that provides usernames and passwords from an external source. The login module implementation should provide username as the public credential and password as the private credential of the `Subject`. The default

implementation `org.apache.kafka.common.security.plain.PlainLoginModule` can be used as an example.

- In production systems, external authentication servers may implement password authentication. Kafka brokers can be integrated with these servers by adding your own implementation of `javax.security.sasl.SaslServer`. The default implementation included in Kafka in the package `org.apache.kafka.common.security.plain` can be used as an example to get started.
  - New providers must be installed and registered in the JVM. Providers can be installed by adding provider classes to the normal CLASSPATH or bundled as a jar file and added to*JAVA_HOME*/lib/ext.
  - Providers can be registered statically by adding a provider to the security properties file*JAVA_HOME*/lib/security/java.security.

```
security.provider.n=providerClassName
```

    where *providerClassName* is the fully qualified name of the new provider and *n* is the preference order with lower numbers indicating higher preference.

  - Alternatively, you can register providers dynamically at runtime by invoking `Security.addProvider` at the beginning of the client application or in a static initializer in the login module. For example:

```
Security.addProvider(new PlainSaslServerProvider());
```

  - For more details, see **JCA Reference**.

1. ## Enabling multiple SASL mechanisms in a broker

   i. Specify configuration for the login modules of all enabled mechanisms in the KafkaServersection of the JAAS config file. For example:

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";

    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret"
    user_alice="alice-secret";
};
```

ii. Enable the SASL mechanisms in server.properties:

```
sasl.enabled.mechanisms=GSSAPI,PLAIN
```

iii. Specify the SASL security protocol and mechanism for inter-broker communication in server.properties if required:

```
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism.inter.broker.protocol=GSSAPI (or PLAIN)
```

iv. Follow the mechanism-specific steps in **GSSAPI (Kerberos)** and **PLAIN** to configure SASL for the enabled mechanisms.

2. Modifying SASL mechanism in a Running Cluster

SASL mechanism can be modified in a running cluster using the following sequence:

i. Enable new SASL mechanism by adding the mechanism to sasl.enabled.mechanisms in server.properties for each broker. Update JAAS config file to include both mechanisms as describedhere. Incrementally bounce the cluster nodes.
ii. Restart clients using the new mechanism.
iii. To change the mechanism of inter-broker communication (if this is required), setsasl.mechanism.inter.broker.protocol in server.properties to the new mechanism and incrementally bounce the cluster again.
iv. To remove old mechanism (if this is required), remove the old mechanism fromsasl.enabled.mechanisms in server.properties and remove the entries for the old mechanism from JAAS config file. Incrementally bounce the cluster again.

7.4 Authorization and ACLs

Kafka ships with a pluggable Authorizer and an out-of-box authorizer implementation that uses zookeeper to store all the acls. Kafka acls are defined in the general format of "Principal P is [Allowed/Denied] Operation O From Host H On Resource R". You can read more about the acl structure on KIP-11. In order to add, remove or list acls you can use the Kafka authorizer CLI. By default, if a Resource R has no associated acls, no one other than super users is allowed to access R. If you want to change that behavior, you can include the following in broker.properties.

```
allow.everyone.if.no.acl.found=true
```

One can also add super users in broker.properties like the following (note that the delimiter is semicolon since SSL user names may contain comma).

```
super.users=User:Bob;User:Alice
```

By default, the SSL user name will be of the form "CN=writeuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown". One can change that by setting a customized PrincipalBuilder in broker.properties like the following.

```
principal.builder.class=CustomizedPrincipalBuilderClass
```

By default, the SASL user name will be the primary part of the Kerberos principal. One can change that by setting `sasl.kerberos.principal.to.local.rules` to a customized rule in broker.properties. The format of `sasl.kerberos.principal.to.local.rules` is a list where each rule works in the same way as the auth_to_local in **Kerberos configuration file (krb5.conf)**. Each rules starts with RULE: and contains an expression in the format n:strings/pattern/replacement/g. See the kerberos documentation for more details. An example of adding a rule to properly translate user@MYDOMAIN.COM to user while also keeping the default rule in place is:

```
sasl.kerberos.principal.to.local.rules=RULE:[1:$1@$0](.*@MYDOMAIN.COM)s/@.*//,DEFAULT
```

Command Line Interface

Kafka Authorization management CLI can be found under bin directory with all the other CLIs. The CLI script is called **kafka-acls.sh**. Following lists all the options that the script supports:

OptionDescriptionDefaultOption type		
--add	Indicates to the script that user is trying to add an acl.	
--remove	Indicates to the script that user is trying to remove an acl.	
--list	Indicates to the script that user is trying to list acls.	
--authorizer	Fully qualified class name of the authorizer.	kafka.security
--authorizer-properties	key=val pairs that will be passed to authorizer for initialization. For the default authorizer the example values are: zookeeper.connect=localhost:2181	
--cluster	Specifies cluster as resource.	
--topic [topic-name]	Specifies the topic as resource.	
--group [group-name]	Specifies the consumer-group as resource.	
--allow-principal	Principal is in PrincipalType:name format that will be added to ACL with Allow permission.	

You can specify multiple --allow-principal in a single command. | | Principal | | --deny-principal | Principal is in PrincipalType:name format that will be added to ACL with Deny permission.

You can specify multiple --deny-principal in a single command. | | Principal | | --allow-host | IP address from which principals listed in --allow-principal will have access. | if --allow-principal is specified defaults to *which translates to "all hosts"* | *Host* | | *--deny-host* | *IP address from which principals listed in --deny-principal will be denied access.* | *if --deny-principal is specified defaults to* which translates to "all hosts" | Host | | --operation | Operation that will be allowed or denied.

Valid values are : Read, Write, Create, Delete, Alter, Describe, ClusterAction, All | All | Operation | | --producer | Convenience option to add/remove acls for producer role. This will generate acls that allows WRITE, DESCRIBE on topic and CREATE on cluster. | | Convenience | | --consumer | Convenience option to add/remove acls for consumer role. This will generate acls that allows READ, DESCRIBE on topic and READ on consumer-group. | | Convenience |

Examples

- **Adding Acls**

  Suppose you want to add an acl "Principals User:Bob and User:Alice are allowed to perform Operation Read and Write on Topic Test-Topic from IP 198.51.100.0 and IP 198.51.100.1". You can do that by executing the CLI with following options:

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add -
-allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100.0 -
-allow-host 198.51.100.1 --operation Read --operation Write --topic Test-topic
```

  By default all principals that don't have an explicit acl that allows access for an operation to a resource are denied. In rare cases where an allow acl is defined that allows access to all but some principal we will have to use the --deny-principal and --deny-host option. For example, if we want to allow all users to Read from Test-topic but only deny User:BadBob from IP 198.51.100.3 we can do so using following commands:

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add -
-allow-principal User:* --allow-host * --deny-principal User:BadBob --deny-host 19
8.51.100.3 --operation Read --topic Test-topic
```

  Note that `--allow-host` and `deny-host` only support IP addresses (hostnames are not supported). Above examples add acls to a topic by specifying --topic [topic-name] as the resource option. Similarly user can add acls to cluster by specifying --cluster and to a consumer group by specifying --group [group-name].

- **Removing Acls**

  Removing acls is pretty much the same. The only difference is instead of --add option users will have to specify --remove option. To remove the acls added by the first example above we can execute the CLI with following options:

```
 bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --remo
ve --allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100
.0 --allow-host 198.51.100.1 --operation Read --operation Write --topic Test-topic
```

- **List Acls**

  We can list acls for any resource by specifying the --list option with the resource. To list all acls for Test-topic we can execute the CLI with following options:

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --list
--topic Test-topic
```

- **Adding or removing a principal as producer or consumer**

  The most common use case for acl management are adding/removing a principal as producer or consumer so we added convenience options to handle these cases. In order to add User:Bob as a producer of Test-topic we can execute the following command:

```
 bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add
--allow-principal User:Bob --producer --topic Test-topic
```

  Similarly to add Alice as a consumer of Test-topic with consumer group Group-1 we just have to pass --consumer option:

```
 bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add
--allow-principal User:Bob --consumer --topic test-topic --group Group-1
```

  Note that for consumer option we must also specify the consumer group. In order to remove a principal from producer or consumer role we just need to pass --remove option.

7.5 Incorporating Security Features in a Running Cluster

You can secure a running cluster via one or more of the supported protocols discussed previously. This is done in phases:

- Incrementally bounce the cluster nodes to open additional secured port(s).
- Restart clients using the secured rather than PLAINTEXT port (assuming you are securing the client-broker connection).
- Incrementally bounce the cluster again to enable broker-to-broker security (if this is required)
- A final incremental bounce to close the PLAINTEXT port.

The specific steps for configuring SSL and SASL are described in sections 7.2 and 7.3. Follow these steps to enable security for your desired protocol(s).

The security implementation lets you configure different protocols for both broker-client and broker-broker communication. These must be enabled in separate bounces. A PLAINTEXT port must be left open throughout so brokers and/or clients can continue to communicate.

When performing an incremental bounce stop the brokers cleanly via a SIGTERM. It's also good practice to wait for restarted replicas to return to the ISR list before moving onto the next node.

As an example, say we wish to encrypt both broker-client and broker-broker communication with SSL. In the first incremental bounce, a SSL port is opened on each node:

```
    listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
```

We then restart the clients, changing their config to point at the newly opened, secured port:

```
    bootstrap.servers = [broker1:9092,...]
    security.protocol = SSL
    ...etc
```

In the second incremental server bounce we instruct Kafka to use SSL as the broker-broker protocol (which will use the same SSL port):

```
    listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
    security.inter.broker.protocol=SSL
```

In the final bounce we secure the cluster by closing the PLAINTEXT port:

```
    listeners=SSL://broker1:9092
    security.inter.broker.protocol=SSL
```

Alternatively we might choose to open multiple ports so that different protocols can be used for broker-broker and broker-client communication. Say we wished to use SSL encryption throughout (i.e. for broker-broker and broker-client communication) but we'd like to add SASL authentication to the broker-client connection also. We would achieve this by opening two additional ports during the first bounce:

```
    listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://broker1:9093
```

We would then restart the clients, changing their config to point at the newly opened, SASL & SSL secured port:

```
    bootstrap.servers = [broker1:9093,...]
    security.protocol = SASL_SSL
    ...etc
```

The second server bounce would switch the cluster to use encrypted broker-broker communication via the SSL port we previously opened on port 9092:

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://broker1:9093
security.inter.broker.protocol=SSL
```

The final bounce secures the cluster by closing the PLAINTEXT port.

```
listeners=SSL://broker1:9092,SASL_SSL://broker1:9093
security.inter.broker.protocol=SSL
```

ZooKeeper can be secured independently of the Kafka cluster. The steps for doing this are covered in section7.6.2.

7.6 ZooKeeper Authentication

7.6.1 New clusters

To enable ZooKeeper authentication on brokers, there are two necessary steps:

1. Create a JAAS login file and set the appropriate system property to point to it as described above
2. Set the configuration property zookeeper.set.acl in each broker to true

The metadata stored in ZooKeeper is such that only brokers will be able to modify the corresponding znodes, but znodes are world readable. The rationale behind this decision is that the data stored in ZooKeeper is not sensitive, but inappropriate manipulation of znodes can cause cluster disruption. We also recommend limiting the access to ZooKeeper via network segmentation (only brokers and some admin tools need access to ZooKeeper if the new consumer and new producer are used).

7.6.2 Migrating clusters

If you are running a version of Kafka that does not support security or simply with security disabled, and you want to make the cluster secure, then you need to execute the following steps to enable ZooKeeper authentication with minimal disruption to your operations:

1. Perform a rolling restart setting the JAAS login file, which enables brokers to authenticate. At the end of the rolling restart, brokers are able to manipulate znodes with strict ACLs, but they will not create znodes with those ACLs
2. Perform a second rolling restart of brokers, this time setting the configuration parameterzookeeper.set.acl to true, which enables the use of secure ACLs when creating znodes
3. Execute the ZkSecurityMigrator tool. To execute the tool, there is this script: ./bin/zookeeper-security-migration.sh with zookeeper.acl set to secure. This tool

traverses the corresponding sub-trees changing the ACLs of the znodes

It is also possible to turn off authentication in a secure cluster. To do it, follow these steps:

1. Perform a rolling restart of brokers setting the JAAS login file, which enables brokers to authenticate, but setting zookeeper.set.acl to false. At the end of the rolling restart, brokers stop creating znodes with secure ACLs, but are still able to authenticate and manipulate all znodes
2. Execute the ZkSecurityMigrator tool. To execute the tool, run this script ./bin/zookeeper-security-migration.sh with zookeeper.acl set to unsecure. This tool traverses the corresponding sub-trees changing the ACLs of the znodes
3. Perform a second rolling restart of brokers, this time omitting the system property that sets the JAAS login file

Here is an example of how to run the migration tool:

```
./bin/zookeeper-security-migration --zookeeper.acl=secure --zookeeper.connection=local
host:2181
```

Run this to see the full list of parameters:

```
./bin/zookeeper-security-migration --help
```

7.6.3 Migrating the ZooKeeper ensemble

It is also necessary to enable authentication on the ZooKeeper ensemble. To do it, we need to perform a rolling restart of the server and set a few properties. Please refer to the ZooKeeper documentation for more detail:

1. **Apache ZooKeeper documentation**
2. **Apache ZooKeeper wiki**

8. Kafka Connect

8.1 Overview

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It makes it simple to quickly define *connectors* that move large collections of data into and out of Kafka. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export job can deliver data from Kafka topics into secondary storage and query systems or into batch systems for offline analysis. Kafka Connect features include:

- **A common framework for Kafka connectors** - Kafka Connect standardizes integration of other data systems with Kafka, simplifying connector development, deployment, and management
- **Distributed and standalone modes** - scale up to a large, centrally managed service supporting an entire organization or scale down to development, testing, and small production deployments
- **REST interface** - submit and manage connectors to your Kafka Connect cluster via an easy to use REST API
- **Automatic offset management** - with just a little information from connectors, Kafka Connect can manage the offset commit process automatically so connector developers do not need to worry about this error prone part of connector development
- **Distributed and scalable by default** - Kafka Connect builds on the existing group management protocol. More workers can be added to scale up a Kafka Connect cluster.
- **Streaming/batch integration** - leveraging Kafka's existing capabilities, Kafka Connect is an ideal solution for bridging streaming and batch data systems

8.2 User Guide

The quickstart provides a brief example of how to run a standalone version of Kafka Connect. This section describes how to configure, run, and manage Kafka Connect in more detail.

Running Kafka Connect

Kafka Connect currently supports two modes of execution: standalone (single process) and distributed. In standalone mode all work is performed in a single process. This configuration is simpler to setup and get started with and may be useful in situations where only one

worker makes sense (e.g. collecting log files), but it does not benefit from some of the features of Kafka Connect such as fault tolerance. You can start a standalone process with the following command:

```
> bin/connect-standalone.sh config/connect-standalone.properties connector1.properties
 [connector2.properties ...]
```

The first parameter is the configuration for the worker. This includes settings such as the Kafka connection parameters, serialization format, and how frequently to commit offsets. The provided example should work well with a local cluster running with the default configuration provided by `config/server.properties`. It will require tweaking to use with a different configuration or production deployment. The remaining parameters are connector configuration files. You may include as many as you want, but all will execute within the same process (on different threads). Distributed mode handles automatic balancing of work, allows you to scale up (or down) dynamically, and offers fault tolerance both in the active tasks and for configuration and offset commit data. Execution is very similar to standalone mode:

```
> bin/connect-distributed.sh config/connect-distributed.properties
```

The difference is in the class which is started and the configuration parameters which change how the Kafka Connect process decides where to store configurations, how to assign work, and where to store offsets and task statues. In the distributed mode, Kafka Connect stores the offsets, configs and task statuses in Kafka topics. It is recommended to manually create the topics for offset, configs and statuses in order to achieve the desired the number of partitions and replication factors. If the topics are not yet created when starting Kafka Connect, the topics will be auto created with default number of partitions and replication factor, which may not be best suited for its usage. In particular, the following configuration parameters are critical to set before starting your cluster:

- `group.id` (default `connect-cluster`) - unique name for the cluster, used in forming the Connect cluster group; note that this **must not conflict** with consumer group IDs
- `config.storage.topic` (default `connect-configs`) - topic to use for storing connector and task configurations; note that this should be a single partition, highly replicated topic. You may need to manually create the topic to ensure single partition for the config topic as auto created topics may have multiple partitions.
- `offset.storage.topic` (default `connect-offsets`) - topic to use for storing offsets; this topic should have many partitions and be replicated
- `status.storage.topic` (default `connect-status`) - topic to use for storing statuses; this topic can have multiple partitions and should be replicated

Note that in distributed mode the connector configurations are not passed on the command line. Instead, use the REST API described below to create, modify, and destroy connectors.

Configuring Connectors

Connector configurations are simple key-value mappings. For standalone mode these are defined in a properties file and passed to the Connect process on the command line. In distributed mode, they will be included in the JSON payload for the request that creates (or modifies) the connector. Most configurations are connector dependent, so they can't be outlined here. However, there are a few common options:

- `name` - Unique name for the connector. Attempting to register again with the same name will fail.
- `connector.class` - The Java class for the connector
- `tasks.max` - The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot achieve this level of parallelism.

The `connector.class` config supports several formats: the full name or alias of the class for this connector. If the connector is org.apache.kafka.connect.file.FileStreamSinkConnector, you can either specify this full name or use FileStreamSink or FileStreamSinkConnector to make the configuration a bit shorter. Sink connectors also have one additional option to control their input:

- `topics` - A list of topics to use as input for this connector

For any other options, you should consult the documentation for the connector.

REST API

Since Kafka Connect is intended to be run as a service, it also provides a REST API for managing connectors. By default this service runs on port 8083. The following are the currently supported endpoints:

- `GET /connectors` - return a list of active connectors
- `POST /connectors` - create a new connector; the request body should be a JSON object containing a string `name` field and a object `config` field with the connector configuration parameters
- `GET /connectors/{name}` - get information about a specific connector
- `GET /connectors/{name}/config` - get the configuration parameters for a specific connector
- `PUT /connectors/{name}/config` - update the configuration parameters for a specific connector
- `GET /connectors/{name}/status` - get current status of the connector, including if it is

running, failed, paused, etc., which worker it is assigned to, error information if it has failed, and the state of all its tasks
- `GET /connectors/{name}/tasks` - get a list of tasks currently running for a connector
- `GET /connectors/{name}/tasks/{taskid}/status` - get current status of the task, including if it is running, failed, paused, etc., which worker it is assigned to, and error information if it has failed
- `PUT /connectors/{name}/pause` - pause the connector and its tasks, which stops message processing until the connector is resumed
- `PUT /connectors/{name}/resume` - resume a paused connector (or do nothing if the connector is not paused)
- `POST /connectors/{name}/restart` - restart a connector (typically because it has failed)
- `POST /connectors/{name}/tasks/{taskId}/restart` - restart an individual task (typically because it has failed)
- `DELETE /connectors/{name}` - delete a connector, halting all tasks and deleting its configuration

Kafka Connect also provides a REST API for getting information about connector plugins:

- `GET /connector-plugins` - return a list of connector plugins installed in the Kafka Connect cluster. Note that the API only checks for connectors on the worker that handles the request, which means you may see inconsistent results, especially during a rolling upgrade if you add new connector jars
- `PUT /connector-plugins/{connector-type}/config/validate` - validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

8.3 Connector Development Guide

This guide describes how developers can write new connectors for Kafka Connect to move data between Kafka and other systems. It briefly reviews a few key concepts and then describes how to create a simple connector.

Core Concepts and APIs

Connectors and Tasks

To copy data between Kafka and another system, users create a `Connector` for the system they want to pull data from or push data to. Connectors come in two flavors: `SourceConnectors` import data from another system (e.g. `JDBCSourceConnector` would import a relational database into Kafka) and `SinkConnectors` export data (e.g. `HDFSSinkConnector` would export the contents of a Kafka topic to an HDFS file). `Connectors` do not perform any data copying themselves: their configuration describes the data to be copied, and the

`Connector` is responsible for breaking that job into a set of `Tasks` that can be distributed to workers. These `Tasks` also come in two corresponding flavors: `SourceTask` and `SinkTask`. With an assignment in hand, each `Task` must copy its subset of the data to or from Kafka. In Kafka Connect, it should always be possible to frame these assignments as a set of input and output streams consisting of records with consistent schemas. Sometimes this mapping is obvious: each file in a set of log files can be considered a stream with each parsed line forming a record using the same schema and offsets stored as byte offsets in the file. In other cases it may require more effort to map to this model: a JDBC connector can map each table to a stream, but the offset is less clear. One possible mapping uses a timestamp column to generate queries incrementally returning new data, and the last queried timestamp can be used as the offset.

Streams and Records

Each stream should be a sequence of key-value records. Both the keys and values can have complex structure -- many primitive types are provided, but arrays, objects, and nested data structures can be represented as well. The runtime data format does not assume any particular serialization format; this conversion is handled internally by the framework. In addition to the key and value, records (both those generated by sources and those delivered to sinks) have associated stream IDs and offsets. These are used by the framework to periodically commit the offsets of data that have been processed so that in the event of failures, processing can resume from the last committed offsets, avoiding unnecessary reprocessing and duplication of events.

Dynamic Connectors

Not all jobs are static, so `Connector` implementations are also responsible for monitoring the external system for any changes that might require reconfiguration. For example, in the `JDBCSourceConnector` example, the `Connector` might assign a set of tables to each `Task`. When a new table is created, it must discover this so it can assign the new table to one of the `Tasks` by updating its configuration. When it notices a change that requires reconfiguration (or a change in the number of `Tasks`), it notifies the framework and the framework updates any corresponding `Tasks`.

Developing a Simple Connector

Developing a connector only requires implementing two interfaces, the `Connector` and `Task`. A simple example is included with the source code for Kafka in the `file` package. This connector is meant for use in standalone mode and has implementations of a `SourceConnector`/`SourceTask` to read each line of a file and emit it as a record and a

`SinkConnector` / `SinkTask` that writes each record to a file. The rest of this section will walk through some code to demonstrate the key steps in creating a connector, but developers should also refer to the full example source code as many details are omitted for brevity.

Connector Example

We'll cover the `SourceConnector` as a simple example. `SinkConnector` implementations are very similar. Start by creating the class that inherits from `SourceConnector` and add a couple of fields that will store parsed configuration information (the filename to read from and the topic to send data to):

```
public class FileStreamSourceConnector extends SourceConnector {
    private String filename;
    private String topic;
```

The easiest method to fill in is `getTaskClass()`, which defines the class that should be instantiated in worker processes to actually read the data:

```
@Override
public Class<? extends Task> getTaskClass() {
    return FileStreamSourceTask.class;
}
```

We will define the `FileStreamSourceTask` class below. Next, we add some standard lifecycle methods, `start()` and `stop()`:

```
@Override
public void start(Map<String, String> props) {
    // The complete version includes error handling as well.
    filename = props.get(FILE_CONFIG);
    topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
    // Nothing to do since no background monitoring is required.
}
```

Finally, the real core of the implementation is in `getTaskConfigs()`. In this case we are only handling a single file, so even though we may be permitted to generate more tasks as per the `maxTasks` argument, we return a list with only one entry:

```
@Override
public List<Map<String, String>> getTaskConfigs(int maxTasks) {
    ArrayList>Map<String, String>> configs = new ArrayList<>();
    // Only one input stream makes sense.
    Map<String, String> config = new Map<>();
    if (filename != null)
        config.put(FILE_CONFIG, filename);
    config.put(TOPIC_CONFIG, topic);
    configs.add(config);
    return configs;
}
```

Although not used in the example, `SourceTask` also provides two APIs to commit offsets in the source system: `commit` and `commitRecord`. The APIs are provided for source systems which have an acknowledgement mechanism for messages. Overriding these methods allows the source connector to acknowledge messages in the source system, either in bulk or individually, once they have been written to Kafka. The `commit` API stores the offsets in the source system, up to the offsets that have been returned by `poll`. The implementation of this API should block until the commit is complete. The `commitRecord` API saves the offset in the source system for each `SourceRecord` after it is written to Kafka. As Kafka Connect will record offsets automatically, `SourceTask`s are not required to implement them. In cases where a connector does need to acknowledge messages in the source system, only one of the APIs is typically required. Even with multiple tasks, this method implementation is usually pretty simple. It just has to determine the number of input tasks, which may require contacting the remote service it is pulling data from, and then divvy them up. Because some patterns for splitting work among tasks are so common, some utilities are provided in `ConnectorUtils` to simplify these cases. Note that this simple example does not include dynamic input. See the discussion in the next section for how to trigger updates to task configs.

Task Example - Source Task

Next we'll describe the implementation of the corresponding `SourceTask`. The implementation is short, but too long to cover completely in this guide. We'll use pseudo-code to describe most of the implementation, but you can refer to the source code for the full example. Just as with the connector, we need to create a class inheriting from the appropriate base `Task` class. It also has some standard lifecycle methods:

```
public class FileStreamSourceTask extends SourceTask<Object, Object> {
    String filename;
    InputStream stream;
    String topic;

    public void start(Map<String, String> props) {
        filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
        stream = openOrThrowError(filename);
        topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    }

    @Override
    public synchronized void stop() {
        stream.close();
    }
```

These are slightly simplified versions, but show that that these methods should be relatively simple and the only work they should perform is allocating or freeing resources. There are two points to note about this implementation. First, the `start()` method does not yet handle resuming from a previous offset, which will be addressed in a later section. Second, the `stop()` method is synchronized. This will be necessary because `SourceTasks` are given a dedicated thread which they can block indefinitely, so they need to be stopped with a call from a different thread in the Worker. Next, we implement the main functionality of the task, the `poll()` method which gets events from the input system and returns a `List<SourceRecord>`:

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition = Collections.singletonMap("filena
me", filename);
                Map<String, Object> sourceOffset = Collections.singletonMap("position"
, streamOffset);
                records.add(new SourceRecord(sourcePartition, sourceOffset, topic, Sch
ema.STRING_SCHEMA, line));
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling stop. Allow t
o return
        // null, and driving thread will handle any shutdown if necessary.
    }
    return null;
}
```

Again, we've omitted some details, but we can see the important steps: the `poll()` method is going to be called repeatedly, and for each call it will loop trying to read records from the file. For each line it reads, it also tracks the file offset. It uses this information to create an output `SourceRecord` with four pieces of information: the source partition (there is only one, the single file being read), source offset (byte offset in the file), output topic name, and output value (the line, and we include a schema indicating this value will always be a string). Other variants of the `SourceRecord` constructor can also include a specific output partition and a key. Note that this implementation uses the normal Java `InputStream` interface and may sleep if data is not available. This is acceptable because Kafka Connect provides each task with a dedicated thread. While task implementations have to conform to the basic `poll()` interface, they have a lot of flexibility in how they are implemented. In this case, an NIO-based implementation would be more efficient, but this simple approach works, is quick to implement, and is compatible with older versions of Java.

Sink Tasks

The previous section described how to implement a simple `SourceTask`. Unlike `SourceConnector` and `SinkConnector`, `SourceTask` and `SinkTask` have very different interfaces because `SourceTask` uses a pull interface and `SinkTask` uses a push interface. Both share the common lifecycle methods, but the `SinkTask` interface is quite different:

```
public abstract class SinkTask implements Task {
    public void initialize(SinkTaskContext context) {
        this.context = context;
    }

    public abstract void put(Collection<SinkRecord> records);

    public abstract void flush(Map<TopicPartition, Long> offsets);
```

The `SinkTask` documentation contains full details, but this interface is nearly as simple as the `SourceTask`. The `put()` method should contain most of the implementation, accepting sets of `SinkRecords`, performing any required translation, and storing them in the destination system. This method does not need to ensure the data has been fully written to the destination system before returning. In fact, in many cases internal buffering will be useful so an entire batch of records can be sent at once, reducing the overhead of inserting events into the downstream data store. The `SinkRecords` contain essentially the same information as `SourceRecords`: Kafka topic, partition, offset and the event key and value. The `flush()` method is used during the offset commit process, which allows tasks to recover from failures and resume from a safe point such that no events will be missed. The method should push any outstanding data to the destination system and then block until the write has been acknowledged. The `offsets` parameter can often be ignored, but is useful in some cases where implementations want to store offset information in the destination store to provide exactly-once delivery. For example, an HDFS connector could do this and use atomic move operations to make sure the `flush()` operation atomically commits the data and offsets to a final location in HDFS.

Resuming from Previous Offsets

The `SourceTask` implementation included a stream ID (the input filename) and offset (position in the file) with each record. The framework uses this to commit offsets periodically so that in the case of a failure, the task can recover and minimize the number of events that are reprocessed and possibly duplicated (or to resume from the most recent offset if Kafka Connect was stopped gracefully, e.g. in standalone mode or due to a job reconfiguration). This commit process is completely automated by the framework, but only the connector knows how to seek back to the right position in the input stream to resume from that location. To correctly resume upon startup, the task can use the `SourceContext` passed into its `initialize()` method to access the offset data. In `initialize()`, we would add a bit more code to read the offset (if it exists) and seek to that position:

```
    stream = new FileInputStream(filename);
    Map<String, Object> offset = context.offsetStorageReader().offset(Collections.sing
letonMap(FILENAME_FIELD, filename));
    if (offset != null) {
        Long lastRecordedOffset = (Long) offset.get("position");
        if (lastRecordedOffset != null)
            seekToOffset(stream, lastRecordedOffset);
    }
```

Of course, you might need to read many keys for each of the input streams. The `OffsetStorageReader` interface also allows you to issue bulk reads to efficiently load all offsets, then apply them by seeking each input stream to the appropriate position.

Dynamic Input/Output Streams

Kafka Connect is intended to define bulk data copying jobs, such as copying an entire database rather than creating many jobs to copy each table individually. One consequence of this design is that the set of input or output streams for a connector can vary over time. Source connectors need to monitor the source system for changes, e.g. table additions/deletions in a database. When they pick up changes, they should notify the framework via the `ConnectorContext` object that reconfiguration is necessary. For example, in a `SourceConnector`:

```
    if (inputsChanged())
        this.context.requestTaskReconfiguration();
```

The framework will promptly request new configuration information and update the tasks, allowing them to gracefully commit their progress before reconfiguring them. Note that in the `SourceConnector` this monitoring is currently left up to the connector implementation. If an extra thread is required to perform this monitoring, the connector must allocate it itself. Ideally this code for monitoring changes would be isolated to the `Connector` and tasks would not need to worry about them. However, changes can also affect tasks, most commonly when one of their input streams is destroyed in the input system, e.g. if a table is dropped from a database. If the `Task` encounters the issue before the `Connector`, which will be common if the `Connector` needs to poll for changes, the `Task` will need to handle the subsequent error. Thankfully, this can usually be handled simply by catching and handling the appropriate exception. `SinkConnectors` usually only have to handle the addition of streams, which may translate to new entries in their outputs (e.g., a new database table). The framework manages any changes to the Kafka input, such as when the set of input topics changes because of a regex subscription. `SinkTasks` should expect new input streams, which may require creating new resources in the downstream system, such as a new table in a database. The trickiest situation to handle in these cases may be conflicts

between multiple `SinkTasks` seeing a new input stream for the first time and simultaneously trying to create the new resource. `SinkConnectors`, on the other hand, will generally require no special code for handling a dynamic set of streams.

Connect Configuration Validation

Kafka Connect allows you to validate connector configurations before submitting a connector to be executed and can provide feedback about errors and recommended values. To take advantage of this, connector developers need to provide an implementation of `config()` to expose the configuration definition to the framework. The following code in `FileStreamSourceConnector` defines the configuration and exposes it to the framework.

```
    private static final ConfigDef CONFIG_DEF = new ConfigDef()
        .define(FILE_CONFIG, Type.STRING, Importance.HIGH, "Source filename.")
        .define(TOPIC_CONFIG, Type.STRING, Importance.HIGH, "The topic to publish data
 to");

    public ConfigDef config() {
        return CONFIG_DEF;
    }
```

`ConfigDef` class is used for specifying the set of expected configurations. For each configuration, you can specify the name, the type, the default value, the documentation, the group information, the order in the group, the width of the configuration value and the name suitable for display in the UI. Plus, you can provide special validation logic used for single configuration validation by overriding the `Validator` class. Moreover, as there may be dependencies between configurations, for example, the valid values and visibility of a configuration may change according to the values of other configurations. To handle this, `ConfigDef` allows you to specify the dependents of a configuration and to provide an implementation of `Recommender` to get valid values and set visibility of a configuration given the current configuration values. Also, the `validate()` method in `Connector` provides a default validation implementation which returns a list of allowed configurations together with configuration errors and recommended values for each configuration. However, it does not use the recommended values for configuration validation. You may provide an override of the default implementation for customized configuration validation, which may use the recommended values.

Working with Schemas

The FileStream connectors are good examples because they are simple, but they also have trivially structured data -- each line is just a string. Almost all practical connectors will need schemas with more complex data formats. To create more complex data, you'll need to work

with the Kafka Connect `data` API. Most structured records will need to interact with two classes in addition to primitive types: `Schema` and `Struct`. The API documentation provides a complete reference, but here is a simple example creating a `Schema` and `Struct`:

```
Schema schema = SchemaBuilder.struct().name(NAME)
    .field("name", Schema.STRING_SCHEMA)
    .field("age", Schema.INT_SCHEMA)
    .field("admin", new SchemaBuilder.boolean().defaultValue(false).build())
    .build();

Struct struct = new Struct(schema)
    .put("name", "Barbara Liskov")
    .put("age", 75)
    .build();
```

If you are implementing a source connector, you'll need to decide when and how to create schemas. Where possible, you should avoid recomputing them as much as possible. For example, if your connector is guaranteed to have a fixed schema, create it statically and reuse a single instance. However, many connectors will have dynamic schemas. One simple example of this is a database connector. Considering even just a single table, the schema will not be predefined for the entire connector (as it varies from table to table). But it also may not be fixed for a single table over the lifetime of the connector since the user may execute an `ALTER TABLE` command. The connector must be able to detect these changes and react appropriately. Sink connectors are usually simpler because they are consuming data and therefore do not need to create schemas. However, they should take just as much care to validate that the schemas they receive have the expected format. When the schema does not match -- usually indicating the upstream producer is generating invalid data that cannot be correctly translated to the destination system -- sink connectors should throw an exception to indicate this error to the system.

Kafka Connect Administration

Kafka Connect's **REST layer** provides a set of APIs to enable administration of the cluster. This includes APIs to view the configuration of connectors and the status of their tasks, as well as to alter their current behavior (e.g. changing configuration and restarting tasks).

When a connector is first submitted to the cluster, the workers rebalance the full set of connectors in the cluster and their tasks so that each worker has approximately the same amount of work. This same rebalancing procedure is also used when connectors increase or decrease the number of tasks they require, or when a connector's configuration is changed.

You can use the REST API to view the current status of a connector and its tasks, including the id of the worker to which each was assigned. For example, querying the status of a file source (using `GET /connectors/file-source/status`) might produce output like the following:

```
{
  "name": "file-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.208:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "192.168.1.209:8083"
    }
  ]
}
```

Connectors and their tasks publish status updates to a shared topic (configured with `status.storage.topic`) which all workers in the cluster monitor. Because the workers consume this topic asynchronously, there is typically a (short) delay before a state change is visible through the status API. The following states are possible for a connector or one of its tasks:

- **UNASSIGNED:** The connector/task has not yet been assigned to a worker.
- **RUNNING:** The connector/task is running.
- **PAUSED:** The connector/task has been administratively paused.
- **FAILED:** The connector/task has failed (usually by raising an exception, which is reported in the status output).

In most cases, connector and task states will match, though they may be different for short periods of time when changes are occurring or if tasks have failed. For example, when a connector is first started, there may be a noticeable delay before the connector and its tasks have all transitioned to the RUNNING state. States will also diverge when tasks fail since Connect does not automatically restart failed tasks. To restart a connector/task manually, you can use the restart APIs listed above. Note that if you try to restart a task while a rebalance is taking place, Connect will return a 409 (Conflict) status code. You can retry after the rebalance completes, but it might not be necessary since rebalances effectively restart all the connectors and tasks in the cluster.

It's sometimes useful to temporarily stop the message processing of a connector. For example, if the remote system is undergoing maintenance, it would be preferable for source connectors to stop polling it for new data instead of filling logs with exception spam. For this use case, Connect offers a pause/resume API. While a source connector is paused,

Connect will stop polling it for additional records. While a sink connector is paused, Connect will stop pushing new messages to it. The pause state is persistent, so even if you restart the cluster, the connector will not begin message processing again until the task has been resumed. Note that there may be a delay before all of a connector's tasks have transitioned to the PAUSED state since it may take time for them to finish whatever processing they were in the middle of when being paused. Additionally, failed tasks will not transition to the PAUSED state until they have been restarted.