目　录

致谢

介绍

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！

阅读地址: Apache Kafka 官方文档中文版

源地址: Kafka 0.10.0 Documentation

Github: BeanMr/apache-kafka-documentation-cn

Gitbook: Apache Kafka Documentation CN

译者: @D2Feng

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，你只是准备支出一些翻译不当，也可以点击段落后‘+’直接评论。

JustDoIT，您的任何建议和尝试都值得尊重！

入门

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。

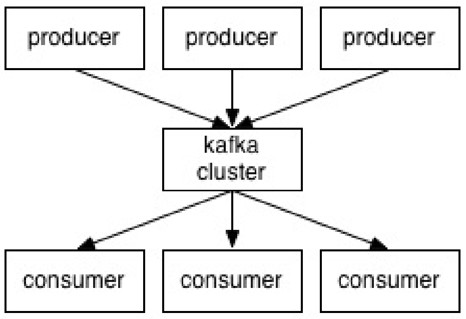
这是什么意思呢？

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：

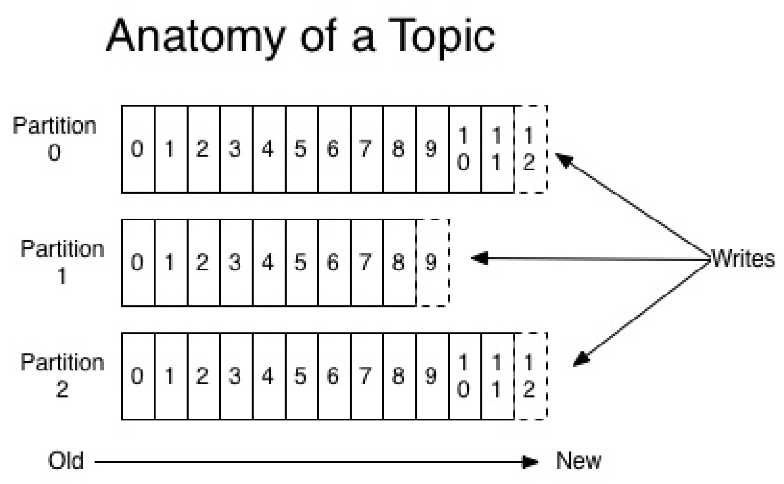


Clients(客户端)和Servers(服务器)通过一个简单的、高效的基于TCP的协议进行交互。官方为Kafka提供一个Java客户端，但更多其他语言的客户端可以在这里找到。

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。

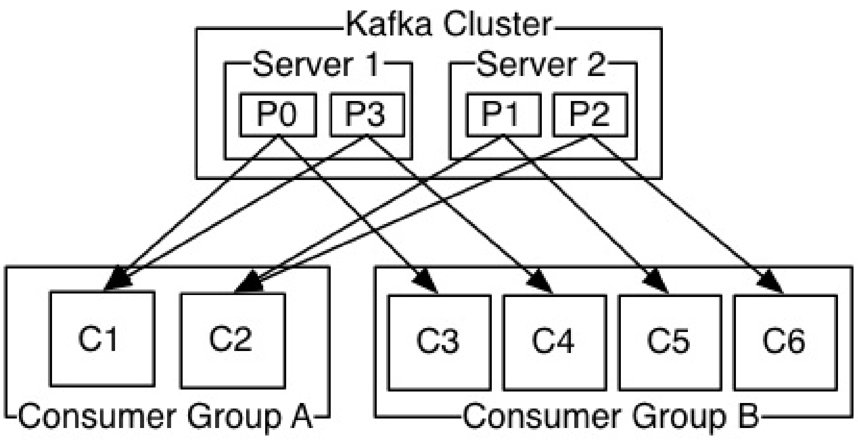
如果所有的消费者实例都采用不同的消费组，那么真个结构就是订阅模式，每一个消息将被广播给每一个消费者。

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。

应用场景

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他们

放在统一的地方（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。

快速入门

1.3 快速入门 Quick Start

本教程假设你从头开始，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的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)
...
```

现在开始启动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 1048576
(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 --partitions 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，除去手工创建topic以外，你也可以将你的brokers配置成当消息发布到一个不存在的topic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意思。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。但是为了更好的理解，我们将我们的集群扩展到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
```

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:

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”负责给定partition的所有读和写请求的响应。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-topic
...
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
```

先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
```

集群领导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-topic
Topic:my-replicated-topic     PartitionCount:1     ReplicationFactor:3     Configs:
    Topic: my-replicated-topic     Partition: 0     Leader: 2     Replicas: 1,2,0     Isr: 2,0
```

而且现在消息的消费仍然能正常进行，即使原来负责写的节点已经失效了。

```
> 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，数据也写回终端是默认的。但是你可能希望从一些其它的数据源或者导出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，可实现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-source.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，然后将它们输出成输出文件的一行行的数据。在启动的过程你讲看到一些日志消息，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 table 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-input.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 --topic
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.StringDeserializer
\
            --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

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。

软件生态

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。

升级

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。

API

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 the
```

```
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(ConsumerConfig 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stream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, Decoder<V>
valueDecoder);

  /**
   *  Create a list of message streams of type T for each topic, using the default decoder.
   *  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及元数据元素的迭代器
   */
  public <K,V> List<KafkaStream<K,V>>
    createMessageStreamsByFilter(TopicFilter topicFilter, int numStreams, Decoder<K> keyDecoder, Decoder<V> valueDecoder);

  /**
   *  Create a list of message streams for topics matching a wildcard, using the default 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 topicFilter, int numStreams);

  /**
   *  Create a list of message streams for topics matching a wildcard, using the default 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 topicFilter);

  /**
   *  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 request);

  /**
   *  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）。

配置

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

必备的配置信息如下：

- `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主机字符串
advertised.host.name	only used when ‘advertise ‘listeners’ are not set. ‘advertised.listeners’ in publish to ZooKeeper for IaaS environments, this m different from the interf broker binds. If this is use the value for ‘host.n Otherwise it will use the from java.net.InetAddress.getC 弃用:只有当’advertised.list 者’listeners’没有设置时使用。用’advertised.listeners’来 ZooKeeper供客户端使用的的Hos 中，这个配置可能与broker绑定的 没有设置那么它将使用已配置的’h 果’host.name’也没有配置它将通 java.net.InetAddress.getC 的值。
advertised.listeners	Listeners to publish to Z clients to use, if differ listeners above. In IaaS may need to be different to which the broker binds set, the value for ‘liste 发布到ZooKeeper供客户端使用的 broker监听的网络接口，例如 PLAINTEXT://192.168.71.31 security_protocol://host_

	面配置的’listeners’不同。在I 和broker绑定的接口不同。假如沪 用’listeners’的值。
advertised.port	DEPRECATED: only used whe ‘advertised.listeners’ or not set. Use ‘advertised. The port to publish to Zo to use. In IaaS environme to be different from the broker binds. If this is publish the same port tha to. 弃用：只有当’advertised.lis 者’listeners’没有配置的时候才 ‘advertised.listeners’代替 客户端使用的端口。在IaaS环境中 Broker绑定的不同。如果没有配置 口值。
auto.create.topics.enable	Enable auto creation of t 是否启动服务器topic自动创建
auto.leader.rebalance.enable	Enables auto leader balan thread checks and trigger required at regular inter 是否启动自动leader均衡，一个后 按需触发leader重选
background.threads	The number of threads to background processing tas 各种后台处理任务使用的线程数
broker.id	The broker id for this se unique broker id will be conflicts between zookeep id’s and user configured generated broker idsstart reserved.broker.max.id + 这个服务器上broker的id。如果 的broker id。为了避免基于Zoo id与用户配置的发生冲突，生成的 reserved.broker.max.id +
compression.type	Specify the final compres given topic. This configu standard compression code ‘snappy’, ‘lz4’). It addi ‘uncompressed’ which is e compression; and ‘produce retain the original compr the producer. 为特定的topic指定最终的压缩类 压缩编码(‘gzip’, ‘snappy’, 受’uncompressed’参数指明不进 时’producer’值表示保留生产者
delete.topic.enable	Enables delete topic. Del the admin tool will have config is turned off 是否启用删除topic。如果这个配

	tool删除topic将失效。
host.name	DEPRECATED: only used whe not set. Use ‘listeners’ of broker. If this is set to this address. If this bind to all interfaces 弃用：只有当’listeners’没有 个配置被设置，它将仅仅绑定到这 绑定到所有的接口上。
leader.imbalance.check.interval.seconds	The frequency with which rebalance check is trigge controller 控制器触发分区重分配检查的周期
leader.imbalance.per.broker.percentage	The ratio of leader imbal broker. The controller wo leader balance if it goes per broker. The value is percentage. 每个broker上的leader不均衡比 的leader不均衡比例超过此数值 均衡。这个数值按照百分比指定。 Replication机制Partition都 表AR第一个称为Preferred Rep Preferred Replica均匀分到b Partition的读写操作都由这个
listeners	Listener List - Comma-sep we will listen on and the Specify hostname as 0.0.0 interfaces. Leave hostnam default interface. Exampl listener lists: PLAINTEXT://myhost:9092,T PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093 监听接口列表 - 一个逗号分隔的 的协议监听这些地址。指定hostn 有的接口。留空hostname将绑定 的监听接口列表为： PLAINTEXT://myhost:9092,T PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093
log.dir	The directory in which th (supplemental for log.dir 日志文件存储的位置（与log.di
log.dirs	The directories in which kept. If not set, the val used 日志文件存储的位置列表，如果没
log.flush.interval.messages	The number of messages ac partition before messages disk 在消息被刷新到磁盘之前允许在一 的数量
	The maximum time in ms th

log.flush.interval.ms	topic is kept in memory b disk. If not set, the val log.flush.scheduler.inter 任意topic上的消息在刷新到硬盘 ms数字。如果没有设置将使用 log.flush.scheduler.inter
log.flush.offset.checkpoint.interval.ms	The frequency with which persistent record of the acts as the log recovery 更新最后一次flush的持久化消息
log.flush.scheduler.interval.ms	The frequency in ms that checks whether any log ne to disk 刷新器检查是否有log需要被刷新
log.retention.bytes	The maximum size of the l it 保留的日志最大的大小
log.retention.hours	The number of hours to ke before deleting it (in ho log.retention.ms property 在删除之前日志文件保存的最长小 log.retention.ms第三优先
log.retention.minutes	The number of minutes to before deleting it (in mi to log.retention.ms prope the value in log.retentio 在删除之前日志文件保存的最长分 log.retention.ms第二优先。如 log.retention.hours中的值将
log.retention.ms	The number of millisecond file before deleting it (If not set, the value in log.retention.minutes is 在删除之前日志文件保存的最长微 设置log.retention.minutes中
log.roll.hours	The maximum time before a rolled out (in hours), se log.roll.ms property 一个新日志段被推出前最长的时间 log.roll.ms第二优先
log.roll.jitter.hours	The maximum jitter to sub logRollTimeMillis (in hou log.roll.jitter.ms proper
log.roll.jitter.ms	The maximum jitter to sub logRollTimeMillis (in mil set, the value in log.rol used
log.roll.ms	The maximum time before a rolled out (in millisecon the value in log.roll.hou
log.segment.bytes	The maximum size of a sin

log.segment.delete.delay.ms	The amount of time to wai file from the filesystem
message.max.bytes	The maximum size of messa can receive
min.insync.replicas	define the minimum number needed to satisfy a produ acks=all (or -1)
num.io.threads	The number of io threads uses for carrying out net
num.network.threads	the number of network thr server uses for handling
num.recovery.threads.per.data.dir	The number of threads per be used for log recovery flushing at shutdown
num.replica.fetchers	Number of fetcher threads messages from a source br this value can increase t parallelism in the follow
offset.metadata.max.bytes	The maximum size for a me associated with an offset
offsets.commit.required.acks	The required acks before accepted. In general, the should not be overridden
offsets.commit.timeout.ms	Offset commit will be del replicas for the offsets commit or this timeout is similar to the producer r
offsets.load.buffer.size	Batch size for reading fr segments when loading off cache.
offsets.retention.check.interval.ms	Frequency at which to che offsets
offsets.retention.minutes	Log retention window in m topic
offsets.topic.compression.codec	Compression codec for the compression may be used t commits
offsets.topic.num.partitions	The number of partitions commit topic (should not deployment)
offsets.topic.replication.factor	The replication factor fo (set higher to ensure ava ensure that the effective of the offsets topic is t value, the number of aliv at least the replication of the first request for If not, either the offset

	will fail or it will get factor of min(alive broke replication factor)
offsets.topic.segment.bytes	The offsets topic segment kept relatively small in faster log compaction and
port	DEPRECATED: only used whe not set. Use ‘listeners’ to listen and accept conn
queued.max.requests	The number of queued requ blocking the network thre
quota.consumer.default	Any consumer distinguishe clientId\/consumer group if it fetches more bytes per-second
quota.producer.default	Any producer distinguishe get throttled if it produ this value per-second
replica.fetch.max.bytes	The number of bytes of me to fetch
replica.fetch.min.bytes	Minimum bytes expected fo response. If not enough b replicaMaxWaitTimeMs
replica.fetch.wait.max.ms	max wait time for each fe issued by follower replic should always be less tha replica.lag.time.max.ms a prevent frequent shrinkin throughput topics
replica.high.watermark.checkpoint.interval.ms	The frequency with which is saved out to disk
replica.lag.time.max.ms	If a follower hasn’t sent or hasn’t consumed up to offset for at least this will remove the follower
replica.socket.receive.buffer.bytes	The socket receive buffer requests
replica.socket.timeout.ms	The socket timeout for ne value should be at least replica.fetch.wait.max.ms
request.timeout.ms	The configuration control amount of time the client response of a request. If not received before the t client will resend the re or fail the request if re exhausted.
socket.receive.buffer.bytes	The SO_RCVBUF buffer of t sockets

socket.request.max.bytes	The maximum number of byt request
socket.send.buffer.bytes	The SO_SNDBUF buffer of t sockets
unclean.leader.election.enable	Indicates whether to enab the ISR set to be elected last resort, even though in data loss
zookeeper.connection.timeout.ms	The max time that the cli establish a connection to set, the value in zookeeper.session.timeout
zookeeper.session.timeout.ms	Zookeeper session timeout
zookeeper.set.acl	Set client to use secure
broker.id.generation.enable	Enable automatic broker i server? When enabled the for reserved.broker.max.i reviewed.
broker.rack	Rack of the broker. This rack aware replication as tolerance. Examples: ‘RAC
connections.max.idle.ms	Idle connections timeout: processor threads close t idle more than this
controlled.shutdown.enable	Enable controlled shutdow
controlled.shutdown.max.retries	Controlled shutdown can f reasons. This determines retries when such failure
controlled.shutdown.retry.backoff.ms	Before each retry, the sy recover from the state th previous failure (Control replica lag etc). This co amount of time to wait be
controller.socket.timeout.ms	The socket timeout for co channels
default.replication.factor	default replication facto automatically created top
fetch.purgatory.purge.interval.requests	The purge interval (in nu of the fetch request purg
group.max.session.timeout.ms	The maximum allowed sessi registered consumers. Lon consumers more time to pr between heartbeats at the time to detect failures.
group.min.session.timeout.ms	The minimum allowed sessi registered consumers. Sho leader to quicker failure

	cost of more frequent con which can overwhelm broke
inter.broker.protocol.version	Specify which version of protocol will be used. Th bumped after all brokers new version. Example of s are: 0.8.0, 0.8.1, 0.8.1. 0.8.2.1, 0.9.0.0, 0.9.0.1 for the full list.
log.cleaner.backoff.ms	The amount of time to sle no logs to clean
log.cleaner.dedupe.buffer.size	The total memory used for across all cleaner thread
log.cleaner.delete.retention.ms	How long are delete recor
log.cleaner.enable	Enable the log cleaner pr server? Should be enabled topics with a cleanup.pol including the internal of disabled those topics wil and continually grow in s
log.cleaner.io.buffer.load.factor	Log cleaner dedupe buffer percentage full the dedup become. A higher value wi to be cleaned at once but hash collisions
log.cleaner.io.buffer.size	The total memory used for buffers across all cleane
log.cleaner.io.max.bytes.per.second	The log cleaner will be t the sum of its read and w less than this value on a
log.cleaner.min.cleanable.ratio	The minimum ratio of dirt for a log to eligible for
log.cleaner.threads	The number of background log cleaning
log.cleanup.policy	The default cleanup polic beyond the retention wind “delete” or “compact”
log.index.interval.bytes	The interval with which w the offset index
log.index.size.max.bytes	The maximum size in bytes index
log.message.format.version	Specify the message forma broker will use to append logs. The value should be ApiVersion. Some examples 0.9.0.0, 0.10.0, check Ap details. By setting a par format version, the user all the existing messages

	smaller or equal than the Setting this value incorr consumers with older vers they will receive message that they don’t understan
log.message.timestamp.difference.max.ms	The maximum difference al timestamp when a broker r and the timestamp specifi If message.timestamp.type message will be rejected in timestamp exceeds this configuration is ignored message.timestamp.type=Lo
log.message.timestamp.type	Define whether the timest is message create time or The value should be eithe ‘LogAppendTime’
log.preallocate	Should pre allocate file segment? If you are using you probably need to set
log.retention.check.interval.ms	The frequency in millisec cleaner checks whether an for deletion
max.connections.per.ip	The maximum number of con from each ip address
max.connections.per.ip.overrides	Per-ip or hostname overri maximum number of connect
num.partitions	The default number of log topic
principal.builder.class	The fully qualified name implements the PrincipalB which is currently used t Principal for connections SecurityProtocol.
producer.purgatory.purge.interval.requests	The purge interval (in nu of the producer request p
replica.fetch.backoff.ms	The amount of time to sle partition error occurs.
reserved.broker.max.id	Max number that can be us
sasl.enabled.mechanisms	The list of SASL mechanis Kafka server. The list ma mechanism for which a sec available. Only GSSAPI is default.
sasl.kerberos.kinit.cmd	Kerberos kinit command pa
sasl.kerberos.min.time.before.relogin	Login thread sleep time b attempts.
	A list of rules for mappi

sasl.kerberos.principal.to.local.rules	names to short names (typ system usernames). The ru in order and the first ru principal name is used to name. Any later rules in ignored. By default, prin form {username}\/{hostnam mapped to {username}. For the format please see **sec and acls**.
sasl.kerberos.service.name	The Kerberos principal na as. This can be defined e JAAS config or in Kafka's
sasl.kerberos.ticket.renew.jitter	Percentage of random jitt renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep u window factor of time fro ticket's expiry has been time it will try to renew
sasl.mechanism.inter.broker.protocol	SASL mechanism used for i communication. Default is
security.inter.broker.protocol	Security protocol used to between brokers. Valid va PLAINTEXT, SSL, SASL_PLAI
ssl.cipher.suites	A list of cipher suites. combination of authentica MAC and key exchange algo negotiate the security se network connection using protocol.By default all t suites are supported.
ssl.client.auth	Configures kafka broker t authentication. The follo common: `ssl.client.auth=requi` required client authentic required. `ssl.client.auth=req` client authentication is requested , if this optio choose not to provide aut information about itself This means client authent needed.
ssl.enabled.protocols	The list of protocols ena connections.
ssl.key.password	The password of the priva store file. This is optio
ssl.keymanager.algorithm	The algorithm used by key for SSL connections. Defa key manager factory algor the Java Virtual Machine.
	The location of the key s

ssl.keystore.location	optional for client and c way authentication for cl
ssl.keystore.password	The store password for th file.This is optional for needed if ssl.keystore.lo configured.
ssl.keystore.type	The file format of the ke is optional for client.
ssl.protocol	The SSL protocol used to SSLContext. Default setti fine for most cases. Allo recent JVMs are TLS, TLSv SSL, SSLv2 and SSLv3 may older JVMs, but their usa due to known security vul
ssl.provider	The name of the security SSL connections. Default default security provider
ssl.trustmanager.algorithm	The algorithm used by tru for SSL connections. Defa trust manager factory alg for the Java Virtual Mach
ssl.truststore.location	The location of the trust
ssl.truststore.password	The password for the trus
ssl.truststore.type	The file format of the tr
authorizer.class.name	The authorizer class that authorization
metric.reporters	A list of classes to use reporters. Implementing t `MetricReporter` interface al classes that will be noti creation. The JmxReporter to register JMX statistic
metrics.num.samples	The number of samples mai metrics.
metrics.sample.window.ms	The window of time a metr computed over.
quota.window.num	The number of samples to
quota.window.size.seconds	The time span of each sam
ssl.endpoint.identification.algorithm	The endpoint identificati validate server hostname certificate.
zookeeper.sync.time.ms	How far a ZK follower can 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 --partitions 1
        --replication-factor 1 --config max.message.bytes=64000 --config flush.messages=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

delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms
flush.messages	None	log.flush.interval.messages
flush.ms	None	log.flush.interval.ms
index.interval.bytes	4096	log.index.interval.bytes
max.message.bytes	1,000,000	message.max.bytes

min.cleanable.dirty.ratio	0.5	log.cleaner.min.cleanable.ratio
min.insync.replicas	1	min.insync.replicas
retention.bytes	None	log.retention.bytes
retention.ms	7 days	log.retention.minutes

segment.bytes	1 GB	log.segment.bytes
segment.index.bytes	10 MB	log.index.size.max.bytes
segment.ms	7 days	log.roll.hours
segment.jitter.ms	0	log.roll.jitter.{ms,hours}

3.2 Producer Configs

Below is the configuration of the Java producer:

NameDescriptionTypeDefaultValid ValuesImportance	
bootstrap.servers	A list of host\/port pairs to establishing the initial conne the Kafka cluster. The client use of all servers irrespectiv servers are specified here for bootstrapping—this list only i initial hosts used to discover set of servers. This list shou the form `host1:port1,host2:port2,...` these servers are just used fo initial connection to discover cluster membership (which may dynamically), this list need n the full set of servers (you m more than one, though, in case is down).
key.serializer	Serializer class for key that the `Serializer` interface.
value.serializer	Serializer class for value tha implements the `Serializer` inter

acks	The number of acknowledgments producer requires the leader t received before considering a complete. This controls the du of records that are sent. The settings are common: `acks=0` If zero then the producer will no any acknowledgment from the se all. The record will be immedi added to the socket buffer and considered sent. No guarantee made that the server has recei record in this case, and the `r` configuration will not take ef the client won't generally kno failures). The offset given ba each record will always be set -1. `acks=1` This will mean the will write the record to its l but will respond without await acknowledgement from all follo this case should the leader fa immediately after acknowledgin record but before the follower replicated it then the record lost. `acks=all` This means the l will wait for the full set of replicas to acknowledge the re guarantees that the record wil lost as long as at least one i replica remains alive. This is strongest available guarantee.
buffer.memory	The total bytes of memory the can use to buffer records wait sent to the server. If records faster than they can be delive server the producer will block for `max.block.ms` after which it throw an exception.This settin correspond roughly to the tota the producer will use, but is bound since not all memory the uses is used for buffering. So additional memory will be used compression (if compression is as well as for maintaining in- requests.
compression.type	The compression type for all d generated by the producer. The is none (i.e. no compression). values are `none` , `gzip` , `snapp` `lz4` . Compression is of full b data, so the efficacy of batch also impact the compression ra batching means better compress
	Setting a value greater than z cause the client to resend any

retries	whose send fails with a potent transient error. Note that thi no different than if the clien the record upon receiving the Allowing retries without setting `max.in.flight.requests.per.c` 1 will potentially change the of records because if two batc sent to a single partition, an first fails and is retried but second succeeds, then the reco second batch may appear first.
ssl.key.password	The password of the private ke key store file. This is option client.
ssl.keystore.location	The location of the key store is optional for client and can for two-way authentication for
ssl.keystore.password	The store password for the key file.This is optional for clie only needed if ssl.keystore.lo configured.
ssl.truststore.location	The location of the trust stor
ssl.truststore.password	The password for the trust sto
batch.size	The producer will attempt to b records together into fewer re whenever multiple records are to the same partition. This he performance on both the client server. This configuration con default batch size in bytes.No will be made to batch records than this size.Requests sent t will contain multiple batches, each partition with data avail sent.A small batch size will m batching less common and may r throughput (a batch size of ze disable batching entirely). A batch size may use memory a bi wastefully as we will always a buffer of the specified batch anticipation of additional rec
client.id	An id string to pass to the se making requests. The purpose o to be able to track the source requests beyond just ip\/port allowing a logical application be included in server-side req logging.
connections.max.idle.ms	Close idle connections after t of milliseconds specified by t config.

linger.ms	The producer groups together a that arrive in between request transmissions into a single ba request. Normally this occurs load when records arrive faste they can be sent out. However circumstances the client may w reduce the number of requests moderate load. This setting ac this by adding a small amount artificial delay—that is, rath immediately sending out a reco producer will wait for up to t delay to allow other records t so that the sends can be batch together. This can be thought analogous to Nagle's algorithm This setting gives the upper b the delay for batching: once w `batch.size` worth of records fo partition it will be sent imme regardless of this setting, ho we have fewer than this many b accumulated for this partition 'linger' for the specified tim for more records to show up. T setting defaults to 0 (i.e. no Setting `linger.ms=5`, for examp have the effect of reducing th of requests sent but would add of latency to records sent in absense of load.
max.block.ms	The configuration controls how long `KafkaProducer.send()` and `KafkaProducer.partitionsFor()` w block.These methods can be blo either because the buffer is f metadata unavailable.Blocking user-supplied serializers or p will not be counted against th timeout.
max.request.size	The maximum size of a request This is also effectively a cap maximum record size. Note that server has its own cap on reco which may be different from th setting will limit the number batches the producer will send single request to avoid sendin requests.
partitioner.class	Partitioner class that impleme the `Partitioner` interface.
receive.buffer.bytes	The size of the TCP receive bu (SO_RCVBUF) to use when readin
	The configuration controls the amount of time the client will

request.timeout.ms	the response of a request. If response is not received befor timeout elapses the client wil the request if necessary or fa request if retries are exhaust
sasl.kerberos.service.name	The Kerberos principal name th runs as. This can be defined e Kafka's JAAS config or in Kafk config.
sasl.mechanism	SASL mechanism used for client connections. This may be any m for which a security provider available. GSSAPI is the defau mechanism.
security.protocol	Protocol used to communicate w brokers. Valid values are: PLA SSL, SASL_PLAINTEXT, SASL_SSL.
send.buffer.bytes	The size of the TCP send buffe (SO_SNDBUF) to use when sendin
ssl.enabled.protocols	The list of protocols enabled connections.
ssl.keystore.type	The file format of the key sto This is optional for client.
ssl.protocol	The SSL protocol used to gener SSLContext. Default setting is which is fine for most cases. values in recent JVMs are TLS, and TLSv1.2. SSL, SSLv2 and SS supported in older JVMs, but t is discouraged due to known se vulnerabilities.
ssl.provider	The name of the security provi for SSL connections. Default v the default security provider JVM.
ssl.truststore.type	The file format of the trust s
timeout.ms	The configuration controls the amount of time the server will acknowledgments from followers the acknowledgment requirement producer has specified with th `acks` configuration. If the rec number of acknowledgments are when the timeout elapses an er be returned. This timeout is m the server side and does not i network latency of the request
	When our memory buffer is exha must either stop accepting new (block) or throw errors. By de setting is false and the produ

block.on.buffer.full	no longer throw a BufferExhaus but instead will use the `max.bl` value to block, after which it throw a TimeoutException. Sett property to true will set the `max.block.ms` to Long.MAX_VALUE *this property is set to true,* _ `metadata.fetch.timeout.ms` is not honored._This parameter is dep and will be removed in a futur Parameter `max.block.ms` should b instead.
interceptor.classes	A list of classes to use as interceptors. Implementing the `ProducerInterceptor` interface al to intercept (and possibly mut records received by the produc they are published to the Kafk By default, there are no inter
max.in.flight.requests.per.connection	The maximum number of unacknow requests the client will send single connection before block that if this setting is set to greater than 1 and there are f sends, there is a risk of mess ordering due to retries (i.e., retries are enabled).
metadata.fetch.timeout.ms	The first time data is sent to we must fetch metadata about t to know which servers host the partitions. This fetch to succ throwing an exception back to client.
metadata.max.age.ms	The period of time in millisec which we force a refresh of me even if we haven't seen any pa leadership changes to proactiv discover any new brokers or pa
metric.reporters	A list of classes to use as me reporters. Implementing the `MetricReporter` interface allows in classes that will be notifi metric creation. The JmxReport always included to register JM statistics.
metrics.num.samples	The number of samples maintain compute metrics.
metrics.sample.window.ms	The window of time a metrics s computed over.
reconnect.backoff.ms	The amount of time to wait bef attempting to reconnect to a g This avoids repeatedly connect host in a tight loop. This bac applies to all requests sent b

	consumer to the broker.
retry.backoff.ms	The amount of time to wait bef attempting to retry a failed r a given topic partition. This repeatedly sending requests in loop under some failure scenar
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time betwee attempts.
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter ad renewal time.
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until specified window factor of tim last refresh to ticket’s expir reached, at which time it will renew the ticket.
ssl.cipher.suites	A list of cipher suites. This combination of authentication, encryption, MAC and key exchan algorithm used to negotiate th settings for a network connect TLS or SSL network protocol.By all the available cipher suite supported.
ssl.endpoint.identification.algorithm	The endpoint identification al validate server hostname using certificate.
ssl.keymanager.algorithm	The algorithm used by key mana factory for SSL connections. D value is the key manager facto algorithm configured for the J Virtual Machine.
ssl.trustmanager.algorithm	The algorithm used by trust ma factory for SSL connections. D value is the trust manager fac algorithm configured for the J 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 consumer processes to which this By setting the same group id mult indicate that they are all part c consumer group.
zookeeper.connect		Specifies the ZooKeeper connectic form `hostname:port` where host and and port of a ZooKeeper server. T through other ZooKeeper nodes whe machine is down you can also spec in the form `hostname1:port1,hostname2:port2,hostname` may also have a ZooKeeper chroot it's ZooKeeper connection string data under some path in the globa namespace. If so the consumer shc chroot path in its connection str to give a chroot path of `/chroot/p` the connection string as `hostname1:port1,hostname2:port2,hostna`
consumer.id	null	Generated automatically if not se
socket.timeout.ms	30 * 1000	The socket timeout for network re timeout set will be max.fetch.wai socket.timeout.ms.
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for net
fetch.message.max.bytes	1024 * 1024	The number of bytes of messages t for each topic-partition in each These bytes will be read into mem partition, so this helps control the consumer. The fetch request s least as large as the maximum mes server allows or else it is possi producer to send messages larger can fetch.
num.consumer.fetchers	1	The number fetcher threads used t
auto.commit.enable	true	If true, periodically commit to z offset of messages already fetche This committed offset will be use fails as the position from which will begin.
auto.commit.interval.ms	60 * 1000	The frequency in ms that the cons committed to zookeeper.
		Max number of message chunks buff

queued.max.message.chunks	2	consumption. Each chunk can be up fetch.message.max.bytes.
rebalance.max.retries	4	When a new consumer joins a consu of consumers attempt to “rebalanc assign partitions to each consume consumers changes while this assi place the rebalance will fail anc setting controls the maximum numb before giving up.
fetch.min.bytes	1	The minimum amount of data the se for a fetch request. If insuffici available the request will wait f to accumulate before answering th
fetch.wait.max.ms	100	The maximum amount of time the se before answering the fetch reques sufficient data to immediately sa fetch.min.bytes
rebalance.backoff.ms	2000	Backoff time between retries duri not set explicitly, the value in zookeeper.sync.time.ms is used.
refresh.leader.backoff.ms	200	Backoff time to wait before tryir leader of a partition that has ju leader.
auto.offset.reset	largest	What to do when there is no initi ZooKeeper or if an offset is out

* 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 initial connection to the Kafk will make use of all servers i servers are specified here for only impacts the initial hosts full set of servers. This list form `host1:port1,host2:port2,...`. S just used for the initial conn full cluster membership (which dynamically), this list need n of servers (you may want more case a server is down).
key.deserializer	Deserializer class for key tha `Deserializer` interface.
value.deserializer	Deserializer class for value t `Deserializer` interface.
fetch.min.bytes	The minimum amount of data the for a fetch request. If insuff the request will wait for that before answering the request. byte means that fetch requests a single byte of data is avail request times out waiting for this to something greater than to wait for larger amounts of can improve server throughput some additional latency.
group.id	A unique string that identifie this consumer belongs to. This the consumer uses either the g functionality by using `subscribe` based offset management strate
heartbeat.interval.ms	The expected time between hear coordinator when using Kafka's facilities. Heartbeats are use consumer's session stays activ rebalancing when new consumers group. The value must be set l than `session.timeout.ms`, but typi higher than 1\/3 of that value even lower to control the expe rebalances.
max.partition.fetch.bytes	The maximum amount of data per will return. The maximum total request will be `#partitions * ma` This size must be at least as message size the server allows

	for the producer to send messa consumer can fetch. If that ha get stuck trying to fetch a la partition.
session.timeout.ms	The timeout used to detect fai group management facilities. W heartbeat is not received with the broker will mark the consu rebalance the group. Since hea when poll() is invoked, a high allows more time for message p consumer's poll loop at the co detect hard failures. See also another option to control the poll loop. Note that the value allowable range as configured configuration by `group.min.session.timeout.ms` and
ssl.key.password	The password of the private ke This is optional for client.
ssl.keystore.location	The location of the key store for client and can be used for for client.
ssl.keystore.password	The store password for the key optional for client and only n ssl.keystore.location is confi
ssl.truststore.location	The location of the trust stor
ssl.truststore.password	The password for the trust sto
auto.offset.reset	What to do when there is no in if the current offset does not server (e.g. because that data deleted):earliest: automatical the earliest offsetlatest: aut offset to the latest offsetnon the consumer if no previous of consumer's groupanything else: consumer.
connections.max.idle.ms	Close idle connections after t milliseconds specified by this
enable.auto.commit	If true the consumer's offset committed in the background.
exclude.internal.topics	Whether records from internal should be exposed to the consu the only way to receive record is subscribing to it.
max.poll.records	The maximum number of records call to poll().
partition.assignment.strategy	The class name of the partitio that the client will use to di ownership amongst consumer ins

	management is used
receive.buffer.bytes	The size of the TCP receive bu when reading data.
request.timeout.ms	The configuration controls the the client will wait for the r the response is not received b elapses the client will resend necessary or fail the request exhausted.
sasl.kerberos.service.name	The Kerberos principal name th can be defined either in Kafka Kafka’s config.
sasl.mechanism	SASL mechanism used for client be any mechanism for which a s available. GSSAPI is the defau
security.protocol	Protocol used to communicate w values are: PLAINTEXT, SSL, SA
send.buffer.bytes	The size of the TCP send buffe when sending data.
ssl.enabled.protocols	The list of protocols enabled
ssl.keystore.type	The file format of the key sto optional for client.
ssl.protocol	The SSL protocol used to gener Default setting is TLS, which Allowed values in recent JVMs TLSv1.2. SSL, SSLv2 and SSLv3 older JVMs, but their usage is known security vulnerabilities
ssl.provider	The name of the security provi connections. Default value is provider of the JVM.
ssl.truststore.type	The file format of the trust s
auto.commit.interval.ms	The frequency in milliseconds offsets are auto-committed to `enable.auto.commit` is set to `tr`
check.crcs	Automatically check the CRC32 This ensures no on-the-wire or the messages occurred. This ch so it may be disabled in cases performance.
client.id	An id string to pass to the se requests. The purpose of this the source of requests beyond allowing a logical application server-side request logging.
fetch.max.wait.ms	The maximum amount of time the before answering the fetch req sufficient data to immediately

	given by fetch.min.bytes.
interceptor.classes	A list of classes to use as in the `ConsumerInterceptor` interface (and possibly mutate) records consumer. By default, there ar
metadata.max.age.ms	The period of time in millisec force a refresh of metadata ev any partition leadership chang discover any new brokers or pa
metric.reporters	A list of classes to use as me Implementing the `MetricReporter` plugging in classes that will metric creation. The JmxReport register JMX statistics.
metrics.num.samples	The number of samples maintain
metrics.sample.window.ms	The window of time a metrics s
reconnect.backoff.ms	The amount of time to wait bef reconnect to a given host. Thi connecting to a host in a tigh applies to all requests sent b broker.
retry.backoff.ms	The amount of time to wait bef a failed request to a given to avoids repeatedly sending requ under some failure scenarios.
sasl.kerberos.kinit.cmd	Kerberos kinit command path.
sasl.kerberos.min.time.before.relogin	Login thread sleep time betwee
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter ad
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until factor of time from last refre has been reached, at which tim the ticket.
ssl.cipher.suites	A list of cipher suites. This of authentication, encryption, algorithm used to negotiate th a network connection using TLS protocol.By default all the av are supported.
ssl.endpoint.identification.algorithm	The endpoint identification al server hostname using server c
ssl.keymanager.algorithm	The algorithm used by key mana connections. Default value is algorithm configured for the J
ssl.trustmanager.algorithm	The algorithm used by trust ma connections. Default value is factory algorithm configured f 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
group.id	A unique string that identifies the Connect cluster group this worker belongs to.
internal.key.converter	Converter class for internal key Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.
internal.value.converter	Converter class for offset value Connect data that implements the `Converter` interface. Used for converting data like offsets and configs.
key.converter	Converter class for key Connect data that implements the `Converter` interface.
offset.storage.topic	kafka topic to store connector offsets in
status.storage.topic	kafka topic to track connector and task status
value.converter	Converter class for value Connect data that implements the `Converter` interface.
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).
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.
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.
session.timeout.ms	The timeout used to detect failures when using Kafka’s group management facilities.
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.
connections.max.idle.ms	Close idle connections after the number of milliseconds

	specified by this config.
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.
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.
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.
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.
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.
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.
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 of new metric creation. The JmxReporter is always 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.
offset.flush.interval.ms	Interval at which to try committing offsets for tasks.
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.
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.
rest.advertised.host.name	If this is set, this is the hostname that will be given out to other workers to connect to.
rest.advertised.port	If this is set, this is the port that will be given out to other workers to connect to.
rest.host.name	Hostname for the REST API. If this is set, it will only bind to this interface.
rest.port	Port for the REST API to listen on.
	The amount of time to wait before attempting to retry a failed request to a given

retry.backoff.ms	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’s expiry 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.
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.
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.

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
	Partition grouper class that	

partition.grouper	implements the `PartitionGrouper` interface.	class
replication.factor	The replication factor for change log topics and repartition topics created by the stream processing application.	int
state.dir	Directory location for state store.	string
timestamp.extractor	Timestamp extractor class that implements the `TimestampExtractor` interface.	class
value.serde	Serializer \/ deserializer class for value that implements the `Serde` interface.	class
buffered.records.per.partition	The maximum number of records to buffer per partition.	int
commit.interval.ms	The frequency with which to save the position of the processor.	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
num.standby.replicas	The number of standby replicas for each task.	int
num.stream.threads	The number of threads to execute stream processing.	int
poll.ms	The amount of time in milliseconds to block waiting for input.	long
state.cleanup.delay.ms	The amount of time in milliseconds to wait before deleting state when a partition has migrated.	long

设计

4. 设计

4.1 设计初衷

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.

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。

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

它必须有很高的吞吐量来支撑像实时日志合并这类高容量的事件流。

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

它必须能很好的处理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。

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

同时它也需要可以处理低延迟的分发需求来支撑与传统消息机制类似的应用场景。

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

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。这些初衷影响着我们的分区和消费者模型。

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.

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。

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.

为了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 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.

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的和网络一样快。

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!**

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!

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.

为了填补这个性能的差异，现在操作系统越来越激进的使用它们的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在这块统一的缓存。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。

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

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. The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
2. 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。
3. Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.
4. 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。

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.

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。

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.

这使人想到一个非常简单的设计：相对于尽可能多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。所有的数据被立即写入一个不需要flush磁盘操作的持久化的文件系统的log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。

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

这种基于页缓存的设计可以参见在这篇关于**Varnish**的论文

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.

Implementation

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>> producerData);

  /* 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 earliest offset available.
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);
}
```

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> topicCountMap);

  /**
   * 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 independent 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.

```
MessageSetSend (fetch result)

total length     : 4 bytes
error code       : 2 bytes
message 1        : x bytes
...
message n        : x bytes
```

```
MultiMessageSetSend (multiFetch result)

total length       : 4 bytes
error code         : 2 bytes
messageSetSend 1
...
messageSetSend n
```

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":
{...:...},"pattern":...,"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 ((persistent 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 node)
```

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 ownership)
```

When rebalancing is triggered at one consumer, rebalancing should be triggered in other consumers within the same group about the same time.

Operations

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
      --consumer.config consumer-1.properties --consumer.config consumer-2.properties
      --producer.config producer.properties --whitelist my-topic
```

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-consumer-group

GROUP                          TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
test-consumer-group            test-foo                       0          1               3               2               test-consumer-group_postamac.local-1456198719410-29ccd54f-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-file 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":"foo2","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 custom-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 increase-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 --entity-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=MessagesInPe
Byte in rate	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSe
Request rate	kafka.network:type=RequestMetrics,name=RequestsPerSec,
Byte out rate	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerS
Log flush rate and time	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeM
# of under replicated partitions (\	ISR\
Is controller active on broker	kafka.controller:type=KafkaController,name=ActiveContr
Leader election rate	kafka.controller:type=ControllerStats,name=LeaderElect
Unclean leader election rate	kafka.controller:type=ControllerStats,name=UncleanLead
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,cl
Lag in messages per follower replica	kafka.server:type=FetcherLagMetrics,name=ConsumerLag,c ([-.\w]+),topic=([-.\w]+),partition=([0-9]+)
Requests waiting in the producer purgatory	kafka.server:type=ProducerRequestPurgatory,name=Purgat
Requests waiting in the fetch purgatory	kafka.server:type=FetchRequestPurgatory,name=Purgatory
Request total time	kafka.network:type=RequestMetrics,name=TotalTimeMs,req
Time the request waiting in the request queue	kafka.network:type=RequestMetrics,name=QueueTimeMs,req
Time the request being processed at the leader	kafka.network:type=RequestMetrics,name=LocalTimeMs,req
Time the request waits for the follower	kafka.network:type=RequestMetrics,name=RemoteTimeMs,re
Time to send the	kafka.network:type=RequestMetrics,name=ResponseSendTim

response	kafka.network:type=RequestMetrics,name=ResponseSendTim
Number of messages the consumer lags behind the producer by	kafka.consumer:type=ConsumerFetcherManager,name=MaxLag
The average fraction of time the network processors are idle	kafka.network:type=SocketServer,name=NetworkProcessorA
The average fraction of time the request handler threads are idle	kafka.server:type=KafkaRequestHandlerPool,name=Request
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	kafka.producer:type=producer-metrics,client-id=([-.\w]+)

	used).	
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]+)
request-latency-max	The maximum request latency in ms.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
record-send-rate	The average number of records sent per second.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
	The average	

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	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]+)
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	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)

outgoing-byte-rate	sent per second for a node.	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]+)
request-latency-avg	The average request latency in ms for a node.	kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
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]+)
	The average	

	retried record sends for a topic.	([-.\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

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

```
   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.
```

Security

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-signed
```

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 -keyalg 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-signed
```

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 -> EndPoint(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@test.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 --producer.config client-ssl.properties
        kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic test --new-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:**
   a. 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.
   b. 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).
   c. 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

a. 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.

a. 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}"
```

a. **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**

a. 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";
};
```

b. 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.

c. 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
```

d. Make sure the keytabs configured in the JAAS file are readable by the operating system user who is starting kafka broker.

e. 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, principal 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 (seehere 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

a. 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.

a. Pass the JAAS config file location as JVM parameter to each Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

b. 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 described**here**. 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:string](regexp)s\/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
—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 198.51.100.3 --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 --remove --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 section**7.6.2**.

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=localhost: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**

Kafka Connect

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:

```java
@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:

```java
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>` :

```java
@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("filename", 
filename);
                Map<String, Object> sourceOffset = Collections.singletonMap("position", 
streamOffset);
                records.add(new SourceRecord(sourcePartition, sourceOffset, topic, 
Schema.STRING_SCHEMA, line));
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling stop. Allow to 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.singletonMap(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.

Kafka Streams

9. Kafka Streams

9.1 Overview

Kafka Streams is a client library for processing and analyzing data stored in Kafka and either write the resulting data back to Kafka or send the final output to an external system. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state. Kafka Streams has a **low barrier to entry**: You can quickly write and run a small-scale proof-of-concept on a single machine; and you only need to run additional instances of your application on multiple machines to scale up to high-volume production workloads. Kafka Streams transparently handles the load balancing of multiple instances of the same application by leveraging Kafka’s parallelism model.

Some highlights of Kafka Streams:

- Designed as a **simple and lightweight client library**, which can be easily embedded in any Java application and integrated with any existing packaging, deployment and operational tools that users have for their streaming applications.
- Has **no external dependencies on systems other than Apache Kafka itself** as the internal messaging layer; notably, it uses Kafka’s partitioning model to horizontally scale processing while maintaining strong ordering guarantees.
- Supports **fault-tolerant local state**, which enables very fast and efficient stateful operations like joins and windowed aggregations.
- Employs **one-record-at-a-time processing** to achieve low processing latency, and supports **event-time based windowing operations**.
- Offers necessary stream processing primitives, along with a **high-level Streams DSL** and a **low-level Processor API**.

9.2 Developer Guide

There is a **quickstart** example that provides how to run a stream processing program coded in the Kafka Streams library. This section focuses on how to write, configure, and execute a Kafka Streams application.

Core Concepts

We first summarize the key concepts of Kafka Streams.

Stream Processing Topology

- A **stream**is the most important abstraction provided by Kafka Streams: it represents an unbounded, continuously updating data set. A stream is an ordered, replayable, and fault-tolerant sequence of immutable data records, where a **data record** is defined as a key-value pair.
- A stream processing application written in Kafka Streams defines its computational logic through one or more **processor topologies**, where a processor topology is a graph of stream processors (nodes) that are connected by streams (edges).
- A **stream processor** is a node in the processor topology; it represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology, applying its operation to it, and may subsequently producing one or more output records to its downstream processors.

Kafka Streams offers two ways to define the stream processing topology: the **Kafka Streams DSL** provides the most common data transformation operations such as `map` and `filter` ; the lower-level **Processor API** allows developers define and connect custom processors as well as to interact with **state stores**.

Time

A critical aspect in stream processing is the notion of **time**, and how it is modeled and integrated. For example, some operations such as **windowing** are defined based on time boundaries.

Common notions of time in streams are:

- **Event time** - The point in time when an event or data record occurred, i.e. was originally created “at the source”.
- **Processing time** - The point in time when the event or data record happens to be processed by the stream processing application, i.e. when the record is being consumed. The processing time may be milliseconds, hours, or days etc. later than the original event time.

Kafka Streams assigns a **timestamp** to every data record via the `TimestampExtractor` interface. Concrete implementations of this interface may retrieve or compute timestamps based on the actual contents of data records such as an embedded timestamp field to provide event-time semantics, or use any other approach such as returning the current wall-clock time at the time of processing, thereby yielding processing-time semantics to stream processing applications. Developers can thus enforce different notions of time depending on their business needs. For example, per-record timestamps describe the progress of a stream with regards to time (although records may be out-of-order within the stream) and are leveraged by time-dependent operations such as joins.

States

Some stream processing applications don’t require state, which means the processing of a message is independent from the processing of all other messages. However, being able to maintain state opens up many possibilities for sophisticated stream processing applications: you can join input streams, or group and aggregate data records. Many such stateful operators are provided by the **Kafka Streams DSL**.

Kafka Streams provides so-called **state stores**, which can be used by stream processing applications to store and query data. This is an important capability when implementing stateful operations. Every task in Kafka Streams embeds one or more state stores that can be accessed via APIs to store and query data required for processing. These state stores can either be a persistent key-value store, an in-memory hashmap, or another convenient data structure. Kafka Streams offers fault-tolerance and automatic recovery for local state stores.

As we have mentioned above, the computational logic of a Kafka Streams application is defined as a **processor topology**. Currently Kafka Streams provides two sets of APIs to define the processor topology, which will be described in the subsequent sections.

Low-Level Processor API

Processor

Developers can define their customized processing logic by implementing the `Processor` interface, which provides `process` and `punctuate` methods. The `process` method is performed on each of the received record; and the `punctuate` method is performed periodically based on elapsed time. In addition, the processor can maintain the current `ProcessorContext` instance variable initialized in the `init` method, and use the context to schedule the punctuation period (`context().schedule`), to forward the modified \/ new key-value pair to downstream processors (`context().forward`), to commit the current processing progress (`context().commit`), etc.

```
public class MyProcessor extends Processor {
    private ProcessorContext context;
    private KeyValueStore kvStore;

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        this.context = context;
        this.context.schedule(1000);
        this.kvStore = (KeyValueStore) context.getStateStore("Counts");
    }

    @Override
    public void process(String dummy, String line) {
        String[] words = line.toLowerCase().split(" ");

        for (String word : words) {
            Integer oldValue = this.kvStore.get(word);

            if (oldValue == null) {
                this.kvStore.put(word, 1);
            } else {
                this.kvStore.put(word, oldValue + 1);
            }
        }
    }

    @Override
    public void punctuate(long timestamp) {
        KeyValueIterator iter = this.kvStore.all();

        while (iter.hasNext()) {
            KeyValue entry = iter.next();
            context.forward(entry.key, entry.value.toString());
        }

        iter.close();
        context.commit();
    }

    @Override
    public void close() {
        this.kvStore.close();
    }
};
```

In the above implementation, the following actions are performed:

- In the `init` method, schedule the punctuation every 1 second and retrieve the local state store by its name “Counts”.
- In the `process` method, upon each received record, split the value string into words, and update their counts into the state store (we will talk about this feature later in the section).
- In the `punctuate` method, iterate the local state store and send the aggregated counts to the downstream processor, and commit the current stream state.

Processor Topology

With the customized processors defined in the Processor API, developers can use the `TopologyBuilder` to build a processor topology by connecting these processors together:

```
    TopologyBuilder builder = new TopologyBuilder();

    builder.addSource("SOURCE", "src-topic")

        .addProcessor("PROCESS1", MyProcessor1::new /* the ProcessorSupplier that can generate MyProcessor1 */, "SOURCE")
        .addProcessor("PROCESS2", MyProcessor2::new /* the ProcessorSupplier that can generate MyProcessor2 */, "PROCESS1")
        .addProcessor("PROCESS3", MyProcessor3::new /* the ProcessorSupplier that can generate MyProcessor3 */, "PROCESS1")

        .addSink("SINK1", "sink-topic1", "PROCESS1")
        .addSink("SINK2", "sink-topic2", "PROCESS2")
        .addSink("SINK3", "sink-topic3", "PROCESS3");
```

There are several steps in the above code to build the topology, and here is a quick walk through:

- First of all a source node named “SOURCE” is added to the topology using the `addSource` method, with one Kafka topic “src-topic” fed to it.
- Three processor nodes are then added using the `addProcessor` method; here the first processor is a child of the “SOURCE” node, but is the parent of the other two processors.
- Finally three sink nodes are added to complete the topology using the `addSink` method, each piping from a different parent processor node and writing to a separate topic.

Local State Store

Note that the Processor API is not limited to only accessing the current records as they arrive, but can also maintain local state stores that keep recently arrived records to use in stateful processing operations such as aggregation or windowed joins. To take advantage of this local states, developers can use the `TopologyBuilder.addStateStore` method when building the processor topology to create the local state and associate it with the processor nodes that needs to access it; or they can connect a created local state store with the existing processor nodes through `TopologyBuilder.connectProcessorAndStateStores`.

```
    TopologyBuilder builder = new TopologyBuilder();

    builder.addSource("SOURCE", "src-topic")

        .addProcessor("PROCESS1", MyProcessor1::new, "SOURCE")
        // create the in-memory state store "COUNTS" associated with processor "PROCESS1"
.addStateStore(Stores.create("COUNTS").withStringKeys().withStringValues().inMemory().build(), "PROCESS1")
        .addProcessor("PROCESS2", MyProcessor3::new /* the ProcessorSupplier that can generate MyProcessor3 */, "PROCESS1")
        .addProcessor("PROCESS3", MyProcessor3::new /* the ProcessorSupplier that can generate MyProcessor3 */, "PROCESS1")

        // connect the state store "COUNTS" with processor "PROCESS2"
        .connectProcessorAndStateStores("PROCESS2", "COUNTS");

        .addSink("SINK1", "sink-topic1", "PROCESS1")
        .addSink("SINK2", "sink-topic2", "PROCESS2")
        .addSink("SINK3", "sink-topic3", "PROCESS3");
```

In the next section we present another way to build the processor topology: the Kafka Streams DSL.

High-Level Streams DSL

To build a processor topology using the Streams DSL, developers can apply the `KStreamBuilder` class, which is extended from the `TopologyBuilder`. A simple example is included with the source code for Kafka in the `streams/examples` package. The rest of this section will walk through some code to demonstrate the key steps in creating a topology using the Streams DSL, but we recommend developers to read the full example source codes for details.

Create Source Streams from Kafka

Either a **record stream** (defined as `KStream`) or a **changelog stream** (defined as `KTable`) can be created as a source stream from one or more Kafka topics (for `KTable` you can only create the source stream from a single topic).

```
KStreamBuilder builder = new KStreamBuilder();

KStream source1 = builder.stream("topic1", "topic2");
KTable source2 = builder.table("topic3");
```

Transform a stream

There is a list of transformation operations provided for `KStream` and `KTable` respectively. Each of these operations may generate either one or more `KStream` and `KTable` objects and can be translated into one or more connected processors into the underlying processor topology. All these transformation methods can be chained together to compose a complex processor topology. Since `KStream` and `KTable` are strongly typed, all these transformation operations are defined as generics functions where users could specify the input and output data types.

Among these transformations, `filter` , `map` , `mapValues` , etc, are stateless transformation operations and can be applied to both `KStream` and `KTable` , where users can usually pass a customized function to these functions as a parameter, such as `Predicate` for `filter` , `KeyValueMapper` for `map` , etc:

```
// written in Java 8+, using lambda expressions
KStream mapped = source1.mapValue(record -> record.get("category"));
```

Stateless transformations, by definition, do not depend on any state for processing, and hence implementation-wise they do not require a state store associated with the stream processor; Stateful transformations, on the other hand, require accessing an associated state for processing and producing outputs. For example, in `join` and `aggregate` operations, a windowing state is usually used to store all the received records within the defined window boundary so far. The operators can then access these accumulated records in the store and compute based on them.

```
// written in Java 8+, using lambda expressions
KTable, Long> counts = source1.aggregateByKey(
    () -> 0L,  // initial value
    (aggKey, value, aggregate) -> aggregate + 1L,   // aggregating value
```

```
        TimeWindows.of("counts",5000L).advanceBy(1000L), // intervals in milliseconds
    );

    KStream joined = source1.leftJoin(source2,
        (record1, record2) -> record1.get("user") + "-" + record2.get("region");
    );
```

Write streams back to Kafka

At the end of the processing, users can choose to (continuously) write the final resulted streams back to a Kafka topic through `KStream.to` and `KTable.to` .

```
    joined.to("topic4");
```

If your application needs to continue reading and processing the records after they have been materialized to a topic via `to` above, one option is to construct a new stream that reads from the output topic; Kafka Streams provides a convenience method called `through` :

```
    // equivalent to
    //
    // joined.to("topic4");
    // materialized = builder.stream("topic4");
    KStream materialized = joined.through("topic4");
```

Besides defining the topology, developers will also need to configure their applications in `StreamsConfig` before running it. A complete list of Kafka Streams configs can be found **here**.