Apache Kafka

A high-throughput distributed messaging system.

- download
- introduction
- uses
- documentation
- quickstart
- performance
- clients
- ecosystem
- security
- faq
- project
  - twitter
  - wiki
  - bugs
  - mailing lists
  - committers
  - powered by
  - papers & talks
- developers
  - code
  - projects
  - contributing
  - coding guide
  - protocol guide
  - unit tests

Kafka 0.10.0 Documentation

Prior releases: 0.7.x, 0.8.0, 0.8.1.X, 0.8.2.X, 0.9.0.X.

1. Getting Started

1.1 Introduction

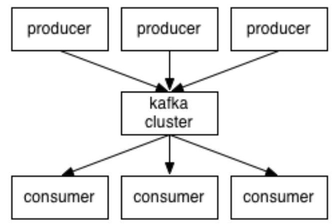
Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

What does all that mean?

First let's review some basic messaging terminology:

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

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

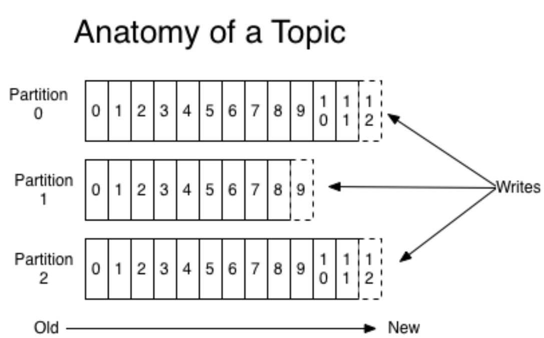


Communication between the clients and the servers is done with a simple, high-performance, language agnostic **TCP protocol**. We provide a Java client for Kafka, but clients are available in **many languages**.

Topics and Logs

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

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



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

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

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

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

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

Distribution

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

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

Producers

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

Consumers

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

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

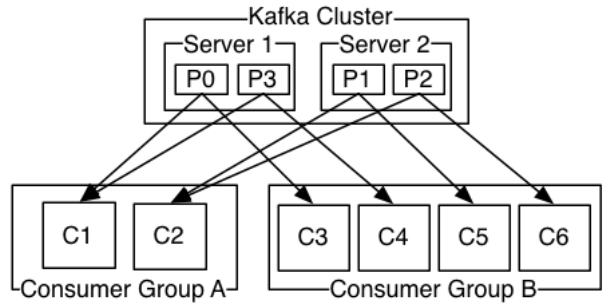
If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers.

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

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

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

A traditional queue retains messages in-order on the server, and if multiple consumers consume from the queue then the server hands out messages in the order they are stored. However, although the server hands out messages in order, the messages are delivered asynchronously to consumers, so they may arrive out of order on different consumers. This effectively means the ordering of the messages is lost in the presence of parallel consumption. Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is no parallelism in processing.



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

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

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

Guarantees

At a high-level Kafka gives the following guarantees:

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

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

1.2 Use Cases

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

Messaging

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

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

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

Website Activity Tracking

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

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

Metrics

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

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

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

Event Sourcing

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

Commit Log

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

1.3 Quick Start

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

Step 1: Download the code

Download the 0.10.0.0 release and un-tar it.

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

Step 2: Start the server

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

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.p
...
```

Now start the Kafka server:

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

Step 3: Create a topic

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

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-facto
```

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

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

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

Step 4: Send some messages

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

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

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

Step 5: Start a consumer

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

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

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

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

Step 6: Setting up a multi-broker cluster

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

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

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

Now edit these new files and set the following properties:

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

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

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

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

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

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

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factoı
```

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

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replica
Topic:my-replicated-topic       PartitionCount:1        ReplicationFactor:3
        Topic: my-replicated-topic      Partition: 0    Leader: 1       Replic
```

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

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-

up to the leader.

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

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

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

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

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

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

Now let's consume these messages:

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

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

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

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

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

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

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

Step 7: Use Kafka Connect to import/export data

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

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

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

```
> bin/connect-standalone.sh config/connect-standalone.properties config/conne
```

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

`test.txt`

and producing them to the topic

`connect-test`

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

`connect-test`

and write them to the file

`test.sink.txt`

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

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

Note that the data is being stored in the Kafka topic

`connect-test`

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

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

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

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

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

Step 8: Use Kafka Streams to process data

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

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

    // Ensure the words are available as record keys for the next aggregate o
```

```
    .map((key, value) -> new KeyValue<>(value, value))

    // Count the occurrences of each word (record key) and store the results i
    .countByKey("Counts")
```

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

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

```
> echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit"
```

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

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

```
> cat file-input.txt | bin/kafka-console-producer.sh --broker-list localhost:9
```

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

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

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

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

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

with the following output data being printed to the console:

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

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

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

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

1.4 Ecosystem

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

1.5 Upgrading From Previous Versions

Upgrading from 0.8.x or 0.9.x to 0.10.0.0

0.10.0.0 has **potential breaking changes** (please review before upgrading) and possible **performance impact following the upgrade**. By following the recommended rolling upgrade plan below, you guarantee no downtime and no performance impact during and following the upgrade.
Note: Because new protocols are introduced, it is important to upgrade your Kafka clusters before upgrading your clients.

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

For a rolling upgrade:

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

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

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

Potential performance impact following upgrade to 0.10.0.0

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

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

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

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

Note: If you have enabled compression on producers, you may notice reduced producer throughput and/or lower compression rate on the broker in some cases. When receiving compressed messages, 0.10.0 brokers avoid recompressing the messages, which in general reduces the latency and improves the throughput. In certain cases, however, this may reduce the batching size on the producer, which could lead to worse throughput. If this happens, users can tune linger.ms and batch.size of the producer for better throughput. In addition, the producer buffer used for compressing messages with snappy is smaller than the one used by the broker, which may have a negative impact on the compression ratio for the messages on disk. We intend to make this configurable in a future Kafka release.

Potential breaking changes in 0.10.0.0

- Starting from Kafka 0.10.0.0, the message format version in Kafka is represented as the Kafka version. For example, message format 0.9.0 refers to the highest message version supported by Kafka 0.9.0.
- Message format 0.10.0 has been introduced and it is used by default. It includes a timestamp field in the messages and relative offsets are used for compressed messages.
- ProduceRequest/Response v2 has been introduced and it is used by default to support message format 0.10.0
- FetchRequest/Response v2 has been introduced and it is used by default to support message format 0.10.0
- MessageFormatter interface was changed from `def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)` to `def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)`
- MessageReader interface was changed from `def readMessage():`

`KeyedMessage[Array[Byte], Array[Byte]]` to `def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]`

- MessageFormatter's package was changed from `kafka.tools` to `kafka.common`
- MessageReader's package was changed from `kafka.tools` to `kafka.common`
- MirrorMakerMessageHandler no longer exposes the `handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])` method as it was never called.
- The 0.7 KafkaMigrationTool is no longer packaged with Kafka. If you need to migrate from 0.7 to 0.10.0, please migrate to 0.8 first and then follow the documented upgrade process to upgrade from 0.8 to 0.10.0.
- The new consumer has standardized its APIs to accept `java.util.Collection` as the sequence type for method parameters. Existing code may have to be updated to work with the 0.10.0 client library.
- LZ4-compressed message handling was changed to use an interoperable framing specification (LZ4f v1.5.1). To maintain compatibility with old clients, this change only applies to Message format 0.10.0 and later. Clients that Produce/Fetch LZ4-compressed messages using v0/v1 (Message format 0.9.0) should continue to use the 0.9.0 framing implementation. Clients that use Produce/Fetch protocols v2 or later should use interoperable LZ4f framing. A list of interoperable LZ4 libraries is available at http://www.lz4.org/

Notable changes in 0.10.0.0

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

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

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

For a rolling upgrade:

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

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

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

Potential breaking changes in 0.9.0.0

- Java 1.6 is no longer supported.
- Scala 2.9 is no longer supported.
- Broker IDs above 1000 are now reserved by default to automatically assigned broker IDs. If your cluster has existing broker IDs above that threshold make sure to increase the reserved.broker.max.id broker configuration property accordingly.
- Configuration parameter replica.lag.max.messages was removed. Partition leaders will no longer consider the number of lagging messages when deciding which replicas are in sync.
- Configuration parameter replica.lag.time.max.ms now refers not just to the time passed since last fetch request from replica, but also to time since the replica last caught up. Replicas that are still fetching messages from leaders but did not catch up to the latest messages in replica.lag.time.max.ms will be considered out of sync.
- Compacted topics no longer accept messages without key and an exception is thrown by the producer if this is attempted. In 0.8.x, a message without key would cause the log compaction thread to subsequently complain and quit (and stop compacting all compacted topics).
- MirrorMaker no longer supports multiple target clusters. As a result it will only accept a single --consumer.config parameter. To mirror multiple source clusters, you will need at least one MirrorMaker instance per source cluster, each with its own consumer configuration.
- Tools packaged under *org.apache.kafka.clients.tools.** have been moved to *org.apache.kafka.tools.**. All included scripts will still function as usual, only custom code directly importing these classes will be affected.
- The default Kafka JVM performance options (KAFKA_JVM_PERFORMANCE_OPTS) have been changed in kafka-run-class.sh.
- The kafka-topics.sh script (kafka.admin.TopicCommand) now exits with non-zero exit code on failure.
- The kafka-topics.sh script (kafka.admin.TopicCommand) will now print a warning when topic names risk metric collisions due to the use of a '.' or '_' in the topic name, and error in the case of an actual collision.
- The kafka-console-producer.sh script (kafka.tools.ConsoleProducer) will use the new producer instead of the old producer be default, and users have to specify 'old-producer' to use the old producer.
- By default all command line tools will print all logging messages to stderr instead of stdout.

Notable changes in 0.9.0.1

- The new broker id generation feature can be disabled by setting broker.id.generation.enable to false.
- Configuration parameter log.cleaner.enable is now true by default. This means topics with a cleanup.policy=compact will now be compacted by default, and 128 MB of heap will be allocated to the cleaner process via log.cleaner.dedupe.buffer.size. You may want to review log.cleaner.dedupe.buffer.size and the other log.cleaner configuration values based on your usage of compacted topics.
- Default value of configuration parameter fetch.min.bytes for the new consumer is now 1 by default.

Deprecations in 0.9.0.0

- Altering topic configuration from the kafka-topics.sh script (kafka.admin.TopicCommand) has been deprecated. Going forward, please use the kafka-configs.sh script (kafka.admin.ConfigCommand) for this functionality.

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

Upgrading from 0.8.1 to 0.8.2

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

Upgrading from 0.8.0 to 0.8.1

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

Upgrading from 0.7

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

2. API

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

2.1 Producer API

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

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

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

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

2.2 Consumer API

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

2.2.1 Old High Level Consumer API

```
class Consumer {
  /**
   *  Create a ConsumerConnector
   *
   *  @param config  at the minimum, need to specify the groupid of the consur
   *                 connection string zookeeper.connect.
   */
  public static kafka.javaapi.consumer.ConsumerConnector createJavaConsumerCo
}

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

  /**
   *  Create a list of message streams of type T for each topic, using the de
   */
  public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(

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

  /**
   *  Create a list of message streams for topics matching a wildcard, using
   */
  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(Topic

  /**
   *  Create a list of message streams for topics matching a wildcard, using
   */
  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(Topic

  /**
   *  Commit the offsets of all topic/partitions connected by this connector.
   */
  public void commitOffsets();

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

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

2.2.2 Old Simple Consumer API

```
class kafka.javaapi.consumer.SimpleConsumer {
  /**
   *  Fetch a set of messages from a topic.
   *
   *  @param request specifies the topic name, topic partition, starting byte
   *  @return a set of fetched messages
   */
  public FetchResponse fetch(kafka.javaapi.FetchRequest request);

  /**
   *  Fetch metadata for a sequence of topics.
   *
   *  @param request specifies the versionId, clientId, sequence of topics.
   *  @return metadata for each topic in the request.
   */
  public kafka.javaapi.TopicMetadataResponse send(kafka.javaapi.TopicMetadataI

  /**
   *  Get a list of valid offsets (up to maxSize) before the given time.
   *
   *  @param request a [[kafka.javaapi.OffsetRequest]] object.
   *  @return a [[kafka.javaapi.OffsetResponse]] object.
   */
  public kafka.javaapi.OffsetResponse getOffsetsBefore(OffsetRequest request);

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

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

2.2.3 New Consumer API

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

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

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

2.3 Streams API

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

```
<dependency>
    <groupId>org.apache.kafka</groupId>
```

```
        <artifactId>kafka-streams</artifactId>
        <version>0.10.0.0</version>
    </dependency>
```

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

3. Configuration

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

3.1 Broker Configs

The essential configurations are the following:

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

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

Name	Description	Type	
zookeeper.connect	Zookeeper host string	string	
advertised.host.name	DEPRECATED: only used when `advertised.listeners` or `listeners` are not set. Use `advertised.listeners` instead. Hostname to publish to ZooKeeper for clients to use. In IaaS environments, this may need to be different from the interface to which the broker binds. If this is not set, it will use the value for `host.name` if configured. Otherwise it will use the value returned from java.net.InetAddress.getCanonicalHostName().	string	null
advertised.listeners	Listeners to publish to ZooKeeper for clients to use, if different than the listeners above. In IaaS environments, this may need to be different from the interface to which the broker binds. If this is not set, the value for `listeners` will be used.	string	null
advertised.port	DEPRECATED: only used when `advertised.listeners` or `listeners` are not set. Use `advertised.listeners` instead. The port to publish to ZooKeeper for clients to use. In IaaS environments, this may need to be different from the port to which the broker binds. If this is not set, it will publish the same port that the broker binds to.	int	null
auto.create.topics.enable	Enable auto creation of topic on the server	boolean	true
auto.leader.rebalance.enable	Enables auto leader balancing. A background thread checks and triggers leader balance if	boolean	true

	required at regular intervals		
background.threads	The number of threads to use for various background processing tasks	int	10
broker.id	The broker id for this server. If unset, a unique broker id will be generated.To avoid conflicts between zookeeper generated broker id's and user configured broker id's, generated broker idsstart from reserved.broker.max.id + 1.	int	-1
compression.type	Specify the final compression type for a given topic. This configuration accepts the standard compression codecs ('gzip', 'snappy', 'lz4'). It additionally accepts 'uncompressed' which is equivalent to no compression; and 'producer' which means retain the original compression codec set by the producer.	string	prod
delete.topic.enable	Enables delete topic. Delete topic through the admin tool will have no effect if this config is turned off	boolean	false
host.name	DEPRECATED: only used when `listeners` is not set. Use `listeners` instead. hostname of broker. If this is set, it will only bind to this address. If this is not set, it will bind to all interfaces	string	""
leader.imbalance.check.interval.seconds	The frequency with which the partition rebalance check is triggered by the controller	long	300
leader.imbalance.per.broker.percentage	The ratio of leader imbalance allowed per broker. The controller would trigger a leader balance if it goes above this value per broker. The value is specified in percentage.	int	10
listeners	Listener List - Comma-separated list of URIs we will listen on and their protocols. Specify hostname as 0.0.0.0 to bind to all interfaces. Leave hostname empty to bind to default interface. Examples of legal listener lists: PLAINTEXT://myhost:9092,TRACE://:9091 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093	string	null
log.dir	The directory in which the log data is kept (supplemental for log.dirs property)	string	/tmp
log.dirs	The directories in which the log data is kept. If not set, the value in log.dir is used	string	null
log.flush.interval.messages	The number of messages accumulated on a log partition before messages are flushed to disk	long	9223
log.flush.interval.ms	The maximum time in ms that a message in any topic is kept in memory before flushed to disk. If not set, the value in	long	null

	log.flush.scheduler.interval.ms is used		
log.flush.offset.checkpoint.interval.ms	The frequency with which we update the persistent record of the last flush which acts as the log recovery point	int	6000
log.flush.scheduler.interval.ms	The frequency in ms that the log flusher checks whether any log needs to be flushed to disk	long	9223
log.retention.bytes	The maximum size of the log before deleting it	long	-1
log.retention.hours	The number of hours to keep a log file before deleting it (in hours), tertiary to log.retention.ms property	int	168
log.retention.minutes	The number of minutes to keep a log file before deleting it (in minutes), secondary to log.retention.ms property. If not set, the value in log.retention.hours is used	int	null
log.retention.ms	The number of milliseconds to keep a log file before deleting it (in milliseconds), If not set, the value in log.retention.minutes is used	long	null
log.roll.hours	The maximum time before a new log segment is rolled out (in hours), secondary to log.roll.ms property	int	168
log.roll.jitter.hours	The maximum jitter to subtract from logRollTimeMillis (in hours), secondary to log.roll.jitter.ms property	int	0
log.roll.jitter.ms	The maximum jitter to subtract from logRollTimeMillis (in milliseconds). If not set, the value in log.roll.jitter.hours is used	long	null
log.roll.ms	The maximum time before a new log segment is rolled out (in milliseconds). If not set, the value in log.roll.hours is used	long	null
log.segment.bytes	The maximum size of a single log file	int	1073
log.segment.delete.delay.ms	The amount of time to wait before deleting a file from the filesystem	long	6000
message.max.bytes	The maximum size of message that the server can receive	int	1000
min.insync.replicas	define the minimum number of replicas in ISR needed to satisfy a produce request with acks=all (or -1)	int	1
num.io.threads	The number of io threads that the server uses for carrying out network requests	int	8
num.network.threads	the number of network threads that the server uses for handling network requests	int	3
num.recovery.threads.per.data.dir	The number of threads per data directory to be used for log recovery at startup and flushing at shutdown	int	1
	Number of fetcher threads used to replicate		

num.replica.fetchers	messages from a source broker. Increasing this value can increase the degree of I/O parallelism in the follower broker.	int	1
offset.metadata.max.bytes	The maximum size for a metadata entry associated with an offset commit	int	4096
offsets.commit.required.acks	The required acks before the commit can be accepted. In general, the default (-1) should not be overridden	short	-1
offsets.commit.timeout.ms	Offset commit will be delayed until all replicas for the offsets topic receive the commit or this timeout is reached. This is similar to the producer request timeout.	int	5000
offsets.load.buffer.size	Batch size for reading from the offsets segments when loading offsets into the cache.	int	5242
offsets.retention.check.interval.ms	Frequency at which to check for stale offsets	long	6000
offsets.retention.minutes	Log retention window in minutes for offsets topic	int	1440
offsets.topic.compression.codec	Compression codec for the offsets topic - compression may be used to achieve "atomic" commits	int	0
offsets.topic.num.partitions	The number of partitions for the offset commit topic (should not change after deployment)	int	50
offsets.topic.replication.factor	The replication factor for the offsets topic (set higher to ensure availability). To ensure that the effective replication factor of the offsets topic is the configured value, the number of alive brokers has to be at least the replication factor at the time of the first request for the offsets topic. If not, either the offsets topic creation will fail or it will get a replication factor of min(alive brokers, configured replication factor)	short	3
offsets.topic.segment.bytes	The offsets topic segment bytes should be kept relatively small in order to facilitate faster log compaction and cache loads	int	1048
port	DEPRECATED: only used when \`listeners\` is not set. Use \`listeners\` instead. the port to listen and accept connections on	int	9092
queued.max.requests	The number of queued requests allowed before blocking the network threads	int	500
quota.consumer.default	Any consumer distinguished by clientId/consumer group will get throttled if it fetches more bytes than this value per-second	long	9223
quota.producer.default	Any producer distinguished by clientId will get throttled if it produces more bytes than this value per-second	long	9223

replica.fetch.max.bytes	The number of bytes of messages to attempt to fetch	int	1048
replica.fetch.min.bytes	Minimum bytes expected for each fetch response. If not enough bytes, wait up to replicaMaxWaitTimeMs	int	1
replica.fetch.wait.max.ms	max wait time for each fetcher request issued by follower replicas. This value should always be less than the replica.lag.time.max.ms at all times to prevent frequent shrinking of ISR for low throughput topics	int	500
replica.high.watermark.checkpoint.interval.ms	The frequency with which the high watermark is saved out to disk	long	5000
replica.lag.time.max.ms	If a follower hasn't sent any fetch requests or hasn't consumed up to the leaders log end offset for at least this time, the leader will remove the follower from isr	long	1000
replica.socket.receive.buffer.bytes	The socket receive buffer for network requests	int	6553
replica.socket.timeout.ms	The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms	int	3000
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	3000
socket.receive.buffer.bytes	The SO_RCVBUF buffer of the socket sever sockets	int	1024
socket.request.max.bytes	The maximum number of bytes in a socket request	int	1048
socket.send.buffer.bytes	The SO_SNDBUF buffer of the socket sever sockets	int	1024
unclean.leader.election.enable	Indicates whether to enable replicas not in the ISR set to be elected as leader as a last resort, even though doing so may result in data loss	boolean	true
zookeeper.connection.timeout.ms	The max time that the client waits to establish a connection to zookeeper. If not set, the value in zookeeper.session.timeout.ms is used	int	null
zookeeper.session.timeout.ms	Zookeeper session timeout	int	6000
zookeeper.set.acl	Set client to use secure ACLs	boolean	false
broker.id.generation.enable	Enable automatic broker id generation on the server? When enabled the value configured for reserved.broker.max.id should be reviewed.	boolean	true
broker.rack	Rack of the broker. This will be used in rack aware replication assignment for fault	string	null

	tolerance. Examples: `RACK1`, `us-east-1d`		
connections.max.idle.ms	Idle connections timeout: the server socket processor threads close the connections that idle more than this	long	6000
controlled.shutdown.enable	Enable controlled shutdown of the server	boolean	true
controlled.shutdown.max.retries	Controlled shutdown can fail for multiple reasons. This determines the number of retries when such failure happens	int	3
controlled.shutdown.retry.backoff.ms	Before each retry, the system needs time to recover from the state that caused the previous failure (Controller fail over, replica lag etc). This config determines the amount of time to wait before retrying.	long	5000
controller.socket.timeout.ms	The socket timeout for controller-to-broker channels	int	3000
default.replication.factor	default replication factors for automatically created topics	int	1
fetch.purgatory.purge.interval.requests	The purge interval (in number of requests) of the fetch request purgatory	int	1000
group.max.session.timeout.ms	The maximum allowed session timeout for registered consumers. Longer timeouts give consumers more time to process messages in between heartbeats at the cost of a longer time to detect failures.	int	3000
group.min.session.timeout.ms	The minimum allowed session timeout for registered consumers. Shorter timeouts leader to quicker failure detection at the cost of more frequent consumer heartbeating, which can overwhelm broker resources.	int	6000
inter.broker.protocol.version	Specify which version of the inter-broker protocol will be used. This is typically bumped after all brokers were upgraded to a new version. Example of some valid values are: 0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2.1, 0.9.0.0, 0.9.0.1 Check ApiVersion for the full list.	string	0.10.
log.cleaner.backoff.ms	The amount of time to sleep when there are no logs to clean	long	1500
log.cleaner.dedupe.buffer.size	The total memory used for log deduplication across all cleaner threads	long	1342
log.cleaner.delete.retention.ms	How long are delete records retained?	long	8640
log.cleaner.enable	Enable the log cleaner process to run on the server? Should be enabled if using any topics with a cleanup.policy=compact including the internal offsets topic. If disabled those topics will not be compacted and continually grow in size.	boolean	true

log.cleaner.io.buffer.load.factor	Log cleaner dedupe buffer load factor. The percentage full the dedupe buffer can become. A higher value will allow more log to be cleaned at once but will lead to more hash collisions	double	0.9
log.cleaner.io.buffer.size	The total memory used for log cleaner I/O buffers across all cleaner threads	int	5242
log.cleaner.io.max.bytes.per.second	The log cleaner will be throttled so that the sum of its read and write i/o will be less than this value on average	double	1.797
log.cleaner.min.cleanable.ratio	The minimum ratio of dirty log to total log for a log to eligible for cleaning	double	0.5
log.cleaner.threads	The number of background threads to use for log cleaning	int	1
log.cleanup.policy	The default cleanup policy for segments beyond the retention window, must be either "delete" or "compact"	string	delet
log.index.interval.bytes	The interval with which we add an entry to the offset index	int	4096
log.index.size.max.bytes	The maximum size in bytes of the offset index	int	1048
log.message.format.version	Specify the message format version the broker will use to append messages to the logs. The value should be a valid ApiVersion. Some examples are: 0.8.2, 0.9.0.0, 0.10.0, check ApiVersion for more details. By setting a particular message format version, the user is certifying that all the existing messages on disk are smaller or equal than the specified version. Setting this value incorrectly will cause consumers with older versions to break as they will receive messages with a format that they don't understand.	string	0.10.
log.message.timestamp.difference.max.ms	The maximum difference allowed between the timestamp when a broker receives a message and the timestamp specified in the message. If message.timestamp.type=CreateTime, a message will be rejected if the difference in timestamp exceeds this threshold. This configuration is ignored if message.timestamp.type=LogAppendTime.	long	9223
log.message.timestamp.type	Define whether the timestamp in the message is message create time or log append time. The value should be either \`CreateTime\` or \`LogAppendTime\`	string	Creat
log.preallocate	Should pre allocate file when create new segment? If you are using Kafka on Windows,	boolean	false

	you probably need to set it to true.		
log.retention.check.interval.ms	The frequency in milliseconds that the log cleaner checks whether any log is eligible for deletion	long	3000
max.connections.per.ip	The maximum number of connections we allow from each ip address	int	2147
max.connections.per.ip.overrides	Per-ip or hostname overrides to the default maximum number of connections	string	""
num.partitions	The default number of log partitions per topic	int	1
principal.builder.class	The fully qualified name of a class that implements the PrincipalBuilder interface, which is currently used to build the Principal for connections with the SSL SecurityProtocol.	class	class org.a
producer.purgatory.purge.interval.requests	The purge interval (in number of requests) of the producer request purgatory	int	1000
replica.fetch.backoff.ms	The amount of time to sleep when fetch partition error occurs.	int	1000
reserved.broker.max.id	Max number that can be used for a broker.id	int	1000
sasl.enabled.mechanisms	The list of SASL mechanisms enabled in the Kafka server. The list may contain any mechanism for which a security provider is available. Only GSSAPI is enabled by default.	list	[GSS
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	6000
sasl.kerberos.principal.to.local.rules	A list of rules for mapping from principal names to short names (typically operating system usernames). The rules are evaluated in order and the first rule that matches a principal name is used to map it to a short name. Any later rules in the list are ignored. By default, principal names of the form {username}/{hostname}@{REALM} are mapped to {username}. For more details on the format please see **security authorization and acls**.	list	[DEF
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8
sasl.mechanism.inter.broker.protocol	SASL mechanism used for inter-broker communication. Default is GSSAPI.	string	GSSA

security.inter.broker.protocol	Security protocol used to communicate between brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAI
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.	list	null
ssl.client.auth	Configures kafka broker to request client authentication. The following settings are common: • `ssl.client.auth=required` If set to required client authentication is required. • `ssl.client.auth=requested` This means client authentication is optional. unlike requested , if this option is set client can choose not to provide authentication information about itself • `ssl.client.auth=none` This means client authentication is not needed.	string	none
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	[TLS
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string	SunX
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null
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.	password	null
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known	string	TLS

	security vulnerabilities.		
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX
ssl.truststore.location	The location of the trust store file.	string	null
ssl.truststore.password	The password for the trust store file.	password	null
ssl.truststore.type	The file format of the trust store file.	string	JKS
authorizer.class.name	The authorizer class that should be used for authorization	string	""
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	3000
quota.window.num	The number of samples to retain in memory	int	11
quota.window.size.seconds	The time span of each sample	int	1
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string	null
zookeeper.sync.time.ms	How far a ZK follower can be behind a ZK leader	int	2000

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 -
       --replication-factor 1 --config max.message.bytes=64000 --config flusl
```

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.

Property	Default	Server Default Property	Description
cleanup.policy	delete	log.cleanup.policy	A string that is either "delete" or "compact" retention policy to use on old log segments ("delete") will discard old segments when t limit has been reached. The "compact" sett compaction on the topic.
delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms	The amount of time to retain delete tombst compacted topics. This setting also gives a which a consumer must complete a read if t ensure that they get a valid snapshot of the delete tombstones may be collected before
flush.messages	None	log.flush.interval.messages	This setting allows specifying an interval at fsync of data written to the log. For exampl would fsync after every message; if it were 5 every five messages. In general we recomm use replication for durability and allow the background flush capabilities as it is more e be overridden on a per-topic basis (see the section).
flush.ms	None	log.flush.interval.ms	This setting allows specifying a time interva fsync of data written to the log. For exampl would fsync after 1000 ms had passed. In ge not set this and use replication for durabilit system's background flush capabilities as it
index.interval.bytes	4096	log.index.interval.bytes	This setting controls how frequently Kafka a offset index. The default setting ensures tha roughly every 4096 bytes. More indexing all to the exact position in the log but makes th probably don't need to change this.
max.message.bytes	1,000,000	message.max.bytes	This is largest message size Kafka will allow topic. Note that if you increase this size you consumer's fetch size so they can fetch mes
min.cleanable.dirty.ratio	0.5	log.cleaner.min.cleanable.ratio	This configuration controls how frequently attempt to clean the log (assuming log com default we will avoid cleaning a log where n has been compacted. This ratio bounds the in the log by duplicates (at 50% at most 50% duplicates). A higher ratio will mean fewer, but will mean more wasted space in the log
			When a producer sets acks to "all", min.insy minimum number of replicas that must ack write to be considered successful. If this mi then the producer will raise an exception (e or NotEnoughReplicasAfterAppend). When

min.insync.replicas	1	min.insync.replicas	min.insync.replicas and acks allow you to e guarantees. A typical scenario would be to c replication factor of 3, set min.insync.replic acks of "all". This will ensure that the produ majority of replicas do not receive a write.
retention.bytes	None	log.retention.bytes	This configuration controls the maximum si before we will discard old log segments to f using the "delete" retention policy. By defau only a time limit.
retention.ms	7 days	log.retention.minutes	This configuration controls the maximum ti before we will discard old log segments to f using the "delete" retention policy. This rep soon consumers must read their data.
segment.bytes	1 GB	log.segment.bytes	This configuration controls the segment file and cleaning is always done a file at a time means fewer files but less granular control c
segment.index.bytes	10 MB	log.index.size.max.bytes	This configuration controls the size of the ir file positions. We preallocate this index file rolls. You generally should not need to char
segment.ms	7 days	log.roll.hours	This configuration controls the period of tin force the log to roll even if the segment file retention can delete or compact old data.
segment.jitter.ms	0	log.roll.jitter.{ms,hours}	The maximum jitter to subtract from logRol

3.2 Producer Configs

Below is the configuration of the Java producer:

Name	Description	Type	
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	
key.serializer	Serializer class for key that implements the `Serializer` interface.	class	
value.serializer	Serializer class for value that implements the `Serializer` interface.	class	
	The number of acknowledgments the producer requires the leader to have received before		

acks	considering a request complete. This controls the durability of records that are sent. The following settings are common: `acks=0` If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the `retries` configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1. `acks=1` This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost. `acks=all` This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.	string	1
buffer.memory	The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will block for `max.block.ms` after which it will throw an exception. This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.	long	3355
compression.type	The compression type for all data generated by the producer. The default is none (i.e. no compression). Valid values are `none`, `gzip`, `snappy`, or `lz4`. Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression).	string	none
	Setting a value greater than zero will cause the client		

retries	to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries without setting `max.in.flight.requests.per.connection` to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first.	int	0
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null
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.	password	null
ssl.truststore.location	The location of the trust store file.	string	null
ssl.truststore.password	The password for the trust store file.	password	null
batch.size	The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes. No attempt will be made to batch records larger than this size. Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent. A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.	int	1638
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	""
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	5400
	The producer groups together any records that arrive in between request transmissions into a single		

linger.ms	batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding a small amount of artificial delay—that is, rather than immediately sending out a record the producer will wait for up to the given delay to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get `batch.size` worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting `linger.ms=5`, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absense of load.	long	0
max.block.ms	The configuration controls how long `KafkaProducer.send()` and `KafkaProducer.partitionsFor()` will block.These methods can be blocked either because the buffer is full or metadata unavailable.Blocking in the user-supplied serializers or partitioner will not be counted against this timeout.	long	6000
max.request.size	The maximum size of a request in bytes. This is also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.	int	1048
partitioner.class	Partitioner class that implements the `Partitioner` interface.	class	class org.a
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	int	3276
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	3000
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null

sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSS/
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAI
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data.	int	1310
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	[TLS\
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null
ssl.truststore.type	The file format of the trust store file.	string	JKS
timeout.ms	The configuration controls the maximum amount of time the server will wait for acknowledgments from followers to meet the acknowledgment requirements the producer has specified with the `acks` configuration. If the requested number of acknowledgments are not met when the timeout elapses an error will be returned. This timeout is measured on the server side and does not include the network latency of the request.	int	3000
block.on.buffer.full	When our memory buffer is exhausted we must either stop accepting new records (block) or throw errors. By default this setting is false and the producer will no longer throw a BufferExhaustException but instead will use the `max.block.ms` value to block, after which it will throw a TimeoutException. Setting this property to true will set the `max.block.ms` to Long.MAX_VALUE. *Also if this property is set to true, parameter* `metadata.fetch.timeout.ms` *is not longer honored.* This parameter is deprecated and will be removed in a future release. Parameter `max.block.ms` should be used instead.	boolean	false
	A list of classes to use as interceptors. Implementing		

interceptor.classes	the ProducerInterceptor interface allows you to intercept (and possibly mutate) the records received by the producer before they are published to the Kafka cluster. By default, there are no interceptors.	list	null
max.in.flight.requests.per.connection	The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).	int	5
metadata.fetch.timeout.ms	The first time data is sent to a topic we must fetch metadata about that topic to know which servers host the topic's partitions. This fetch to succeed before throwing an exception back to the client.	long	6000
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long	3000
metric.reporters	A list of classes to use as metrics reporters. Implementing the MetricReporter interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	3000
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	long	50
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	6000
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8

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

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`

Property	Default	Description
group.id		A string that uniquely identifies the group of consumer processe belongs. By setting the same group id multiple processes indica the same consumer group.
zookeeper.connect		Specifies the ZooKeeper connection string in the form `hostnar` port are the host and port of a ZooKeeper server. To allow conne ZooKeeper nodes when that ZooKeeper machine is down you ca hosts in the form `hostname1:port1,hostname2:port2,` The server may also have a ZooKeeper chroot path as part of it's string which puts its data under some path in the global ZooKee consumer should use the same chroot path in its connection str chroot path of `/chroot/path` you would give the connection `hostname1:port1,hostname2:port2,hostname3:po`
consumer.id	null	Generated automatically if not set.
socket.timeout.ms	30 * 1000	The socket timeout for network requests. The actual timeout se socket.timeout.ms.
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests

fetch.message.max.bytes	1024 * 1024	The number of bytes of messages to attempt to fetch for each tc request. These bytes will be read into memory for each partitior memory used by the consumer. The fetch request size must be a maximum message size the server allows or else it is possible fo messages larger than the consumer can fetch.
num.consumer.fetchers	1	The number fetcher threads used to fetch data.
auto.commit.enable	true	If true, periodically commit to ZooKeeper the offset of messages consumer. This committed offset will be used when the process which the new consumer will begin.
auto.commit.interval.ms	60 * 1000	The frequency in ms that the consumer offsets are committed tc
queued.max.message.chunks	2	Max number of message chunks buffered for consumption. Each fetch.message.max.bytes.
rebalance.max.retries	4	When a new consumer joins a consumer group the set of consur "rebalance" the load to assign partitions to each consumer. If th changes while this assignment is taking place the rebalance will controls the maximum number of attempts before giving up.
fetch.min.bytes	1	The minimum amount of data the server should return for a fetc data is available the request will wait for that much data to accu the request.
fetch.wait.max.ms	100	The maximum amount of time the server will block before answ there isn't sufficient data to immediately satisfy fetch.min.bytes
rebalance.backoff.ms	2000	Backoff time between retries during rebalance. If not set explicit zookeeper.sync.time.ms is used.
refresh.leader.backoff.ms	200	Backoff time to wait before trying to determine the leader of a p its leader.
auto.offset.reset	largest	What to do when there is no initial offset in ZooKeeper or if an o * 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 ava after the specified interval
exclude.internal.topics	true	Whether messages from internal topics (such as offsets) should consumer.
client.id	group id value	The client id is a user-specified string sent in each request to hel logically identify the application making the request.
zookeeper.session.timeout.ms	6000	ZooKeeper session timeout. If the consumer fails to heartbeat tc 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 connecti
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 ret fetch/commit requests.
offsets.channel.socket.timeout.ms	10000	Socket timeout when reading responses for offset fetch/commit also used for ConsumerMetadata requests that are used to quer

offsets.commit.max.retries	5	Retry the offset commit up to this many times on failure. This re offset commits during shut-down. It does not apply to commits commit thread. It also does not apply to attempts to query for tl before committing offsets. i.e., if a consumer metadata request 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 addition to Kafka). This is required during migration from zooke to kafka-based offset storage. With respect to any given consum this off after all instances within that group have been migrated commits offsets to the broker (instead of directly to ZooKeeper)
partition.assignment.strategy	range	Select between the "range" or "roundrobin" strategy for assigni streams. The round-robin partition assignor lays out all the available par available consumer threads. It then proceeds to do a round-rob partition to consumer thread. If the subscriptions of all consum then the partitions will be uniformly distributed. (i.e., the partiti be within a delta of exactly one across all consumer threads.) Ro permitted only if: (a) Every topic has the same number of strean instance (b) The set of subscribed topics is identical for every co the group. Range partitioning works on a per-topic basis. For each topic, w partitions in numeric order and the consumer threads in lexicog divide the number of partitions by the total number of consume determine the number of partitions to assign to each consumer. divide, then the first few consumers will have one extra partitior

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:

Name	Description	Type	
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	list	

	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).		
key.deserializer	Deserializer class for key that implements the `Deserializer` interface.	class	
value.deserializer	Deserializer class for value that implements the `Deserializer` interface.	class	
fetch.min.bytes	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than 1 will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.	int	1
group.id	A unique string that identifies the consumer group this consumer belongs to. This property is required if the consumer uses either the group management functionality by using `subscribe(topic)` or the Kafka-based offset management strategy.	string	""
heartbeat.interval.ms	The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000
	The maximum amount of data per-		

max.partition.fetch.bytes	partition the server will return. The maximum total memory used for a request will be `#partitions * max.partition.fetch.bytes`. This size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch. If that happens, the consumer can get stuck trying to fetch a large message on a certain partition.	int	1048576
session.timeout.ms	The timeout used to detect failures when using Kafka's group management facilities. When a consumer's heartbeat is not received within the session timeout, the broker will mark the consumer as failed and rebalance the group. Since heartbeats are sent only when poll() is invoked, a higher session timeout allows more time for message processing in the consumer's poll loop at the cost of a longer time to detect hard failures. See also `max.poll.records` for another option to control the processing time in the poll loop. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.	int	30000
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null
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.	password	null
ssl.truststore.location	The location of the trust store file.	string	null
ssl.truststore.password	The password for the trust store file.	password	null
	What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):		

auto.offset.reset	• earliest: automatically reset the offset to the earliest offset • latest: automatically reset the offset to the latest offset • none: throw exception to the consumer if no previous offset is found for the consumer's group • anything else: throw exception to the consumer.	string	latest
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	540000
enable.auto.commit	If true the consumer's offset will be periodically committed in the background.	boolean	true
exclude.internal.topics	Whether records from internal topics (such as offsets) should be exposed to the consumer. If set to `true` the only way to receive records from an internal topic is subscribing to it.	boolean	true
max.poll.records	The maximum number of records returned in a single call to poll().	int	2147483647
partition.assignment.strategy	The class name of the partition assignment strategy that the client will use to distribute partition ownership amongst consumer instances when group management is used	list	[org.apache.kafk
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	int	65536
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	40000
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSSAPI
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEXT

send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data.	int	131072
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	[TLSv1.2, TLSv1.1
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null
ssl.truststore.type	The file format of the trust store file.	string	JKS
auto.commit.interval.ms	The frequency in milliseconds that the consumer offsets are auto-committed to Kafka if `enable.auto.commit` is set to `true`.	long	5000
check.crcs	Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance.	boolean	true
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	""
fetch.max.wait.ms	The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch.min.bytes.	int	500
interceptor.classes	A list of classes to use as interceptors. Implementing the `ConsumerInterceptor` interface allows you to intercept (and possibly mutate) records received by the consumer. By default, there are no interceptors.	list	null

metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long	300000
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	long	50
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/bin/kinit
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	60000
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.	list	null
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server	string	null

	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.	string	SunX509
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX

3.4 Kafka Connect Configs

Below is the configuration of the Kafka Connect framework.

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

cluster	namespace so multiple Kafka Connect clusters or instances may co-exist while sharing a single Kafka cluster.	string	connect
heartbeat.interval.ms	The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000
session.timeout.ms	The timeout used to detect failures when using Kafka's group management facilities.	int	30000
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null
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.	password	null
ssl.truststore.location	The location of the trust store file.	string	null
ssl.truststore.password	The password for the trust store file.	password	null
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	540000
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	int	32768
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	40000
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSSAPI
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEX
	The size of the TCP send buffer (SO_SNDBUF) to		

send.buffer.bytes	use when sending data.	int	131072
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	[TLSv1.2, TLSv1.1,
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null
ssl.truststore.type	The file format of the trust store file.	string	JKS
worker.sync.timeout.ms	When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before giving up, leaving the group, and waiting a backoff period before rejoining.	int	3000
worker.unsync.backoff.ms	When the worker is out of sync with other workers and fails to catch up within worker.sync.timeout.ms, leave the Connect cluster for this long before rejoining.	int	300000
access.control.allow.methods	Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD.	string	""
access.control.allow.origin	Value to set the Access-Control-Allow-Origin header to for REST API requests.To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '*' to allow access from any domain. The default value only allows access from the domain of the REST API.	string	""
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string	""
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or	long	300000

	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.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000
offset.flush.interval.ms	Interval at which to try committing offsets for tasks.	long	60000
offset.flush.timeout.ms	Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt.	long	5000
reconnect.backoff.ms	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	long	50
rest.advertised.host.name	If this is set, this is the hostname that will be given out to other workers to connect to.	string	null
rest.advertised.port	If this is set, this is the port that will be given out to other workers to connect to.	int	null
rest.host.name	Hostname for the REST API. If this is set, it will only bind to this interface.	string	null
rest.port	Port for the REST API to listen on.	int	8083
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/bin/l
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	60000
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8
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	list	null

	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.	string	null
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string	SunX509
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX
task.shutdown.graceful.timeout.ms	Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially.	long	5000

3.5 Kafka Streams Configs

Below is the configuration of the Kafka Streams client library.

Name	Description	Type	
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	
	An id string to pass to the server when making requests. The purpose of this is		

client.id	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	class org.apache.kafka.common.s
partition.grouper	Partition grouper class that implements the `PartitionGrouper` interface.	class	class org.apache.kafka.streams.p
replication.factor	The replication factor for change log topics and repartition topics created by the stream processing application.	int	1
state.dir	Directory location for state store.	string	/tmp/kafka-streams
timestamp.extractor	Timestamp extractor class that implements the `TimestampExtractor` interface.	class	class org.apache.kafka.streams.proces
value.serde	Serializer / deserializer class for value that implements the `Serde` interface.	class	class org.apache.kafka.common.s
buffered.records.per.partition	The maximum number of records to buffer per partition.	int	1000
commit.interval.ms	The frequency with which to save the position of the processor.	long	30000
metric.reporters	A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	[]
metrics.num.samples	The number of samples maintained to compute metrics.	int	2
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000
num.standby.replicas	The number of standby replicas for each task.	int	0
num.stream.threads	The number of threads to execute stream processing.	int	1
poll.ms	The amount of time in milliseconds to block waiting for input.	long	100
state.cleanup.delay.ms	The amount of time in milliseconds to wait before deleting state when a partition has migrated.	long	60000

4. Design

4.1 Motivation

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

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

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

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

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

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

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

4.2 Persistence

Don't fear the filesystem!

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

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a **JBOD** configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this **ACM Queue article**; they actually find that **sequential disk access can in some cases be faster than random memory access!**

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

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

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

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

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

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

Constant Time Suffices

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

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

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

4.3 Efficiency

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

We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.

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

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

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

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

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

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

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

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

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

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

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

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

End-to-end Batch Compression

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

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

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

4.4 The Producer

Load balancing

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

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

Asynchronous send

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

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

4.5 The Consumer

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

Push vs. pull

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

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

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

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

Consumer Position

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

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

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

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

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

Offline Data Load

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

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

4.6 Message Delivery Semantics

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

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

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

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

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

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

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

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

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

deduplication.

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

4.7 Replication

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Unclean leader election: What if they all die?

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

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

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

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

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

Availability and Durability Guarantees

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

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

Replica Management

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

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

4.8 Log Compaction

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

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

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

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

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

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

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

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

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

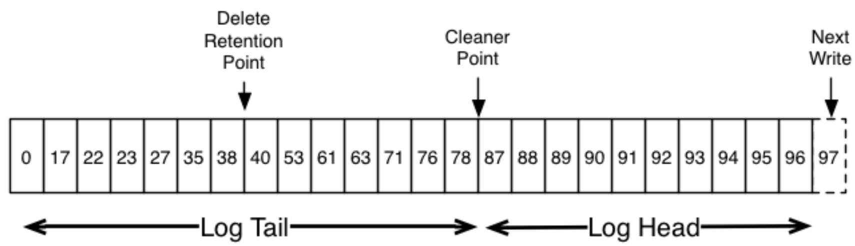
Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each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

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

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

Log Compaction Basics

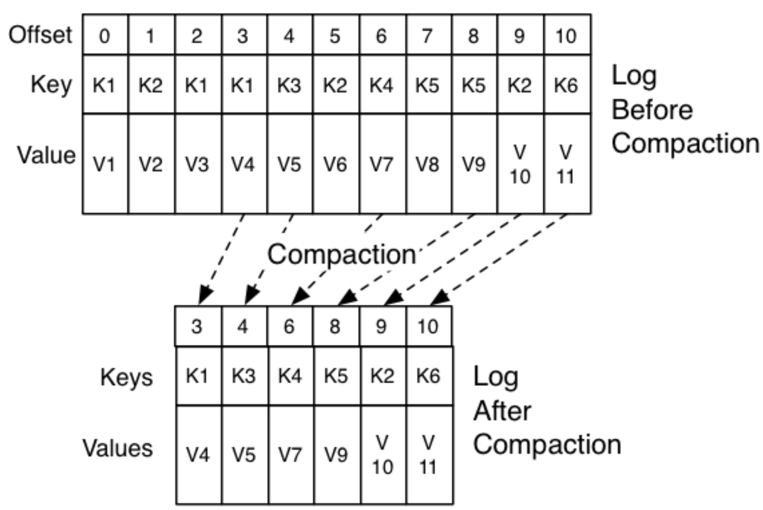
Here is a high-level picture that shows the logical structure of a Kafka log with the offset for each message.



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

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

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



What guarantees does log compaction provide?

Log compaction guarantees the following:

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

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

Log Compaction Details

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

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

Configuring The Log Cleaner

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

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

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

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

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

Further cleaner configurations are described **here**.

Log Compaction Limitations

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

4.9 Quotas

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

Why are quotas necessary?

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

Enforcement

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

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

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

Quota overrides

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

5. Implementation

5.1 API Design

Producer APIs

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

```
class Producer {

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

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

  /* 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
   *              if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from t
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int m
}
```

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 ov
   * 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> to

  /**
   * 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 :
 *    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 tl
 * 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

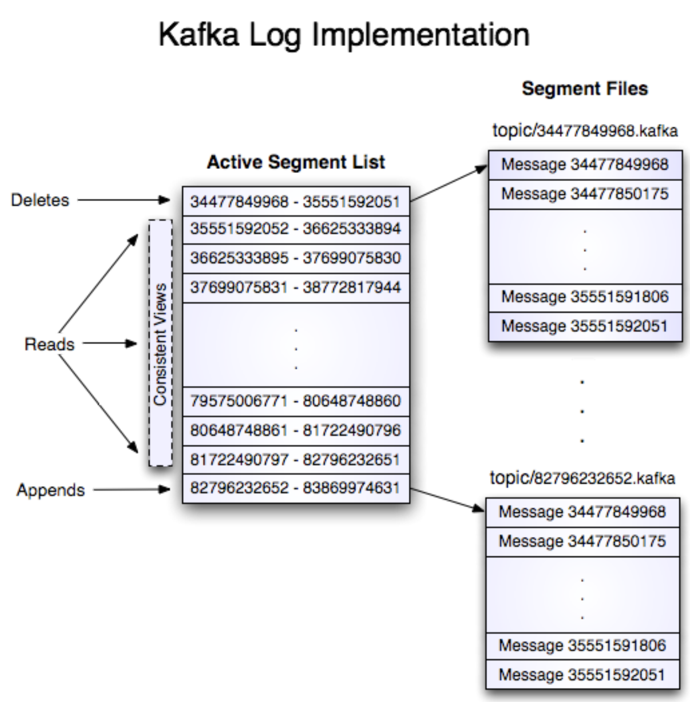
A log for a topic named "my_topic" with two partitions consists of two directories (namely `my_topic_0` and `my_topic_1`) populated with data files containing the messages for that topic. The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer *N* storing the message 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 +
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":[...],"h
```

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":...,"
```

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":{..
```

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
```

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 (ephe
```

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 releas
```

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

6. Operations

Here is some information on actually running Kafka as a production system based on usage and experience at LinkedIn. Please send us any additional tips you know of.

6.1 Basic Kafka Operations

This section will review the most common operations you will perform on your Kafka cluster. All of the tools reviewed in this section are available under the `bin/` directory of the Kafka distribution and each tool will print details on all possible commandline options if it is run with no arguments.

Adding and removing topics

You have the option of either adding topics manually or having them be created automatically when data is first published to a non-existent topic. If topics are auto-created then you may want to tune the default **topic configurations** used for auto-created topics.

Topics are added and modified using the topic tool:

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --create --topic my_to
      --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_top:
      --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_top:
```

To remove a config:

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_top:
```

And finally deleting a topic:

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --delete --topic my_to
```

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
```

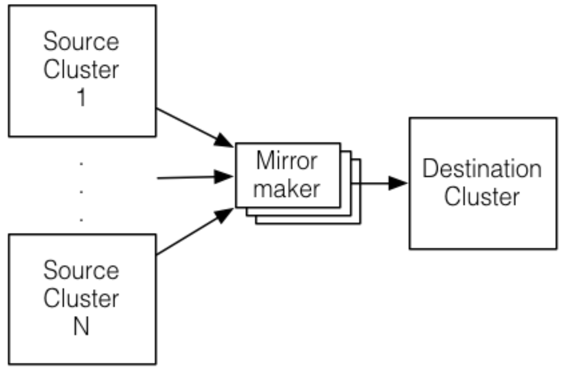
When a topic is **created**, **modified** or replicas are **redistributed**, the rack constraint will be honoured, ensuring replicas span as many racks as they can (a partition will span min(#racks, replication-factor) different racks).

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.p
       --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 local
Group           Topic                          Pid Offset          logSize
my-group        my-topic                       0   0               0
```

```
my-group        my-topic                      1    0                 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

GROUP                          TOPIC                          PARTITION  CURRI
test-consumer-group            test-foo                       0          1
```

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 broker1:909:
```

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
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-j
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-j
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]},{"t
```

Then, use the json file with the --execute option to start the reassignment process:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-j
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-js
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-j
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-js
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: 
```

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 'produ
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 clie
Configs for clients:clientA are producer_byte_rate=1024,consumer_byte_rate=20
```

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:G1HeapRegio
-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:

Description	Mbean name	
Message in rate	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec	
Byte in rate	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec	
Request rate	kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|FetchConsumer|FetchFollower}	
Byte out rate	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec	
Log flush rate and time	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs	
# of under replicated partitions (|ISR| < |all replicas|)	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	0
Is controller active on broker	kafka.controller:type=KafkaController,name=ActiveControllerCount	only shou
Leader election rate	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	non- brok
Unclean leader election rate	kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec	0
Partition counts	kafka.server:type=ReplicaManager,name=PartitionCount	most
Leader replica counts	kafka.server:type=ReplicaManager,name=LeaderCount	most
ISR shrink rate	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	If a b some shrin agair once caug expe shrin is 0.
ISR expansion rate	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	See a
Max lag in messages btw follower and leader replicas	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	lag sl the n prod
Lag in messages per follower replica	kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=([-.\w]+),topic=([-.\w]+),partition=([0-9]+)	lag sl the n prod
Requests waiting in the producer purgatory	kafka.server:type=ProducerRequestPurgatory,name=PurgatorySize	non-
Requests waiting in the fetch purgatory	kafka.server:type=FetchRequestPurgatory,name=PurgatorySize	size c fetch cons
	kafka.network:type=RequestMetrics,name=TotalTimeMs,request=	brok remo

Request total time	{Produce|FetchConsumer|FetchFollower}	time
Time the request waiting in the request queue	kafka.network:type=RequestMetrics,name=QueueTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Time the request being processed at the leader	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Time the request waits for the follower	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce|FetchConsumer|FetchFollower}	non-: wher
Time to send the response	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Number of messages the consumer lags behind the producer by	kafka.consumer:type=ConsumerFetcherManager,name=MaxLag,clientId=([-.\w]+)	
The average fraction of time the network processors are idle	kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent	betw
The average fraction of time the request handler threads are idle	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent	betw
Quota metrics per client-id	kafka.server:type={Produce|Fetch},client-id==([-.\w]+)	Two indic ms tl Ideal the d of th

New producer monitoring

The following metrics are available on new producer instances.

Metric/Attribute name	Description	Mbea
waiting-threads	The number of user threads blocked waiting for buffer memory to enqueue their records.	kafka.producer:type=pr ([-.\w]+)
buffer-total-bytes	The maximum amount of buffer memory the client can use (whether or not it is currently used).	kafka.producer:type=pr ([-.\w]+)
buffer-available-bytes	The total amount of buffer memory that is not being used (either unallocated or in the free list).	kafka.producer:type=pr ([-.\w]+)
bufferpool-wait-time	The fraction of time an appender waits for space allocation.	kafka.producer:type=pr ([-.\w]+)
batch-size-avg	The average number of bytes sent per partition per-request.	kafka.producer:type=pr

		([-.\w]+)
batch-size-max	The max number of bytes sent per partition per-request.	kafka.producer:type=pr ([-.\w]+)
compression-rate-avg	The average compression rate of record batches.	kafka.producer:type=pr ([-.\w]+)
record-queue-time-avg	The average time in ms record batches spent in the record accumulator.	kafka.producer:type=pr ([-.\w]+)
record-queue-time-max	The maximum time in ms record batches spent in the record accumulator.	kafka.producer:type=pr ([-.\w]+)
request-latency-avg	The average request latency in ms.	kafka.producer:type=pr ([-.\w]+)
request-latency-max	The maximum request latency in ms.	kafka.producer:type=pr ([-.\w]+)
record-send-rate	The average number of records sent per second.	kafka.producer:type=pr ([-.\w]+)
records-per-request-avg	The average number of records per request.	kafka.producer:type=pr ([-.\w]+)
record-retry-rate	The average per-second number of retried record sends.	kafka.producer:type=pr ([-.\w]+)
record-error-rate	The average per-second number of record sends that resulted in errors.	kafka.producer:type=pr ([-.\w]+)
record-size-max	The maximum record size.	kafka.producer:type=pr ([-.\w]+)
record-size-avg	The average record size.	kafka.producer:type=pr ([-.\w]+)
requests-in-flight	The current number of in-flight requests awaiting a response.	kafka.producer:type=pr ([-.\w]+)
metadata-age	The age in seconds of the current producer metadata being used.	kafka.producer:type=pr ([-.\w]+)
connection-close-rate	Connections closed per second in the window.	kafka.producer:type=pr ([-.\w]+)
connection-creation-rate	New connections established per second in the window.	kafka.producer:type=pr ([-.\w]+)
network-io-rate	The average number of network operations (reads or writes) on all connections per second.	kafka.producer:type=pr ([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second to all servers.	kafka.producer:type=pr ([-.\w]+)
request-rate	The average number of requests sent per second.	kafka.producer:type=pr ([-.\w]+)
request-size-avg	The average size of all requests in the window.	kafka.producer:type=pr ([-.\w]+)
request-size-max	The maximum size of any request sent in the window.	kafka.producer:type=pr ([-.\w]+)
incoming-byte-rate	Bytes/second read off all sockets.	kafka.producer:type=pr ([-.\w]+)

response-rate	Responses received sent per second.	kafka.producer:type=pr ([-.\w]+)
select-rate	Number of times the I/O layer checked for new I/O to perform per second.	kafka.producer:type=pr ([-.\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=pr ([-.\w]+)
io-wait-ratio	The fraction of time the I/O thread spent waiting.	kafka.producer:type=pr ([-.\w]+)
io-time-ns-avg	The average length of time for I/O per select call in nanoseconds.	kafka.producer:type=pr ([-.\w]+)
io-ratio	The fraction of time the I/O thread spent doing I/O.	kafka.producer:type=pr ([-.\w]+)
connection-count	The current number of active connections.	kafka.producer:type=pr ([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
request-rate	The average number of requests sent per second for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
request-size-avg	The average size of all requests in the window for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
request-size-max	The maximum size of any request sent in the window for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
incoming-byte-rate	The average number of responses received per second for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
request-latency-avg	The average request latency in ms for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
request-latency-max	The maximum request latency in ms for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
response-rate	Responses received sent per second for a node.	kafka.producer:type=pr metrics,client-id=([-.\w]
record-send-rate	The average number of records sent per second for a topic.	kafka.producer:type=pr metrics,client-id=([-.\w]
byte-rate	The average number of bytes sent per second for a topic.	kafka.producer:type=pr metrics,client-id=([-.\w]
compression-rate	The average compression rate of record batches for a topic.	kafka.producer:type=pr metrics,client-id=([-.\w]
record-retry-rate	The average per-second number of retried record sends for a topic.	kafka.producer:type=pr metrics,client-id=([-.\w]
record-error-rate	The average per-second number of record sends that resulted in errors for a topic.	kafka.producer:type=pr metrics,client-id=([-.\w]
produce-throttle-time-max	The maximum time in ms a request was throttled by a broker.	kafka.producer:type=pr metrics,client-id=([-.\w]

produce-throttle-time-avg	The average time in ms a request was throttled by a broker.	kafka.producer:type=pr metrics,client-id=([-.\w]-

We recommend monitoring GC time and other stats and various server stats such as CPU utilization, I/O service time, etc. On the client side, we recommend monitoring the message/byte rate (global and per topic), request rate/size/time, and on the consumer side, max lag in messages among all partitions and min fetch request rate. For a consumer to keep up, max lag needs to be less than a threshold and min fetch rate needs to be larger than 0.

Audit

The final alerting we do is on the correctness of the data delivery. We audit that every message that is sent is consumed by all consumers and measure the lag for this to occur. For important topics we alert if a certain completeness is not achieved in a certain time period. The details of this are discussed in KAFKA-260.

6.7 ZooKeeper

Stable version

The current stable branch is 3.4 and the latest release of that branch is 3.4.6, which is the one ZkClient 0.7 uses. ZkClient is the client layer Kafka uses to interact with ZooKeeper.

Operationalizing ZooKeeper

Operationally, we do the following for a healthy ZooKeeper installation:

- Redundancy in the physical/hardware/network layout: try not to put them all in the same rack, decent (but don't go nuts) hardware, try to keep redundant power and network paths, etc. A typical ZooKeeper ensemble has 5 or 7 servers, which tolerates 2 and 3 servers down, respectively. If you have a small deployment, then using 3 servers is acceptable, but keep in mind that you'll only be able to tolerate 1 server down in this case.
- I/O segregation: if you do a lot of write type traffic you'll almost definitely want the transaction logs on a dedicated disk group. Writes to the transaction log are synchronous (but batched for performance), and consequently, concurrent writes can significantly affect performance. ZooKeeper snapshots can be one such a source of concurrent writes, and ideally should be written on a disk group separate from the transaction log. Snapshots are writtent to disk asynchronously, so it is typically ok to share with the operating system and message log files. You can configure a server to use a separate disk group with the dataLogDir parameter.
- Application segregation: Unless you really understand the application patterns of other apps that you want to install on the same box, it can be a good idea to run ZooKeeper in isolation (though this can be a balancing act with the capabilities of the hardware).
- Use care with virtualization: It can work, depending on your cluster layout and read/write patterns and SLAs, but the tiny overheads introduced by the virtualization layer can add up and throw off ZooKeeper, as it can be very time sensitive
- ZooKeeper configuration: It's java, make sure you give it 'enough' heap space (We usually run them with 3-5G, but that's mostly due to the data set size we have here). Unfortunately we don't have a good formula for it, but keep in mind that allowing for more ZooKeeper state means that snapshots can become large, and large snapshots affect recovery time. In fact, if the snapshot becomes too large (a few gigabytes), then you may need to increase the initLimit parameter to give enough time for servers to recover and join the ensemble.

- Monitoring: Both JMX and the 4 letter words (4lw) commands are very useful, they do overlap in some cases (and in those cases we prefer the 4 letter commands, they seem more predictable, or at the very least, they work better with the LI monitoring infrastructure)
- Don't overbuild the cluster: large clusters, especially in a write heavy usage pattern, means a lot of intracluster communication (quorums on the writes and subsequent cluster member updates), but don't underbuild it (and risk swamping the cluster). Having more servers adds to your read capacity.

Overall, we try to keep the ZooKeeper system as small as will handle the load (plus standard growth capacity planning) and as simple as possible. We try not to do anything fancy with the configuration or application layout as compared to the official release as well as keep it as self contained as possible. For these reasons, we tend to skip the OS packaged versions, since it has a tendency to try to put things in the OS standard hierarchy, which can be 'messy', for want of a better way to word it.

7. Security

7.1 Security Overview

In release 0.9.0.0, the Kafka community added a number of features that, used either separately or together, increases security in a Kafka cluster. These features are considered to be of beta quality. The following security measures are currently supported:

1. Authentication of connections to brokers from clients (producers and consumers), other brokers and tools, using either SSL or SASL (Kerberos). SASL/PLAIN can also be used from release 0.10.0.0 onwards.
2. Authentication of connections from brokers to ZooKeeper
3. Encryption of data transferred between brokers and clients, between brokers, or between brokers and tools using SSL (Note that there is a performance degradation when SSL is enabled, the magnitude of which depends on the CPU type and the JVM implementation.)
4. Authorization of read / write operations by clients
5. Authorization is pluggable and integration with external authorization services is supported

It's worth noting that security is optional - non-secured clusters are supported, as well as a mix of authenticated, unauthenticated, encrypted and non-encrypted clients. The guides below explain how to configure and use the security features in both clients and brokers.

7.2 Encryption and Authentication using SSL

Apache Kafka allows clients to connect over SSL. By default SSL is disabled but can be turned on as needed.

1. **Generate SSL key and certificate for each Kafka broker**

   The first step of deploying HTTPS is to generate the key and the certificate for each machine in the cluster. You can use Java's keytool utility to accomplish this task. We will generate the key into a temporary keystore initially so that we can export and sign it later with CA.

   ```
   keytool -keystore server.keystore.jks -alias localhost -validity
   ```

   You need to specify two parameters in the above command:
   1. 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.
   2. validity: the valid time of the certificate in days.

   Ensure that common name (CN) matches exactly with the fully qualified domain name (FQDN) of the server. The client compares the CN with the DNS domain name to ensure that it is indeed connecting to the desired server, not the malicious one.

2. **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 -fil
```

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 -fil
```

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.

3. **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 -f
```

Then sign it with the CA:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cer
```

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
keytool -keystore server.keystore.jks -alias localhost -import -fi
```

The definitions of the parameters are the following:

1. keystore: the location of the keystore
2. ca-cert: the certificate of the CA
3. ca-key: the private key of the CA
4. ca-password: the passphrase of the CA

5. cert-file: the exported, unsigned certificate of the server
6. 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 3
#Step 2
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
keytool -keystore server.truststore.jks -alias CARoot -import -fil
keytool -keystore client.truststore.jks -alias CARoot -import -fil
#Step 3
keytool -keystore server.keystore.jks -alias localhost -certreq -f
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cer
keytool -keystore server.keystore.jks -alias CARoot -import -file
keytool -keystore server.keystore.jks -alias localhost -import -fi
```

4. **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.j
ssl.truststore.password=test1234
```

Optional settings that are worth considering:

1. 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.)
2. 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)
3. 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)
4. ssl.keystore.type=JKS
5. 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)
```

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 Chinta
issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddres
```

If the certificate does not show up or if there are any other error messages then your keystore is not setup properly.

5. **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.j
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:

1. ssl.provider (Optional). The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
2. 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.
3. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. It should list at least one of the protocols configured on the broker side
4. ssl.truststore.type=JKS
5. ssl.keystore.type=JKS

Examples using console-producer and console-consumer:

```
kafka-console-producer.sh --broker-list localhost:9093 --topic tes
kafka-console-consumer.sh --bootstrap-server localhost:9093 --topi
```

7.3 Authentication using SASL

1. **SASL configuration for Kafka brokers**

1. Select one or more supported mechanisms to enable in the broker. `GSSAPI` and `PLAIN` are the mechanisms currently supported in Kafka.
2. Add a JAAS config file for the selected mechanisms as described in the examples for setting up **GSSAPI (Kerberos)** or **PLAIN**.
3. 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.cor
```

4. 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)
```

5. Enable one or more SASL mechanisms in server.properties:

```
sasl.enabled.mechanisms=GSSAPI (,PLAIN)
```

6. 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)
```

7. 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**.
   Important notes:
   1. `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.
   2. `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 property `zookeeper.sasl.client` to the appropriate name (*e.g.*, `-Dzookeeper.sasl.client=ZkClient`).
   3. 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`).

2. 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:

1. Select a SASL mechanism for authentication.
2. 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.
3. 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.cor
```

4. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI (or PLAIN)
```

5. Follow the steps in **GSSAPI (Kerberos)** or **PLAIN** to configure SASL for the selected mechanism.

3. ## Authentication using SASL/Kerberos

   1. ### Prerequisites

      1. **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.
      2. **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/{hostn
         sudo /usr/sbin/kadmin.local -q "ktadd -k /etc/security/keytabs
         ```

      3. **Make sure all hosts can be reachable using hostnames** - it is a Kerberos requirement that all your hosts can be resolved with their FQDNs.

   2. ### Configuring Kafka Brokers

      1. 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";
         };
         ```

         `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.
      2. 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
         ```

3. Make sure the keytabs configured in the JAAS file are readable by the operating system user who is starting kafka broker.
4. 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
```

   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
```

3. **Configuring Kafka Clients**

To configure SASL authentication on the clients:

1. 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;
};
```

2. Pass the JAAS and optionally krb5 file locations as JVM parameters to each client JVM (see **here** for more details):

```
-Djava.security.krb5.conf=/etc/kafka/krb5.conf
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas
```

3. Make sure the keytabs configured in the kafka_client_jaas.conf are readable by the operating system user who is starting kafka client.
4. 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
```

4. ## 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.

1. **Configuring Kafka Brokers**

    1. 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 re
            username="admin"
            password="admin-secret"
            user_admin="admin-secret"
            user_alice="alice-secret";
        };
        ```

        This configuration defines two users (*admin* and *alice*). The properties `username` and `password` 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.
    2. Pass the JAAS config file location as JVM parameter to each Kafka broker:

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

    3. 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
        ```

2. **Configuring Kafka Clients**

    To configure SASL authentication on the clients:

    1. 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 re
            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*.
    2. Pass the JAAS config file location as JVM parameter to each client JVM:

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

    3. Configure the following properties in producer.properties or consumer.properties:

        ```
        security.protocol=SASL_SSL
        sasl.mechanism=PLAIN
        ```

3. **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**.

5. **Enabling multiple SASL mechanisms in a broker**

   1. Specify configuration for the login modules of all enabled mechanisms in the `KafkaServer` section 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 requir
          username="admin"
          password="admin-secret"
          user_admin="admin-secret"
          user_alice="alice-secret";
      };
      ```

   2. Enable the SASL mechanisms in server.properties:

      ```
      sasl.enabled.mechanisms=GSSAPI,PLAIN
      ```

   3. 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)
```

4. Follow the mechanism-specific steps in **GSSAPI (Kerberos)** and **PLAIN** to configure SASL for the enabled mechanisms.

6. **Modifying SASL mechanism in a Running Cluster**

SASL mechanism can be modified in a running cluster using the following sequence:

1. 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.
2. Restart clients using the new mechanism.
3. To change the mechanism of inter-broker communication (if this is required), set `sasl.mechanism.inter.broker.protocol` in server.properties to the new mechanism and incrementally bounce the cluster again.
4. To remove old mechanism (if this is required), remove the old mechanism from `sasl.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/@.*//,
```

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:

Option	Description	Default
--add	Indicates to the script that user is trying to add an acl.	
--remove	Indicates to the script that user is trying to remove an acl.	
--list	Indicates to the script that user is trying to list acls.	
--authorizer	Fully qualified class name of the authorizer.	kafka.security.auth.SimpleA
--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.	
--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.	
--allow-host	IP address from which principals listed in --allow-principal will have access.	if --allow-principal is specifie to * which translates to "all h
--deny-host	IP address from which principals listed in --deny-principal will be denied access.	if --deny-principal is specifie to * which translates to "all h
--operation	Operation that will be allowed or denied. Valid values are : Read, Write, Create, Delete, Alter, Describe, ClusterAction, All	All
--producer	Convenience option to add/remove acls for producer role. This will generate acls that allows WRITE, DESCRIBE on topic and CREATE on cluster.	
--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.	

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
  ```

  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
```

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:218
  ```

- **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
  ```

- **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:218
  ```

  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:218
  ```

  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://broke
```

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://broke
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 parameter `zookeeper.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.connect:
```

Run this to see the full list of parameters:

```
./bin/zookeeper-security-migration --help
```

7.6.3 Migrating the ZooKeeper ensemble

It is also necessary to enable authentication on the ZooKeeper ensemble. To do it, we need to perform a rolling restart of the server and set a few properties. Please refer to the ZooKeeper documentation for more detail:

1. Apache ZooKeeper documentation
2. Apache ZooKeeper wiki

8. Kafka Connect

8.1 Overview

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It makes it simple to quickly define *connectors* that move large collections of data into and out of Kafka. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export job can deliver data from Kafka topics into secondary storage and query systems or into batch systems for offline analysis. Kafka Connect features include:

- **A common framework for Kafka connectors** - Kafka Connect standardizes integration of other data systems with Kafka, simplifying connector development, deployment, and management
- **Distributed and standalone modes** - scale up to a large, centrally managed service supporting an entire organization or scale down to development, testing, and small production deployments
- **REST interface** - submit and manage connectors to your Kafka Connect cluster via an easy to use REST API
- **Automatic offset management** - with just a little information from connectors, Kafka Connect can manage the offset commit process automatically so connector developers do not need to worry about this error prone part of connector development
- **Distributed and scalable by default** - Kafka Connect builds on the existing group management protocol. More workers can be added to scale up a Kafka Connect cluster.
- **Streaming/batch integration** - leveraging Kafka's existing capabilities, Kafka Connect is an ideal solution for bridging streaming and batch data systems

8.2 User Guide

The quickstart provides a brief example of how to run a standalone version of Kafka Connect. This section describes how to configure, run, and manage Kafka Connect in more detail.

Running Kafka Connect

Kafka Connect currently supports two modes of execution: standalone (single process) and distributed. In standalone mode all work is performed in a single process. This configuration is simpler to setup and get started with and may be useful in situations where only one worker makes sense (e.g. collecting log files), but it does not benefit from some of the features of Kafka Connect such as fault tolerance. You can start a standalone process with the following command:

```
> bin/connect-standalone.sh config/connect-standalone.properties connector1.p
```

The first parameter is the configuration for the worker. This includes settings such as the Kafka connection parameters, serialization format, and how frequently to commit offsets. The provided example should work well with a local cluster running with the default configuration provided by `config/server.properties`. It will require tweaking to use with a different configuration or production deployment. The remaining parameters are connector configuration files. You may include as many as you want, but all will execute within the same process (on different threads). Distributed mode handles automatic balancing of work, allows you to scale up (or down) dynamically, and offers fault tolerance both in the active tasks and for configuration and offset commit data. Execution is very similar to standalone mode:

```
> bin/connect-distributed.sh config/connect-distributed.properties
```

The difference is in the class which is started and the configuration parameters which change how the Kafka Connect process decides where to store configurations, how to assign work, and where to store offsets and task statues. In the distributed mode, Kafka Connect stores the offsets, configs and task statuses in Kafka topics. It is recommended to manually create the topics for offset, configs and statuses in order to achieve the desired the number of partitions and replication factors. If the topics are not yet created when starting Kafka Connect, the topics will be auto created with default number of partitions and replication factor, which may not be best suited for its usage. In particular, the following configuration parameters are critical to set before starting your cluster:

- `group.id` (default `connect-cluster`) - unique name for the cluster, used in forming the Connect cluster group; note that this **must not conflict** with consumer group IDs

- `config.storage.topic` (default `connect-configs`) - topic to use for storing connector and task configurations; note that this should be a single partition, highly replicated topic. You may need to manually create the topic to ensure single partition for the config topic as auto created topics may have multiple partitions.
- `offset.storage.topic` (default `connect-offsets`) - topic to use for storing offsets; this topic should have many partitions and be replicated
- `status.storage.topic` (default `connect-status`) - topic to use for storing statuses; this topic can have multiple partitions and should be replicated

Note that in distributed mode the connector configurations are not passed on the command line. Instead, use the REST API described below to create, modify, and destroy connectors.

Configuring Connectors

Connector configurations are simple key-value mappings. For standalone mode these are defined in a properties file and passed to the Connect process on the command line. In distributed mode, they will be included in the JSON payload for the request that creates (or modifies) the connector. Most configurations are connector dependent, so they can't be outlined here. However, there are a few common options:

- `name` - Unique name for the connector. Attempting to register again with the same name will fail.
- `connector.class` - The Java class for the connector
- `tasks.max` - The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot achieve this level of parallelism.

The `connector.class` config supports several formats: the full name or alias of the class for this connector. If the connector is org.apache.kafka.connect.file.FileStreamSinkConnector, you can either specify this full name or use FileStreamSink or FileStreamSinkConnector to make the configuration a bit shorter. Sink connectors also have one additional option to control their input:

- `topics` - A list of topics to use as input for this connector

For any other options, you should consult the documentation for the connector.

REST API

Since Kafka Connect is intended to be run as a service, it also provides a REST API for managing connectors. By default this service runs on port 8083. The following are the currently supported endpoints:

- `GET /connectors` - return a list of active connectors
- `POST /connectors` - create a new connector; the request body should be a JSON object containing a string `name` field and a object `config` field with the connector configuration parameters
- `GET /connectors/{name}` - get information about a specific connector
- `GET /connectors/{name}/config` - get the configuration parameters for a specific connector
- `PUT /connectors/{name}/config` - update the configuration parameters for a specific connector
- `GET /connectors/{name}/status` - get current status of the connector, including if it is running, failed, paused, etc., which worker it is assigned to, error information if it has failed, and the state of all its tasks
- `GET /connectors/{name}/tasks` - get a list of tasks currently running for a connector
- `GET /connectors/{name}/tasks/{taskid}/status` - get current status of the task, including if it is running, failed, paused, etc., which worker it is assigned to, and error information if it has failed
- `PUT /connectors/{name}/pause` - pause the connector and its tasks, which stops message

processing until the connector is resumed
- `PUT /connectors/{name}/resume` - resume a paused connector (or do nothing if the connector is not paused)
- `POST /connectors/{name}/restart` - restart a connector (typically because it has failed)
- `POST /connectors/{name}/tasks/{taskId}/restart` - restart an individual task (typically because it has failed)
- `DELETE /connectors/{name}` - delete a connector, halting all tasks and deleting its configuration

Kafka Connect also provides a REST API for getting information about connector plugins:

- `GET /connector-plugins`- return a list of connector plugins installed in the Kafka Connect cluster. Note that the API only checks for connectors on the worker that handles the request, which means you may see inconsistent results, especially during a rolling upgrade if you add new connector jars
- `PUT /connector-plugins/{connector-type}/config/validate` - validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

8.3 Connector Development Guide

This guide describes how developers can write new connectors for Kafka Connect to move data between Kafka and other systems. It briefly reviews a few key concepts and then describes how to create a simple connector.

Core Concepts and APIs

Connectors and Tasks

To copy data between Kafka and another system, users create a `Connector` for the system they want to pull data from or push data to. Connectors come in two flavors: `SourceConnectors` import data from another system (e.g. `JDBCSourceConnector` would import a relational database into Kafka) and `SinkConnectors` export data (e.g. `HDFSSinkConnector` would export the contents of a Kafka topic to an HDFS file). `Connectors` do not perform any data copying themselves: their configuration describes the data to be copied, and the `Connector` is responsible for breaking that job into a set of `Tasks` that can be distributed to workers. These `Tasks` also come in two corresponding flavors: `SourceTask` and `SinkTask`. With an assignment in hand, each `Task` must copy its subset of the data to or from Kafka. In Kafka Connect, it should always be possible to frame these assignments as a set of input and output streams consisting of records with consistent schemas. Sometimes this mapping is obvious: each file in a set of log files can be considered a stream with each parsed line forming a record using the same schema and offsets stored as byte offsets in the file. In other cases it may require more effort to map to this model: a JDBC connector can map each table to a stream, but the offset is less clear. One possible mapping uses a timestamp column to generate queries incrementally returning new data, and the last queried timestamp can be used as the offset.

Streams and Records

Each stream should be a sequence of key-value records. Both the keys and values can have complex structure -- many primitive types are provided, but arrays, objects, and nested data structures can be represented as well. The runtime data format does not assume any particular serialization format; this conversion is handled internally by the framework. In addition to the key and value, records (both those generated by sources and those delivered to sinks) have associated stream IDs and offsets. These are used by the framework to periodically commit the offsets of data that have been processed so that in the event of failures, processing can resume from the last committed offsets, avoiding unnecessary reprocessing and duplication of events.

Dynamic Connectors

Not all jobs are static, so `Connector` implementations are also responsible for monitoring the external system for any changes that might require reconfiguration. For example, in the `JDBCSourceConnector` example, the `Connector` might assign a set of tables to each `Task`. When a new table is created, it must discover this so it can assign the new table to one of the `Tasks` by updating its configuration. When it notices a change that requires reconfiguration (or a change in the number of `Tasks`), it notifies the framework and the framework updates any corresponding `Tasks`.

Developing a Simple Connector

Developing a connector only requires implementing two interfaces, the `Connector` and `Task`. A simple example is included with the source code for Kafka in the `file` package. This connector is meant for use in standalone mode and has implementations of a `SourceConnector`/`SourceTask` to read each line of a file and emit it as a record and a `SinkConnector`/`SinkTask` that writes each record to a file. The rest of this section will walk through some code to demonstrate the key steps in creating a connector, but developers should also refer to the full example source code as many details are omitted for brevity.

Connector Example

We'll cover the `SourceConnector` as a simple example. `SinkConnector` implementations are very similar. Start by creating the class that inherits from `SourceConnector` and add a couple of fields that will store parsed configuration information (the filename to read from and the topic to send data to):

```
public class FileStreamSourceConnector extends SourceConnector {
    private String filename;
    private String topic;
```

The easiest method to fill in is `getTaskClass()`, which defines the class that should be instantiated in worker processes to actually read the data:

```
@Override
public Class<? extends Task> getTaskClass() {
    return FileStreamSourceTask.class;
}
```

We will define the `FileStreamSourceTask` class below. Next, we add some standard lifecycle methods, `start()` and `stop()`:

```
@Override
public void start(Map<String, String> props) {
    // The complete version includes error handling as well.
    filename = props.get(FILE_CONFIG);
    topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
    // Nothing to do since no background monitoring is required.
}
```

Finally, the real core of the implementation is in `getTaskConfigs()`. In this case we are only handling a single file, so even though we may be permitted to generate more tasks as per the `maxTasks` argument, we return a list with only one entry:

```
@Override
public List<Map<String, String>> getTaskConfigs(int maxTasks) {
    ArrayList>Map<String, String>> configs = new ArrayList<>();
    // Only one input stream makes sense.
    Map<String, String> config = new Map<>();
    if (filename != null)
```

```
        config.put(FILE_CONFIG, filename);
    config.put(TOPIC_CONFIG, topic);
    configs.add(config);
    return configs;
}
```

Although not used in the example, `SourceTask` also provides two APIs to commit offsets in the source system: `commit` and `commitRecord`. The APIs are provided for source systems which have an acknowledgement mechanism for messages. Overriding these methods allows the source connector to acknowledge messages in the source system, either in bulk or individually, once they have been written to Kafka. The `commit` API stores the offsets in the source system, up to the offsets that have been returned by `poll`. The implementation of this API should block until the commit is complete. The `commitRecord` API saves the offset in the source system for each `SourceRecord` after it is written to Kafka. As Kafka Connect will record offsets automatically, `SourceTask`s are not required to implement them. In cases where a connector does need to acknowledge messages in the source system, only one of the APIs is typically required. Even with multiple tasks, this method implementation is usually pretty simple. It just has to determine the number of input tasks, which may require contacting the remote service it is pulling data from, and then divvy them up. Because some patterns for splitting work among tasks are so common, some utilities are provided in `ConnectorUtils` to simplify these cases. Note that this simple example does not include dynamic input. See the discussion in the next section for how to trigger updates to task configs.

Task Example - Source Task

Next we'll describe the implementation of the corresponding `SourceTask`. The implementation is short, but too long to cover completely in this guide. We'll use pseudo-code to describe most of the implementation, but you can refer to the source code for the full example. Just as with the connector, we need to create a class inheriting from the appropriate base `Task` class. It also has some standard lifecycle methods:

```
public class FileStreamSourceTask extends SourceTask<Object, Object> {
    String filename;
    InputStream stream;
    String topic;

    public void start(Map<String, String> props) {
        filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
        stream = openOrThrowError(filename);
        topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    }

    @Override
    public synchronized void stop() {
        stream.close();
    }
```

These are slightly simplified versions, but show that that these methods should be relatively simple and the only work they should perform is allocating or freeing resources. There are two points to note about this implementation. First, the `start()` method does not yet handle resuming from a previous offset, which will be addressed in a later section. Second, the `stop()` method is synchronized. This will be necessary because `SourceTasks` are given a dedicated thread which they can block indefinitely, so they need to be stopped with a call from a different thread in the Worker. Next, we implement the main functionality of the task, the `poll()` method which gets events from the input system and returns a `List<SourceRecord>`:

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition = Collections.singletonMa
                Map<String, Object> sourceOffset = Collections.singletonMap("
```

```
                records.add(new SourceRecord(sourcePartition, sourceOffset, t
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling stop
        // 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(Collect
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 pub

    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.

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 t
        .addProcessor("PROCESS2", MyProcessor2::new /* the ProcessorSupplier t
        .addProcessor("PROCESS3", MyProcessor3::new /* the ProcessorSupplier t

        .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 processo
        .addStateStore(Stores.create("COUNTS").withStringKeys().withStringVal
        .addProcessor("PROCESS2", MyProcessor3::new /* the ProcessorSupplier t
        .addProcessor("PROCESS3", MyProcessor3::new /* the ProcessorSupplier t

        // 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
    HoppingWindows.of("counts").with(5000L).every(1000L), // intervals in
);

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**.