You're viewing documentation for an older version of Kafka - check out our current documentation here.

A distributed streaming platform

Download

@apachekafka

# Documentation

## Kafka 0.8.2

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

You're viewing documentation for an older version of Kafka - check out our current documentation here.

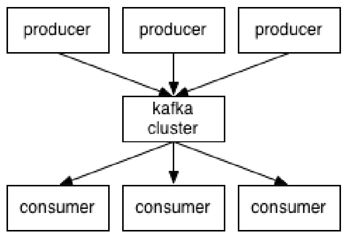
Kafka™ is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

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.

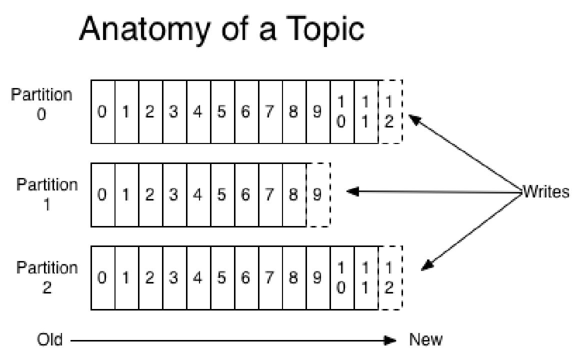
Download

@apachekafka

## 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.

HOME
INTRODUCTION
QUICKSTART
USE CASES
DOCUMENTATION
PERFORMANCE
POWERED BY
PROJECT INFO
ECOSYSTEM
CLIENTS
EVENTS
CONTACT US
APACHE

## 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 cluster of consumers instead of a single process.

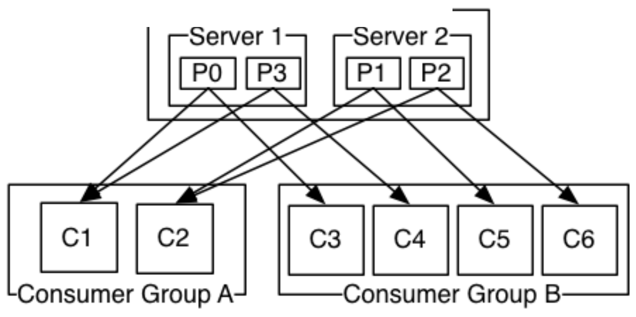
Kafka has stronger ordering guarantees than a traditional messaging system, too.

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.

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

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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.



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.

### 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 end up doing stage-wise processing of data where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption. For example a processing flow for article recommendation might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage might attempt to match this content to users. This creates a graph of real-time data flow out of the individual topics. Storm and Samza are popular frameworks for implementing these kinds of transformations.

### 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 0.8.2.0 release and un-tar it.

```
> tar -xzf kafka_2.10-0.8.2.0.tgz
> cd kafka_2.10-0.8.2.0
```

### Step 2: Start the server

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

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

You're viewing documentation for an older version of Kafka - check out our current documentation here.

```
> bin/kafka-server-start.sh config/server.properties
[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)
...
```

### Step 3: Create a topic

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

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

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

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

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

### Step 4: Send some messages

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

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

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

### Step 5: Start a consumer

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

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
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
    port=9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    port=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 &
...
```

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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

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

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

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

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

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

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

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

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

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

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

Now let's consume these messages:

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

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

```
> ps | grep server-1.properties
7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bin/java...
> kill -9 7564
```

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

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

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

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

## 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 0.8.1 to 0.8.2.0

0.8.2.0 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.

You're viewing documentation for an older version of Kafka - check out our current documentation here.

0.8, the release in which added replication, was our first backwards-incompatible release: major changes were made to the API, ZooKeeper data structures, and protocol, and configuration. The upgrade from 0.7 to 0.8.x requires a special tool for migration. This migration can be done without downtime.

## 2. API

We are in the process of rewritting the JVM clients for Kafka. As of 0.8.2 Kafka includes a newly rewritten Java producer. The next release will include an equivalent Java consumer. These new clients are meant to supplant the existing Scala clients, but for compatability they will co-exist for some time. These clients are available in a seperate jar with minimal dependencies, while the old Scala clients remain packaged with the server.

### 2.1 Producer API

As of the 0.8.2 release 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 maven co-ordinates:

```
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>0.8.2.0</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 High Level Consumer API

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

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

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

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

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

  /**
   *  Create a list of message streams for topics matching a wildcard, using the default decoder, with one stream.
   */
  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter topicFilter);

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

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

You're viewing documentation for an older version of Kafka - check out our current documentation here.

## 2.3 Simple Consumer API

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

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

  /**
   *  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();
}
```

Download

@apachekafka

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.

## 2.4 Kafka Hadoop Consumer API

Providing a horizontally scalable solution for aggregating and loading data into Hadoop was one of our basic use cases. To support this use case, we provide a Hadoop-based consumer which spawns off many map tasks to pull data from the Kafka cluster in parallel. This provides extremely fast pull-based Hadoop data load capabilities (we were able to fully saturate the network with only a handful of Kafka servers).

Usage information on the hadoop consumer can be found here.

## 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.

| PROPERTY | DEFAULT | DESCRIPTION |
|---|---|---|
| broker.id | | Each broker is uniquely identified by a non-negative integer id. This id serves as the broker's "name" and allows the broker to be moved to a different host/port without confusing consumers. You can choose any number you like so long as it is unique. |
| log.dirs | /tmp/kafka-logs | A comma-separated list of one or more directories in which Kafka data is stored. Each new partition that is created will be placed in the directory which currently has the fewest partitions. |
| port | 9092 | The port on which the server accepts client connections. |
| zookeeper.connect | null | Specifies the ZooKeeper connection string in the form `hostname:port`, where hostname and port are the host and port for a node in your ZooKeeper cluster. To allow connecting through other ZooKeeper nodes when that host is down you can also specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:port3`.<br><br>ZooKeeper also allows you to add a "chroot" path which will make all kafka data for this cluster appear under a particular path. This is a way to setup multiple Kafka clusters or other applications on the same ZooKeeper cluster. To do this give a connection string in the form `hostname1:port1,hostname2:port2,hostname3:port3/chroot/path` which would put all this cluster's data under the path `/chroot/path`. Note that consumers must use the same connection string. |
| message.max.bytes | 1000000 | The maximum size of a message that the server can receive. It is important that this property be in sync with the maximum fetch size your consumers use or else an unruly producer will be able to publish messages too large for consumers to consume. |
| num.network.threads | 3 | The number of network threads that the server uses for handling network requests. You probably don't need to change this. |
| num.io.threads | 8 | The number of I/O threads that the server uses for executing requests. You should have at least as many threads as you have disks. |
| background.threads | 10 | The number of threads to use for various background processing tasks such as file deletion. You should not need to change this. |
| queued.max.requests | 500 | The number of requests that can be queued up for processing by the I/O threads before the network threads stop reading in new requests. |
| host.name | null | 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, and publish one to ZK. |
| advertised.host.name | null | If this is set this is the hostname that will be given out to producers, consumers, and other brokers to connect to. |

ıeeds to be set if this port

You're viewing documentation for an older version of Kafka - check out our current documentation here.

| | | |
|---|---|---|
| socket.send.buffer.bytes | 100 * 1024 | The SO_SNDBUFF buffer the server prefers for socket connections. |
| socket.receive.buffer.bytes | 100 * 1024 | The SO_RCVBUFF buffer the server prefers for socket connections. |
| socket.request.max.bytes | 100 * 1024 * 1024 | The maximum request size the server will allow. This prevents the server from running out of memory and should be smaller than the Java heap size. |
| num.partitions | 1 | The default number of partitions per topic if a partition count isn't given at topic creation time. |
| log.segment.bytes | 1024 * 1024 * 1024 | The log for a topic partition is stored as a directory of segment files. This setting controls the size to which a segment file will grow before a new segment is rolled over in the log. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.roll.{ms,hours} | 24 * 7 hours | This setting will force Kafka to roll a new log segment even if the log.segment.bytes size has not been reached. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.cleanup.policy | delete | This can take either the value *delete* or *compact*. If *delete* is set, log segments will be deleted when they reach the size or time limits set. If *compact* is set log compaction will be used to clean out obsolete records. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.retention.{ms,minutes,hours} | 7 days | The amount of time to keep a log segment before it is deleted, i.e. the default data retention window for all topics. Note that if both log.retention.minutes and log.retention.bytes are both set we delete a segment when either limit is exceeded. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.retention.bytes | -1 | The amount of data to retain in the log for each topic-partitions. Note that this is the limit per-partition so multiply by the number of partitions to get the total data retained for the topic. Also note that if both log.retention.hours and log.retention.bytes are both set we delete a segment when either limit is exceeded. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.retention.check.interval.ms | 5 minutes | The period with which we check whether any log segment is eligible for deletion to meet the retention policies. |
| log.cleaner.enable | false | This configuration must be set to true for log compaction to run. |
| log.cleaner.threads | 1 | The number of threads to use for cleaning logs in log compaction. |
| log.cleaner.io.max.bytes.per.second | Double.MaxValue | The maximum amount of I/O the log cleaner can do while performing log compaction. This setting allows setting a limit for the cleaner to avoid impacting live request serving. |
| log.cleaner.dedupe.buffer.size | 500*1024*1024 | The size of the buffer the log cleaner uses for indexing and deduplicating logs during cleaning. Larger is better provided you have sufficient memory. |
| log.cleaner.io.buffer.size | 512*1024 | The size of the I/O chunk used during log cleaning. You probably don't need to change this. |
| log.cleaner.io.buffer.load.factor | 0.9 | The load factor of the hash table used in log cleaning. You probably don't need to change this. |
| log.cleaner.backoff.ms | 15000 | The interval between checks to see if any logs need cleaning. |
| log.cleaner.min.cleanable.ratio | 0.5 | This configuration controls how frequently the log compactor will attempt to clean the log (assuming log compaction is enabled). By default we will avoid cleaning a log where more than 50% of the log has been compacted. This ratio bounds the maximum space wasted in the log by duplicates (at 50% at most 50% of the log could be duplicates). A higher ratio will mean fewer, more efficient cleanings but will mean more wasted space in the log. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.cleaner.delete.retention.ms | 1 day | The amount of time to retain delete tombstone markers for log compacted topics. This setting also gives a bound on the time in which a consumer must complete a read if they begin from offset 0 to ensure that they get a valid snapshot of the final stage (otherwise delete tombstones may be collected before they complete their scan). This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.index.size.max.bytes | 10 * 1024 * 1024 | The maximum size in bytes we allow for the offset index for each log segment. Note that we will always pre-allocate a sparse file with this much space and shrink it down when the log rolls. If the index fills up we will roll a new log segment even if we haven't reached the log.segment.bytes limit. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| log.index.interval.bytes | 4096 | The byte interval at which we add an entry to the offset index. When executing a fetch request the server must do a linear scan for up to this many bytes to find the correct position in the log to begin and end the fetch. So setting this value to be larger will mean larger index files (and a bit more memory usage) but less scanning. However the server will never add more than one index entry per log append (even if more than log.index.interval worth of messages are appended). In general you probably don't need to mess with this value. |
| log.flush.interval.messages | Long.MaxValue | The number of messages written to a log partition before we force an fsync on the log. Setting this lower will sync data to disk more often but will have a major impact on performance. We generally recommend that people make use of replication for durability rather than depending on single-server fsync, however this setting can be used to be extra certain. |
| log.flush.scheduler.interval.ms | Long.MaxValue | The frequency in ms that the log flusher checks whether any log is eligible to be flushed to disk. |
| log.flush.interval.ms | Long.MaxValue | The maximum time between fsync calls on the log. If used in conjuction with log.flush.interval.messages the log will be flushed when either criteria is met. |
| log.delete.delay.ms | 60000 | The period of time we hold log files around after they are removed from the in-memory segment index. This period of time allows any in-progress reads to complete uninterrupted without locking. You generally don't need to change this. |
| log.flush.offset.checkpoint.interval.ms | 60000 | The frequency with which we checkpoint the last flush point for logs for recovery. You should not need to change this. |
| log.segment.delete.delay.ms | 60000 | the amount of time to wait before deleting a file from the filesystem. |
| auto.create.topics.enable | true | Enable auto creation of topic on the server. If this is set to true then attempts to produce data or fetch metadata for a non-existent topic will automatically create it with the default replication factor and number of partitions. |
| controller.socket.timeout.ms | 30000 | The socket timeout for commands from the partition management controller to the replicas. |
| controller.message.queue.size | Int.MaxValue | The buffer size for controller-to-broker-channels |
| default.replication.factor | 1 | The default replication factor for automatically created topics. |
| replica.lag.time.max.ms | 10000 | If a follower hasn't sent any fetch requests for this window of time, the leader will remove the follower from ISR (in-sync replicas) and treat it as dead. |

Download

@apachekafka

You're viewing documentation for an older version of Kafka - check out our current documentation here.

and treat it as dead.

Download

@apachekafka

| | | |
|---|---|---|
| replica.socket.timeout.ms | 30 * 1000 | The socket timeout for network requests to the leader for replicating data. |
| replica.socket.receive.buffer.bytes | 64 * 1024 | The socket receive buffer for network requests to the leader for replicating data. |
| replica.fetch.max.bytes | 1024 * 1024 | The number of byes of messages to attempt to fetch for each partition in the fetch requests the replicas send to the leader. |
| replica.fetch.wait.max.ms | 500 | The maximum amount of time to wait time for data to arrive on the leader in the fetch requests sent by the replicas to the leader. |
| replica.fetch.min.bytes | 1 | Minimum bytes expected for each fetch response for the fetch requests from the replica to the leader. If not enough bytes, wait up to replica.fetch.wait.max.ms for this many bytes to arrive. |
| num.replica.fetchers | 1 | Number of threads used to replicate messages from leaders. Increasing this value can increase the degree of I/O parallelism in the follower broker. |
| replica.high.watermark.checkpoint.interval.ms | 5000 | The frequency with which each replica saves its high watermark to disk to handle recovery. |
| fetch.purgatory.purge.interval.requests | 1000 | The purge interval (in number of requests) of the fetch request purgatory. |
| producer.purgatory.purge.interval.requests | 1000 | The purge interval (in number of requests) of the producer request purgatory. |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the server fails to heartbeat to ZooKeeper within this period of time it is considered dead. If you set this too low the server may be falsely considered dead; if you set it too high it may take too long to recognize a truly dead server. |
| zookeeper.connection.timeout.ms | 6000 | The maximum amount of time that the client waits to establish a connection to zookeeper. |
| zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader. |
| controlled.shutdown.enable | true | Enable controlled shutdown of the broker. If enabled, the broker will move all leaders on it to some other brokers before shutting itself down. This reduces the unavailability window during shutdown. |
| controlled.shutdown.max.retries | 3 | Number of retries to complete the controlled shutdown successfully before executing an unclean shutdown. |
| controlled.shutdown.retry.backoff.ms | 5000 | Backoff time between shutdown retries. |
| auto.leader.rebalance.enable | true | If this is enabled the controller will automatically try to balance leadership for partitions among the brokers by periodically returning leadership to the "preferred" replica for each partition if it is available. |
| leader.imbalance.per.broker.percentage | 10 | The percentage of leader imbalance allowed per broker. The controller will rebalance leadership if this ratio goes above the configured value per broker. |
| leader.imbalance.check.interval.seconds | 300 | The frequency with which to check for leader imbalance. |
| offset.metadata.max.bytes | 4096 | The maximum amount of metadata to allow clients to save with their offsets. |
| max.connections.per.ip | Int.MaxValue | The maximum number of connections that a broker allows from each ip address. |
| max.connections.per.ip.overrides | | Per-ip or hostname overrides to the default maximum number of connections. |
| connections.max.idle.ms | 600000 | Idle connections timeout: the server socket processor threads close the connections that idle more than this. |
| log.roll.jitter.{ms,hours} | 0 | The maximum jitter to subtract from logRollTimeMillis. |
| num.recovery.threads.per.data.dir | 1 | The number of threads per data directory to be used for log recovery at startup and flushing at shutdown. |
| unclean.leader.election.enable | true | 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. |
| delete.topic.enable | false | Enable delete topic. |
| offsets.topic.num.partitions | 50 | The number of partitions for the offset commit topic. Since changing this after deployment is currently unsupported, we recommend using a higher setting for production (e.g., 100-200). |
| offsets.topic.retention.minutes | 1440 | Offsets that are older than this age will be marked for deletion. The actual purge will occur when the log cleaner compacts the offsets topic. |
| offsets.retention.check.interval.ms | 600000 | The frequency at which the offset manager checks for stale offsets. |
| offsets.topic.replication.factor | 3 | The replication factor for the offset commit topic. A higher setting (e.g., three or four) is recommended in order to ensure higher availability. If the offsets topic is created when fewer brokers than the replication factor then the offsets topic will be created with fewer replicas. |
| offsets.topic.segment.bytes | 104857600 | Segment size for the offsets topic. Since it uses a compacted topic, this should be kept relatively low in order to facilitate faster log compaction and loads. |
| offsets.load.buffer.size | 5242880 | An offset load occurs when a broker becomes the offset manager for a set of consumer groups (i.e., when it becomes a leader for an offsets topic partition). This setting corresponds to the batch size (in bytes) to use when reading from the offsets segments when loading offsets into the offset manager's cache. |
| offsets.commit.required.acks | -1 | The number of acknowledgements that are required before the offset commit can be accepted. This is similar to the producer's acknowledgement setting. In general, the default should not be overridden. |
| offsets.commit.timeout.ms | 5000 | The offset commit will be delayed until this timeout or the required number of replicas have received the offset commit. This is similar to the producer request timeout. |

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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

### Topic-level configuration

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

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

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

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

To remove an override you can do

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

Download

@apachekafka

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". This string designates the retention policy to use on old log segments. The default policy ("delete") will discard old segments when their retention time or size limit has been reached. The "compact" setting will enable log compaction on the topic. |
| delete.retention.ms | 86400000 (24 hours) | log.cleaner.delete.retention.ms | The amount of time to retain delete tombstone markers for log compacted topics. This setting also gives a bound on the time in which a consumer must complete a read if they begin from offset 0 to ensure that they get a valid snapshot of the final stage (otherwise delete tombstones may be collected before they complete their scan). |
| flush.messages | None | log.flush.interval.messages | This setting allows specifying an interval at which we will force an fsync of data written to the log. For example if this was set to 1 we would fsync after every message; if it were 5 we would fsync after every five messages. In general we recommend you not set this and use replication for durability and allow the operating system's background flush capabilities as it is more efficient. This setting can be overridden on a per-topic basis (see the per-topic configuration section). |
| flush.ms | None | log.flush.interval.ms | This setting allows specifying a time interval at which we will force an fsync of data written to the log. For example if this was set to 1000 we would fsync after 1000 ms had passed. In general we recommend you not set this and use replication for durability and allow the operating system's background flush capabilities as it is more efficient. |
| index.interval.bytes | 4096 | log.index.interval.bytes | This setting controls how frequently Kafka adds an index entry to it's offset index. The default setting ensures that we index a message roughly every 4096 bytes. More indexing allows reads to jump closer to the exact position in the log but makes the index larger. You probably don't need to change this. |
| max.message.bytes | 1,000,000 | message.max.bytes | This is largest message size Kafka will allow to be appended to this topic. Note that if you increase this size you must also increase your consumer's fetch size so they can fetch messages this large. |
| min.cleanable.dirty.ratio | 0.5 | log.cleaner.min.cleanable.ratio | This configuration controls how frequently the log compactor will attempt to clean the log (assuming log compaction is enabled). By default we will avoid cleaning a log where more than 50% of the log has been compacted. This ratio bounds the maximum space wasted in the log by duplicates (at 50% at most 50% of the log could be duplicates). A higher ratio will mean fewer, more efficient cleanings but will mean more wasted space in the log. |
| min.insync.replicas | 1 | min.insync.replicas | When a producer sets request.required.acks to -1, min.insync.replicas specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).<br>When used together, min.insync.replicas and request.required.acks allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of 3, set min.insync.replicas to 2, and produce with request.required.acks of -1. This will ensure that the producer raises an exception if a majority of replicas do not receive a write. |
| retention.bytes | None | log.retention.bytes | This configuration controls the maximum size a log can grow to before we will discard old log segments to free up space if we are using the "delete" retention policy. By default there is no size limit only a time limit. |
| retention.ms | 7 days | log.retention.minutes | This configuration controls the maximum time we will retain a log before we will discard old log segments to free up space if we are using the "delete" retention policy. This represents an SLA on how soon consumers must read their data. |
| segment.bytes | 1 GB | log.segment.bytes | This configuration controls the segment file size for the log. Retention and cleaning is always done a file at a time so a larger segment size means fewer files but less granular control over retention. |
| segment.index.bytes | 10 MB | log.index.size.max.bytes | This configuration controls the size of the index that maps offsets to file positions. We preallocate this index file and shrink it only after log rolls. You generally should not need to change this setting. |
| segment.ms | 7 days | log.roll.hours | This configuration controls the period of time after which Kafka will force the log to roll even if the segment file isn't full to ensure that retention can delete or compact old data. |
| segment.jitter.ms | 0 | log.roll.jitter.{ms,hours} | The maximum jitter to subtract from logRollTimeMillis. |

## 3.2 Consumer Configs

The essential consumer configurations are the following:

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

You're viewing documentation for an older version of Kafka - check out our current documentation here.

| | | |
|---|---|---|
| group.id | | A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group. |
| zookeeper.connect | | Specifies the ZooKeeper connection string in the form `hostname:port` where host and port are the host and port of a ZooKeeper server. To allow connecting through other ZooKeeper nodes when that ZooKeeper machine is down you can also specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:port3` .<br><br>The server may also have a ZooKeeper chroot path as part of it's ZooKeeper connection string which puts its data under some path in the global ZooKeeper namespace. If so the consumer should use the same chroot path in its connection string. For example to give a chroot path of `/chroot/path` you would give the connection string as `hostname1:port1,hostname2:port2,hostname3:port3/chroot/path` . |
| consumer.id | null | Generated automatically if not set. |
| socket.timeout.ms | 30 * 1000 | The socket timeout for network requests. The actual timeout set will be max.fetch.wait + socket.timeout.ms. |
| socket.receive.buffer.bytes | 64 * 1024 | The socket receive buffer for network requests |
| fetch.message.max.bytes | 1024 * 1024 | The number of byes of messages to attempt to fetch for each topic-partition in each fetch request. These bytes will be read into memory for each partition, so this helps control the memory used by the consumer. The fetch request 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. |
| 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 already fetched by the consumer. This committed offset will be used when the process fails as the position from which the new consumer will begin. |
| auto.commit.interval.ms | 60 * 1000 | The frequency in ms that the consumer offsets are committed to zookeeper. |
| queued.max.message.chunks | 2 | Max number of message chunks buffered for consumption. Each chunk can be up to fetch.message.max.bytes. |
| rebalance.max.retries | 4 | When a new consumer joins a consumer group the set of consumers attempt to "rebalance" the load to assign partitions to each consumer. If the set of consumers changes while this assignment is taking place the rebalance will fail and retry. This setting controls the maximum number of attempts before giving up. |
| fetch.min.bytes | 1 | 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. |
| fetch.wait.max.ms | 100 | The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy fetch.min.bytes |
| rebalance.backoff.ms | 2000 | Backoff time between retries during rebalance. |
| refresh.leader.backoff.ms | 200 | Backoff time to wait before trying to determine the leader of a partition that has just lost its leader. |
| auto.offset.reset | largest | What to do when there is no initial offset in ZooKeeper or if an offset is out of range:<br>* smallest : automatically reset the offset to the smallest offset<br>* largest : automatically reset the offset to the largest offset<br>* anything else: throw exception to the consumer |
| consumer.timeout.ms | -1 | Throw a timeout exception to the consumer if no message is available for consumption after the specified interval |
| exclude.internal.topics | true | Whether messages from internal topics (such as offsets) should be exposed to the consumer. |
| partition.assignment.strategy | range | Select a strategy for assigning partitions to consumer streams. Possible values: range, roundrobin. |
| client.id | group id value | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur. |
| zookeeper.connection.timeout.ms | 6000 | The max time that the client waits while establishing a connection to zookeeper. |
| zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader |
| offsets.storage | zookeeper | Select where offsets should be stored (zookeeper or kafka). |
| offsets.channel.backoff.ms | 1000 | The backoff period when reconnecting the offsets channel or retrying failed offset fetch/commit requests. |
| offsets.channel.socket.timeout.ms | 10000 | Socket timeout when reading responses for offset fetch/commit requests. This timeout is also used for ConsumerMetadata requests that are used to query for the offset manager. |
| offsets.commit.max.retries | 5 | Retry the offset commit up to this many times on failure. This retry count only applies to offset commits during shut-down. It does not apply to commits originating from the auto-commit thread. It also does not apply to attempts to query for the offset coordinator before committing offsets. i.e., if a consumer metadata request fails for any reason, it will be retried and that retry does not count toward this limit. |
| dual.commit.enabled | true | If you are using "kafka" as offsets.storage, you can dual commit offsets to ZooKeeper (in addition to Kafka). This is required during migration from zookeeper-based offset storage to kafka-based offset storage. With respect to any given consumer group, it is safe to turn this off after all instances within that group have been migrated to the new version that commits offsets to the broker (instead of directly to ZooKeeper). |
| partition.assignment.strategy | range | Select between the "range" or "roundrobin" strategy for assigning partitions to consumer streams.<br><br>The round-robin partition assignor lays out all the available partitions and all the available consumer threads. It then proceeds to do a round-robin assignment from partition to consumer thread. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts will be within a delta of exactly one across all consumer threads.) Round-robin assignment is permitted only if: (a) Every topic has the same number of streams within a consumer instance (b) The set of subscribed topics is identical for every consumer instance within the group. |

Download

@apachekafka

You're viewing documentation for an older version of Kafka - check out our current documentation here.

| | | |
|---|---|---|
| | | ler and the consumer ns (threads) to determine will have one extra partition. |

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

## 3.3 Producer Configs

Essential configuration properties for the producer include:

- `metadata.broker.list`
- `request.required.acks`
- `producer.type`
- `serializer.class`

| PROPERTY | DEFAULT | DESCRIPTION |
|---|---|---|
| metadata.broker.list | | This is for bootstrapping and the producer will only use it for getting metadata (topics, partitions and replicas). The socket connections for sending the actual data will be established based on the broker information returned in the metadata. The format is host1:port1,host2:port2, and the list can be a subset of brokers or a VIP pointing to a subset of brokers. |
| request.required.acks | 0 | This value controls when a produce request is considered completed. Specifically, how many other brokers must have committed the data to their log and acknowledged this to the leader? Typical values are<br>• 0, which means that the producer never waits for an acknowledgement from the broker (the same behavior as 0.7). This option provides the lowest latency but the weakest durability guarantees (some data will be lost when a server fails).<br>• 1, which means that the producer gets an acknowledgement after the leader replica has received the data. This option provides better durability as the client waits until the server acknowledges the request as successful (only messages that were written to the now-dead leader but not yet replicated will be lost).<br>• -1, The producer gets an acknowledgement after all in-sync replicas have received the data. This option provides the greatest level of durability. However, it does not completely eliminate the risk of message loss because the number of in sync replicas may, in rare cases, shrink to 1. If you want to ensure that some minimum number of replicas (typically a majority) receive a write, then you must set the topic-level min.insync.replicas setting. Please read the Replication section of the design documentation for a more in-depth discussion. |
| request.timeout.ms | 10000 | The amount of time the broker will wait trying to meet the request.required.acks requirement before sending back an error to the client. |
| producer.type | sync | This parameter specifies whether the messages are sent asynchronously in a background thread. Valid values are (1) async for asynchronous send and (2) sync for synchronous send. By setting the producer to async we allow batching together of requests (which is great for throughput) but open the possibility of a failure of the client machine dropping unsent data. |
| serializer.class | kafka.serializer.DefaultEncoder | The serializer class for messages. The default encoder takes a byte[] and returns the same byte[]. |
| key.serializer.class | | The serializer class for keys (defaults to the same as for messages if nothing is given). |
| partitioner.class | kafka.producer.DefaultPartitioner | The partitioner class for partitioning messages amongst sub-topics. The default partitioner is based on the hash of the key. |
| compression.codec | none | This parameter allows you to specify the compression codec for all data generated by this producer. Valid values are "none", "gzip" and "snappy". |
| compressed.topics | null | This parameter allows you to set whether compression should be turned on for particular topics. If the compression codec is anything other than NoCompressionCodec, enable compression only for specified topics if any. If the list of compressed topics is empty, then enable the specified compression codec for all topics. If the compression codec is NoCompressionCodec, compression is disabled for all topics |
| message.send.max.retries | 3 | This property will cause the producer to automatically retry a failed send request. This property specifies the number of retries when such failures occur. Note that setting a non-zero value here can lead to duplicates in the case of network errors that cause a message to be sent but the acknowledgement to be lost. |
| retry.backoff.ms | 100 | Before each retry, the producer refreshes the metadata of relevant topics to see if a new leader has been elected. Since leader election takes a bit of time, this property specifies the amount of time that the producer waits before refreshing the metadata. |
| topic.metadata.refresh.interval.ms | 600 * 1000 | The producer generally refreshes the topic metadata from brokers when there is a failure (partition missing, leader not available...). It will also poll regularly (default: every 10min so 600000ms). If you set this to a negative value, metadata will only get refreshed on failure. If you set this to zero, the metadata will get refreshed after each message sent (not recommended). Important note: the refresh happen only AFTER the message is sent, so if the producer never sends a message the metadata is never refreshed |
| queue.buffering.max.ms | 5000 | Maximum time to buffer data when using async mode. For example a setting of 100 will try to batch together 100ms of messages to send at once. This will improve throughput but adds message delivery latency due to the buffering. |
| queue.buffering.max.messages | 10000 | The maximum number of unsent messages that can be queued up the producer when using async mode before either the producer must be blocked or data must be dropped. |
| queue.enqueue.timeout.ms | -1 | The amount of time to block before dropping messages when running in async mode and the buffer has reached queue.buffering.max.messages. If set to 0 events will be enqueued immediately or dropped if the queue is full (the producer send call will never block). If set to -1 the producer will block indefinitely and never willingly drop a send. |
| batch.num.messages | 200 | The number of messages to send in one batch when using async mode. The producer will wait until either this number of messages are ready to send or queue.buffer.max.ms is reached. |
| send.buffer.bytes | 100 * 1024 | Socket write buffer size |
| client.id | "" | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. |

More details about producer configuration can be found in the scala class `kafka.producer.ProducerConfig`.

Download

@apachekafka

You're viewing documentation for an older version of Kafka - check out our current documentation here.

We are working on a replacement for our existing producer. The code is available in trunk now and can be considered beta quality. Below is the configuration for the new producer.

| NAME | TYPE | DEFAULT | IMPORTANCE | DESCRIPTION |
|---|---|---|---|---|
| bootstrap.servers | list | | high | A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. Data will be load balanced over 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). If no server in this list is available sending data will fail until on becomes available. |
| acks | string | 1 | high | The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are common:<br>• `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.<br>• `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.<br>• `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.<br>• Other settings such as `acks=2` are also possible, and will require the given number of acknowledgements but this is generally less useful. |
| buffer.memory | long | 33554432 | high | 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 either block or throw an exception based on the preference specified by `block.on.buffer.full` .<br><br>This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests. |
| compression.type | string | none | high | The compression type for all data generated by the producer. The default is none (i.e. no compression). Valid values are `none` , `gzip` , or `snappy` . Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression). |
| retries | int | 0 | high | Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries will potentially change the ordering of records because if two records are sent to a single partition, and the first fails and is retried but the second succeeds, then the second record may appear first. |
| batch.size | int | 16384 | medium | 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.<br><br>No attempt will be made to batch records larger than this size.<br><br>Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.<br><br>A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records. |
| client.id | string | | medium | The 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 with the request. The application can set any string it wants as this has no functional purpose other than in logging and metrics. |
| linger.ms | long | 0 | medium | The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding 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. |
| max.request.size | int | 1048576 | medium | The maximum size of a request. 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. |
| receive.buffer.bytes | int | 32768 | medium | The size of the TCP receive buffer to use when reading data |
| send.buffer.bytes | int | 131072 | medium | The size of the TCP send buffer to use when sending data |
| timeout.ms | int | 30000 | medium | The configuration controls the maximum amount of time the server will wait for |

Download

@apachekafka

You're viewing documentation for an older version of Kafka - check out our current documentation here.

| | | | | |
|---|---|---|---|---|
| | | | | t requirements the quested number of or will be returned. This timeout is measured on the server side and does not include the network latency of the request. |
| block.on.buffer.full | boolean | true | low | When our memory buffer is exhausted we must either stop accepting new records (block) or throw errors. By default this setting is true and we block, however in some scenarios blocking is not desirable and it is better to immediately give an error. Setting this to `false` will accomplish that: the producer will throw a BufferExhaustedException if a recrord is sent and the buffer space is full. |
| metadata.fetch.timeout.ms | long | 60000 | low | 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 configuration controls the maximum amount of time we will block waiting for the metadata fetch to succeed before throwing an exception back to the client. |
| metadata.max.age.ms | long | 300000 | low | The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions. |
| metric.reporters | list | [] | low | A list of classes to use as metrics reporters. Implementing the `MetricReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics. |
| metrics.num.samples | int | 2 | low | The number of samples maintained to compute metrics. |
| metrics.sample.window.ms | long | 30000 | low | The metrics system maintains a configurable number of samples over a fixed window size. This configuration controls the size of the window. For example we might maintain two samples each measured over a 30 second period. When a window expires we erase and overwrite the oldest window. |
| reconnect.backoff.ms | long | 10 | low | The amount of time to wait before attempting to reconnect to a given host when a connection fails. This avoids a scenario where the client repeatedly attempts to connect to a host in a tight loop. |
| retry.backoff.ms | long | 100 | low | The amount of time to wait before attempting to retry a failed produce request to a given topic partition. This avoids repeated sending-and-failing in a tight loop. |

Download

@apachekafka

## 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 new the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

Supporting these uses led use to a design with a number of unique elements, more akin to a database log then 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.

You're viewing documentation for an older version of Kafka - check out our current documentation here.

ge that all operations are O(1) server can now take full 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 deleting messages as soon as they are consumed, we can retain messages for a relative 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.

Download

@apachekafka

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 and Snappy 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 appropriate 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 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 ing systems in this fashion led

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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 structure 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 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, or 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

ows automatic failover to

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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 definition of, how far behind is too far, is controlled by the replica.lag.max.messages configuration and the definition of a stuck replica 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 request.required.acks setting that the producer uses.

Download

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.

@apachekafka

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 on 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. In our current release we choose the second strategy and favor choosing a potentially inconsistent replica when all replicas in the ISR are dead. In the future, we would like to make this configurable to better 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.

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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 request.required.acks=-1, 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 request.required.acks=-1 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 required.acks=-1 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.

Download

@apachekafka

## 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.

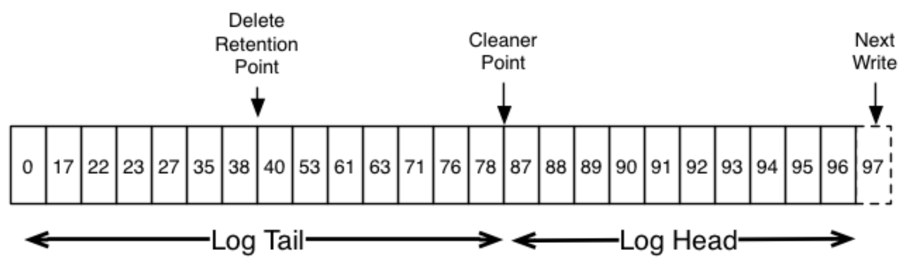
Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each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



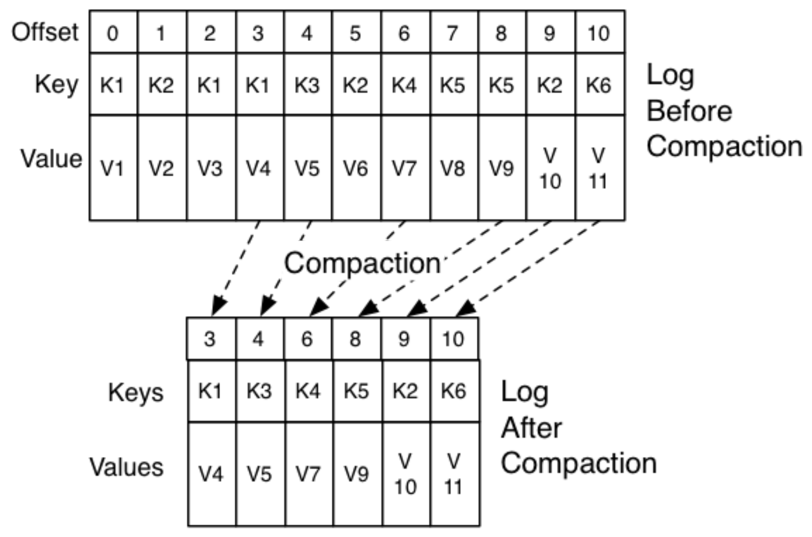
the tail of the log. The picture

You're viewing documentation for an older version of Kafka - check out our current documentation here.

changes. Note also that all [illegible] 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:



Download

@apachekafka

### 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 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.
2. Log compaction is not yet compatible with compressed topics.

## 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 */
```

You're viewing documentation for an older version of Kafka - check out our current documentation here.

```
  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.

Download

@apachekafka

## 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$.MODULE$.LATIEST_TIME(), get from the latest offset available.
   *              if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the earliest offset available.
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);
}
```

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

### High-level API

```

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

interface ConsumerConnector {

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

  /**
   * You can also obtain a list of KafkaStreams, that iterate over messages
```

You're viewing documentation for an older version of Kafka - check out our current documentation here.

```
  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.

Download

@apachekafka

## 5.3 Messages

Messages consist of a fixed-size header and variable length opaque byte array payload. The header contains a format version and a CRC32 checksum to detect corruption or truncation. Leaving the payload 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

```
        /**
         * A message. The format of an N byte message is the following:
         *
         * If magic byte is 0
         *
         * 1. 1 byte "magic" identifier to allow format changes
         *
         * 2. 4 byte CRC32 of the payload
         *
         * 3. N - 5 byte payload
         *
         * If magic byte is 1
         *
         * 1. 1 byte "magic" identifier to allow format changes
         *
         * 2. 1 byte "attributes" identifier to allow annotations on the message independent of the version (e.g. compression enabled,
         *
         * 3. 4 byte CRC32 of the payload
         *
         * 4. N - 6 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.

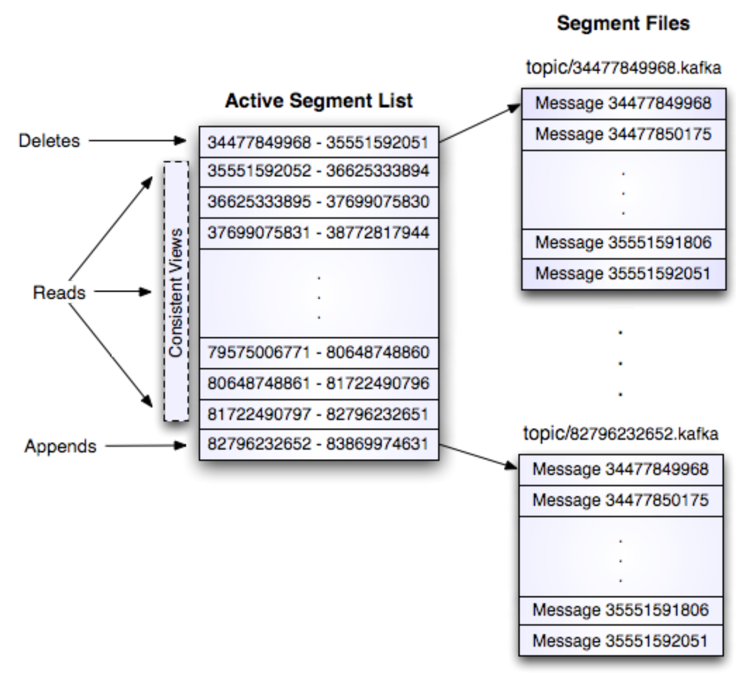
The exact binary format for messages is versioned and maintained as a standard interface so message sets can be transfered between producer, broker, and client without recopying or conversion when desirable. This format is as follows:

```
On-disk format of a message

message length : 4 bytes (value: 1+4+n)
"magic" value  : 1 byte
crc            : 4 bytes
payload        : n 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.

You're viewing documentation for an older version of Kafka - check out our current documentation here.



Download

@apachekafka

### 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 parameter *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 need 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-existant 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 not 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 ConsumerMetadataRequest to any

om the offsets manager
look at these code samples

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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.

Download

@apachekafka

### 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] --> host:port (ephemeral node)
```

This is a list of all present broker nodes, each of which provides a unique logical broker id which identifies it to consumers (which must be given as part of its configuration). On startup, a broker node registers itself by creating a znode with the logical broker id under /brokers/ids. The purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) is 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]/[0...N] --> nPartions (ephemeral node)
```

Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

### Consumers and Consumer Groups

Consumers of topics also register themselves in ZooKeeper, in order to coordinate with each other and balance the consumption of data. Consumers can also store their offsets in ZooKeeper by setting `offsets.storage=zookeeper` . However, this offset storage mechanism will be deprecated in a future release. Therefore, it is recommended to migrate offsets storage to Kafka.

Multiple consumers can form a group and jointly consume a single topic. Each consumer in the same group is given a shared group_id. For example if one consumer is your foobar process, which is run across three machines, then you might assign this group of consumers the id "foobar". This group id is provided in the configuration of the consumer, and is your way to tell the consumer which group it belongs to.

The consumers in a group divide up the partitions as fairly as possible, each partition is consumed by exactly one consumer in a consumer group.

### Consumer Id Registry

In addition to the group_id which is shared by all consumers in a group, each consumer is given a transient, unique consumer_id (of the form hostname:uuid) for identification purposes. Consumer ids are registered in the following directory.

```
/consumers/[group_id]/ids/[consumer_id] --> {"topic1": #streams, ..., "topicN": #streams} (ephemeral node)
```

Each of the consumers in the group registers under its group and creates a znode with its consumer_id. The value of the znode contains a map of <topic, #streams>. This id is simply used to identify each of the consumers which is currently active within a group. This is an ephemeral node so it will disappear if the consumer process dies.

### Consumer Offsets

Consumers track the maximum offset they have consumed in each partition. This value is stored in a ZooKeeper directory if `offsets.storage=zookeeper` . This valued is stored in a ZooKeeper directory.

```
/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id] --> offset_counter_value ((persistent node)
```

### Partition Owner registry

Each broker partition is consumed by a single consumer within a given consumer group. The consumer must establish its ownership of a given partition before any consumption can begin. To establish its ownership, a consumer writes its own id in an ephemeral node under the particular broker partition it is claiming.

You're viewing documentation for an older version of Kafka - check out our current documentation here.

### 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

Download

@apachekafka

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 $C_i$ subscribes to
2.   let $P_T$ be all partitions producing topic T
3.   let $C_G$ be all consumers in the same group as $C_i$ that consume topic T
4.   sort $P_T$ (so partitions on the same broker are clustered together)
5.   sort $C_G$
6.   let i be the index position of $C_i$ in $C_G$ and let N = size($P_T$)/size($C_G$)
7.   assign partitions from i*N to (i+1)*N - 1 to consumer $C_i$
8.   remove current entries owned by $C_i$ from the partition owner registry
9.   add newly assigned partitions to the partition owner registry
     (we may need to re-try this until the original partition owner releases its ownership)
```

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

## 6. OPERATIONS

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

### 6.1 Basic Kafka Operations

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

### Adding and removing topics

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

Topics are added and modified using the topic tool:

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --create --topic my_topic_name
      --partitions 20 --replication-factor 3 --config x=y
```

The replication factor controls how many servers will replicate each message that is written. If you have a replication factor of 3 then up to 2 servers can fail before you will lose access to your data. We recommend you use a replication factor of 2 or 3 so that you can transparently bounce machines without interrupting data consumption.

The partition count controls how many logs the topic will be sharded into. There are several impacts of the partition count. First each partition must fit entirely on a single server. So if you have 20 partitions the full data set (and read and write load) will be handled by no more than 20 servers (no counting replicas). Finally the partition count impacts the maximum parallelism of your consumers. This is discussed in greater detail in the concepts section.

The configurations added on the command line override the default settings the server has for things like the length of time data should be retained. The complete set of per-topic configurations is documented here.

### Modifying topics

You can change the configuration or partitioning of a topic using the same topic tool.

To add partitions you can do

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name
      --partitions 40
```

Be aware that one use case for partitions is to semantically partition data, and adding partitions doesn't change the partitioning of existing data so this may disturb consumers if they rely on that partition. That is if data is partitioned by `hash(key) % number_of_partitions` then this partitioning will potentially be shuffled by adding partitions but Kafka will not attempt to automatically redistribute data in any way.

To add configs:

You're viewing documentation for an older version of Kafka - check out our current documentation here.

To remove a config:

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

And finally deleting a topic:

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

Topic deletion option is disabled by default. To enable it set the server config

```
delete.topic.enable=true
```

Kafka does not currently support reducing the number of partitions for a topic or changing the replication factor.

Download

@apachekafka

## 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 later cases Kafka supports a more graceful mechanism for stoping a server then 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 happen 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:

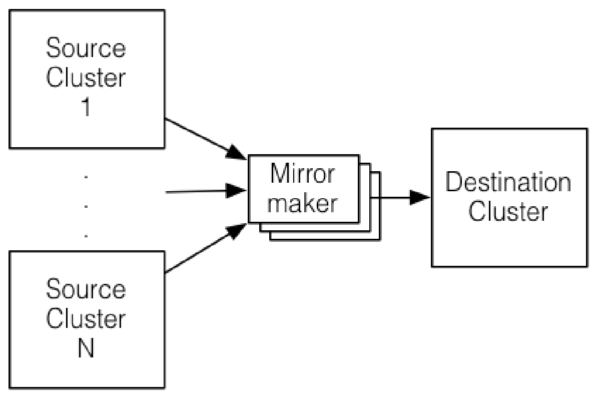
```
  > bin/kafka-preferred-replica-election.sh --zookeeper zk_host:port/chroot
```

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
```

## 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 one or more source clusters 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:

You're viewing documentation for an older version of Kafka - check out our current documentation here.

```
        --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.

Sometime 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.

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 --zkconnect localhost:2181 --group test
Group           Topic                          Pid Offset          logSize         Lag             Owner
my-group        my-topic                       0   0               0               0               test_jkreps-mn-1394154511599-60744496
my-group        my-topic                       1   0               0               0               test_jkreps-mn-1394154521217-1a0be91
```

Download

@apachekafka

## 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. In 0.8.1, the partition reassignment tool does not have the capability to automatically study the data distribution in a Kafka cluster and move partitions around to attain an even load distribution. As such, the admin has to figure out which topics or partitions should be moved around.

The partition reassignment tool can run in 3 mutually exclusive modes -

- --generate: In this mode, given a list of topics and a list of brokers, the tool generates a candidate reassignment to move all partitions of the specified topics to the new brokers. This option merely provides a convenient way to generate a partition reassignment plan given a list of topics and target brokers.
- --execute: In this mode, the tool kicks off the reassignment of partitions based on the user provided reassignment plan. (using the --reassignment-json-file option). This can either be a custom reassignment plan hand crafted by the admin or provided by using the --generate option
- --verify: In this mode, the tool verifies the status of the reassignment for all partitions listed during the last --execute. The status can be either of successfully completed, failed or in progress

### Automatically migrating data to new machines

The partition reassignment tool can be used to move some topics off of the current set of brokers to the newly added brokers. This is typically useful while expanding an existing cluster since it is easier to move entire topics to the new set of brokers, than moving one partition at a time. When used to do this, the user should provide a list of topics that should be moved to the new set of brokers and a target list of new brokers. The tool then evenly distributes all partitions for the given list of topics across the new set of brokers. During this move, the replication factor of the topic is kept constant. Effectively the replicas for all partitions for the input list of topics are moved from the old set of brokers to the newly added brokers.

For instance, the following example will move all partitions for topics foo1,foo2 to the new set of brokers 5,6. At the end of this move, all partitions for topics foo1 and foo2 will *only* exist on brokers 5,6

Since, the tool accepts the input list of topics as a json file, you first need to identify the topics you want to move and create the json file as follows-

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"},
            {"topic": "foo2"}],
 "version":1
}
```

Once the json file is ready, use the partition reassignment tool to generate a candidate assignment-

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "5,6" --gene
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]}]
}
```

You're viewing documentation for an older version of Kafka - check out our current documentation here.

ment has not started, it merely tells you the current assignment and the proposed new assignment. The current assignment should be saved in case you want to rollback to it. The new assignment should be saved in a json file (e.g. expand-cluster-reassignment.json) to be input to the tool with the --execute option as follows-

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
              {"topic":"foo1","partition":0,"replicas":[3,4]},
              {"topic":"foo2","partition":2,"replicas":[1,2]},
              {"topic":"foo2","partition":0,"replicas":[3,4]},
              {"topic":"foo1","partition":1,"replicas":[2,3]},
              {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
              {"topic":"foo1","partition":0,"replicas":[5,6]},
              {"topic":"foo2","partition":2,"replicas":[5,6]},
              {"topic":"foo2","partition":0,"replicas":[5,6]},
              {"topic":"foo1","partition":1,"replicas":[5,6]},
              {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

Finally, the --verify option can be used with the tool to check the status of the partition reassignment. Note that the same expand-cluster-reassignment.json (used with the --execute option) should be used with the --verify option

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo1,1] is in progress
Reassignment of partition [foo1,2] is in progress
Reassignment of partition [foo2,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
Reassignment of partition [foo2,2] completed successfully
```

#### Custom partition assignment and migration

The partition reassignment tool can also be used to selectively move replicas of a partition to a specific set of brokers. When used in this manner, it is assumed that the user knows the reassignment plan and does not require the tool to generate a candidate reassignment, effectively skipping the --generate step and moving straight to the --execute step

For instance, the following example moves partition 0 of topic foo1 to brokers 5,6 and partition 1 of topic foo2 to brokers 2,3

The first step is to hand craft the custom reassignment plan in a json file-

```
> cat custom-reassignment.json
{"version":1,"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},{"topic":"foo2","partition":1,"replicas":[2,3]}]}
```

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

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[1,2]},
              {"topic":"foo2","partition":1,"replicas":[3,4]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
              {"topic":"foo2","partition":1,"replicas":[2,3]}]
}
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same expand-cluster-reassignment.json (used with the --execute option) should be used with the --verify option

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
```

### Decommissioning brokers

The partition reassignment tool does not have the ability to automatically generate a reassignment plan for decommissioning brokers yet. As such, the admin has to come up with a reassignment plan to move the replica for all partitions hosted on the broker to be decommissioned, to the rest of the brokers. This can be relatively tedious as the reassignment needs to ensure that all the replicas are not moved from the decommissioned broker to only one other broker. To make this process effortless, we plan to add tooling support for decommissioning brokers in 0.8.2.

You're viewing documentation for an older version of Kafka - check out our current documentation here.

Increasing the replication factor of an existing partition is easy. Just specify the extra replicas in the custom reassignment json file and use it with the --execute option to increase the replication factor of the specified partitions.

For instance, the following example increases the replication factor of partition 0 of topic foo from 1 to 3. Before increasing the replication factor, the partition's only replica existed on broker 5. As part of increasing the replication factor, we will add more replicas on brokers 6 and 7.

The first step is to hand craft the custom reassignment plan in a json file-

```
> cat increase-replication-factor.json
{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

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

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same increase-replication-factor.json (used with the --execute option) should be used with the --verify option

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-replication-factor.json --verify
Status of partition reassignment:
Reassignment of partition [foo,0] completed successfully
```

You can also verify the increase in replication factor with the kafka-topics tool-

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --topic foo --describe
Topic:foo       PartitionCount:1        ReplicationFactor:3     Configs:
        Topic: foo      Partition: 0    Leader: 5       Replicas: 5,6,7 Isr: 5,6,7
```

## 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 server 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
replica.lag.max.messages=4000
```

You're viewing documentation for an older version of Kafka - check out our current documentation here.

```
# 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

We're currently running JDK 1.7 u51, and we've switched over to the G1 collector. If you do this (and we highly recommend it), make sure you're on u51. We tried out u21 in testing, but we had a number of problems with the GC implementation in that version. Our tuning looks like this:

```
-Xms4g -Xmx4g -XX:PermSize=48m -XX:MaxPermSize=48m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
```

For reference, here are the stats on one of LinkedIn's busiest clusters (at peak): - 15 brokers - 15.5k partitions (replication factor 2) - 400k messages/sec in - 70 MB/sec inbound, 400 MB/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 more 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.

You likely don't need to do much OS-level tuning though there are a few things that will help performance.

Two configurations that may be important:

- We upped the number of file descriptors since we have lots of topics and lots of connections.
- We upped the max socket buffer size to enable high-performance data transfer between data centers 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. As of 0.8 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 and 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.

can introduce latency as

You're viewing documentation for an older version of Kafka - check out our current documentation here.

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

Download

@apachekafka

### Ext4 Notes

Ext4 may or may not be the best filesystem for Kafka. Filesystems like XFS supposedly handle locking during fsync better. We have only tried Ext4, though.

It is not necessary to tune these settings, however those wanting to optimize performance have a few knobs that will help:

- 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 to fire up jconsole and point it at a running kafka client or server; this will all browsing all metrics with JMX.

We pay particular we do graphing and alerting on the following metrics:

| DESCRIPTION | MBEAN NAME | NORMAL VALUE |
|---|---|---|
| Message in rate | kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec | |
| Byte in rate | kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec | |
| Request rate | kafka.network:type=RequestMetrics,name=RequestsPerSec,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 one broker in the cluster should have 1 |
| Leader election rate | kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs | non-zero when there are broker failures |
| Unclean leader election rate | kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec | 0 |
| Partition counts | kafka.server:type=ReplicaManager,name=PartitionCount | mostly even across brokers |
| Leader replica counts | kafka.server:type=ReplicaManager,name=LeaderCount | mostly even across brokers |
| ISR shrink rate | kafka.server:type=ReplicaManager,name=IsrShrinksPerSec | If a broker goes down, ISR for some of the partitions will shrink. When that broker is up again, ISR will be expanded once the replicas are fully caught up. Other than that, the expected value for both ISR shrink rate and expansion rate is 0. |
| ISR expansion rate | kafka.server:type=ReplicaManager,name=IsrExpandsPerSec | See above |
| Max lag in messages btw follower and leader replicas | kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica | < replica.lag.max.messages |
| Lag in messages per follower replica | kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=([-.\w]+),topic=([-.\w]+),partition=([0-9]+) | < replica.lag.max.messages |

You're viewing documentation for an older version of Kafka - check out our current documentation here.

| | | |
|---|---|---|
| Requests waiting in the fetch purgatory | kafka.server:type=FetchRequestPurgatory,name=PurgatorySize | size depends on fetch.wait.max.ms in the consumer |
| Request total time | kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce\|FetchConsumer\|FetchFollower} | broken into queue, local, remote and response send 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-zero for produce requests when ack=-1 |
| 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 | between 0 and 1, ideally > 0.3 |
| The average fraction of time the request handler threads are idle | kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent | between 0 and 1, ideally > 0.3 |

Download

@apachekafka

## New producer monitoring

The following metrics are available on new producer instances.

| METRIC/ATTRIBUTE NAME | DESCRIPTION | MBEAN NAME |
|---|---|---|
| waiting-threads | The number of user threads blocked waiting for buffer memory to enqueue their records | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| buffer-total-bytes | The maximum amount of buffer memory the client can use (whether or not it is currently used). | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| buffer-available-bytes | The total amount of buffer memory that is not being used (either unallocated or in the free list). | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| bufferpool-wait-time | The fraction of time an appender waits for space allocation. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| batch-size-avg | The average number of bytes sent per partition per-request. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| batch-size-max | The max number of bytes sent per partition per-request. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| compression-rate-avg | The average compression rate of record batches. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-queue-time-avg | The average time in ms record batches spent in the record accumulator. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-queue-time-max | The maximum time in ms record batches spent in the record accumulator | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| request-latency-avg | The average request latency in ms | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| request-latency-max | The maximum request latency in ms | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-send-rate | The average number of records sent per second. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| records-per-request-avg | The average number of records per request. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-retry-rate | The average per-second number of retried record sends | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-error-rate | The average per-second number of record sends that resulted in errors | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-size-max | The maximum record size | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| record-size-avg | The average record size | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| requests-in-flight | The current number of in-flight requests awaiting a response. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| metadata-age | The age in seconds of the current producer metadata being used. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| connection-close-rate | Connections closed per second in the window. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| connection-creation-rate | New connections established per second in the window. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| network-io-rate | The average number of network operations (reads or writes) on all connections per second. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| outgoing-byte-rate | The average number of outgoing bytes sent per second to all servers. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| request-rate | The average number of requests sent per second. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| request-size-avg | The average size of all requests in the window. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| request-size-max | The maximum size of any request sent in the window. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| incoming-byte-rate | Bytes/second read off all sockets | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| response-rate | Responses received sent per second. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| select-rate | Number of times the I/O layer checked for new I/O to perform per second | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |

You're viewing documentation for an older version of Kafka - check out our current documentation here.

Download

@apachekafka

| | | |
|---|---|---|
| | | s,client-id=([-.\w]+) |
| io-wait-ratio | The fraction of time the I/O thread spent waiting. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| io-time-ns-avg | The average length of time for I/O per select call in nanoseconds. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| io-ratio | The fraction of time the I/O thread spent doing I/O | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| connection-count | The current number of active connections. | kafka.producer:type=producer-metrics,client-id=([-.\w]+) |
| outgoing-byte-rate | The average number of outgoing bytes sent per second for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| request-rate | The average number of requests sent per second for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| request-size-avg | The average size of all requests in the window for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| request-size-max | The maximum size of any request sent in the window for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| incoming-byte-rate | The average number of responses received per second for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| request-latency-avg | The average request latency in ms for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| request-latency-max | The maximum request latency in ms for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| response-rate | Responses received sent per second for a node. | kafka.producer:type=producer-node-metrics,client-id=([-.\w]+),node-id=([0-9]+) |
| record-send-rate | The average number of records sent per second for a topic. | kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+) |
| byte-rate | The average number of bytes sent per second for a topic. | kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+) |
| compression-rate | The average compression rate of record batches for a topic. | kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+) |
| record-retry-rate | The average per-second number of retried record sends for a topic | kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+) |
| record-error-rate | The average per-second number of record sends that resulted in errors for a topic. | kafka.producer:type=producer-topic-metrics,client-id=([-.\w]+),topic=([-.\w]+) |

We recommend monitor GC time and other stats and various server stats such as CPU utilization, I/O service time, etc. On the client side, we recommend monitor 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

At LinkedIn, we are running ZooKeeper 3.3.*. Version 3.3.3 has known serious issues regarding ephemeral node deletion and session expirations. After running into those issues in production, we upgraded to 3.3.4 and have been running that smoothly for over a year now.

### 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.
- I/O segregation: if you do a lot of write type traffic you'll almost definitely want the transaction logs on a different disk group than application logs and snapshots (the write to the ZooKeeper service has a synchronous write to disk, which can be slow).
- 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 and monitoring: 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. As far as 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).
- Try to run on a 3-5 node cluster: ZooKeeper writes use quorums and inherently that means having an odd number of machines in a cluster. Remember that a 5 node cluster will cause writes to slow down compared to a 3 node cluster, but will allow more fault tolerance.

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.

The contents of this website are © 2016 Apache Software Foundation under the terms of the Apache License v2. Apache Kafka, Kafka, and the Kafka logo are either registered trademarks or trademarks of The Apache Software Foundation in the United States and other countries.