CHAPTER 2

Installing Kafka

This chapter describes how to get started running the Apache Kafka broker, including how to set up Apache Zookeeper, which is used by Kafka for storing metadata for the brokers. The chapter will also cover the basic configuration options that should be reviewed for a Kafka deployment, as well as criteria for selecting the correct hardware to run the brokers on. Finally, we cover how to install multiple Kafka brokers together as part of a single cluster, and some specific concerns when shifting to using Kafka in a production environment.

First Things First

Choosing an Operating System

Apache Kafka is a Java application, and is run under many operating systems. This includes Windows, OS X, Linux, and others. The installation steps in this chapter will be focused on setting up and using Kafka in a Linux environment, as this is the most common OS on which it is installed. This is also the recommended OS for deploying Kafka for general use.

Installing Java

Prior to installing either Zookeeper or Kafka, you will need a Java environment set up and functioning. This should be a Java 8 version, and can be the version provided by your operating system or one directly downloaded from java.com. While Zookeeper and Kafka will work with a runtime edition of Java, it may be more convenient when developing tools and applications to have the full Java Development Kit. As such, the rest of the installation steps will assume you have installed JDK version 8, update 51 in `/usr/java/jdk1.8.0_51`.

Installing Zookeeper

Apache Kafka uses Zookeeper to store metadata information about the Kafka cluster, as well as consumer client details. While it is possible to run a Zookeeper server using scripts contained within the Kafka distribution, it is trivial to install a full version of Zookeeper from the distribution.

Kafka has been tested extensively with the stable 3.4.6 release of Zookeeper. Download that version of Zookeeper from apache.org at *http://mirror.cc.columbia.edu/pub/software/apache/zookeeper/zookeeper-3.4.6/zookeeper-3.4.6.tar.gz*.

Standalone Server

The following example installs Zookeeper with a basic configuration in `/usr/local/zookeeper`, storing its data in `/var/lib/zookeeper`

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

You can now validate that Zookeeper is running correctly in standalone mode by connecting to the client port and sending the four letter command 'srvr':

```
# telnet localhost 2181
Trying ::1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
#
```

Zookeeper Ensemble

A Zookeeper cluster is called an "ensemble". Due to the consensus protocol used, it is recommended that ensembles contain an odd number of servers (e.g. 3, 5, etc.) as a majority of ensemble members (a quorum) must be working for Zookeeper to respond to requests. This means in a 3-node ensemble, you can run with one node missing. With a 5-node ensemble, you can run with two nodes missing.

> Consider running Zookeeper in a 5-node ensemble. In order to make configuration changes to the ensemble, including swapping a node, you will need to reload nodes one at a time. If your ensemble cannot tolerate more than one node being down, doing maintenance work introduces additional risk.

To configure Zookeeper servers in an ensemble, they must have a common configuration that lists all servers, and each server needs a `myid` file in the data directory which specifies the ID number of the server. If the hostnames of the servers in the ensemble are `zoo1.example.com`, `zoo2.example.com`, and `zoo3.example.com`, the configuration file may be:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

In this configuration, the `initLimit` is the amount of time to allow for followers to connect with a leader. The `syncLimit` value limits how far out of sync followers can be with the leader. Both values are a number of `tickTime` units, which makes the `initLimit` 20 * 2000 ms, or 40 seconds. The configuration also lists each server in the ensemble. The servers are specified in the format `server.X=hostname:peerPort:leaderPort`, with the following parameters:

- `X` is the ID number of the server. This must be an integer, but it does not need to be zero-based or sequential
- `hostname` is the hostname or IP address of the server
- `peerPort` is is the TCP port over which servers in the ensemble communicate with each other
- `leaderPort` is the TCP port over which leader election is performed

Clients only need to be able to connect to the ensemble over the clientPort, but the members of the ensemble must be able to communicate with each other over all three ports.

In addition to the shared configuration file, each server must have a file in the `dataDir` directory with the name `myid`. This file must contain the ID number of the server, which must match the configuration file. Once these steps are complete, the servers will start up and communicate with each other in an ensemble.

Installing a Kafka Broker

Once Java and Zookeeper are configured, you are ready to install Apache Kafka. The current release of Kafka can be downloaded at *http://kafka.apache.org/downloads.html*. At press time, that version is 0.8.2.1 running under Scala version 2.10.0.

The following example installs Kafka in `/usr/local/kafka`, configured to use the Zookeeper server started previously and to store the message log segments stored in `/tmp/kafka-logs`:

```
# tar -zxf kafka_2.10-0.8.2.1.tgz
# mv kafka_2.10-0.8.2.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Once the Kafka broker is started, we can verify it is working by performing some simple operations against the cluster creating a test topic, producing some messages, and consuming the same messages:

Create and verify a topic:

```
# /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --zookeeper localhost:2181
--describe --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
#
```

Produce messages to a test topic:

```
# /usr/local/kafka/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test
Test Message 1
Test Message 2
^D
#
```

Consume messages from a test topic:

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic test --from-beginning
Test Message 1
Test Message 2
^C
Consumed 2 messages
#
```

Broker Configuration

The example configuration that is provided with the Kafka distribution is sufficient to run a standalone server as a proof of concept, but it will not be sufficient for most installations. There are numerous configuration options for Kafka which control all aspects of setup and tuning. Many options can be left to the default settings, as they deal with tuning aspects of the Kafka broker that will not be applicable until you have a specific use case to work with and a requirement to adjust them.

General Broker

There are several broker configurations that should be reviewed when deploying Kafka for any environment other than a standalone broker on a single server. These parameters deal with the basic configuration of the broker, and most of them must be changed to run properly in a cluster with other brokers.

broker.id

Every Kafka broker must have an integer identifier, which is set using the `broker.id` configuration. By default, this integer is set to 0, but it can be any value. The most important thing is that it must be unique within a single Kafka cluster. The selection of this number is arbitrary, and it can be moved between brokers if necessary for maintenance tasks. A good guideline is to set this value to something intrinsic to the host so that when performing maintenance it is not onerous to map broker ID numbers to hosts. For example, if your hostnames contain a unique number (such as `host1.example.com`, `host2.example.com`, etc.), that is a good choice for the broker.id value.

port

The example configuration file starts Kafka with a listener on TCP port 9092. This can be set to any available port by changing the `port` confiugration parameter. Keep in mind that if a port lower than 1024 is chosen, Kafka must be started as root. Running Kafka as root is not a recommended configuration.

zookeeper.connect

The location of the Zookeeper used for storing the broker metadata is set using the `zookeeper.connect` configuration parameter. The example configuration uses a Zookeeper running on port 2181 on the local host, which is specified as `localhost:2181`. The format for this parameter is a semicolon separated list of `hostname:port/path` strings, where the parts are:

- `hostname` is the hostname or IP address of the Zookeeper server
- `port` is the client port number for the server
- `/path` is an optional Zookeeper path to use as a chroot environment for the Kafka cluster. If it is omitted, the root path is used.

If a chroot path is specified and does not exist, it will be created by the broker when it starts up.

> It is generally considered to be a good practice to use a chroot path for the Kafka cluster. This allows the Zookeeper ensemble to be shared with other applications, including other Kafka clusters, without a conflict. It is also best to specify multiple Zookeeper servers (which are all part of the same ensemble) in this configuration separated by semicolons. This allows the Kafka broker to connect to another member of the Zookeeper ensemble in the case of a server failure.

log.dirs

Kafka persists all messages to disk, and these log segments are stored in the directories specified in the log.dirs configuration. This is a comma separated list of paths on the local system. If more than one path is specified, the broker will store partitions on them in a "least used" fashion with one partition's log segments stored within the same path. Note that the broker will place a new partition in the path that has the least number of partitions currently stored in it, not the least amount of disk space used.

num.recovery.threads.per.data.dir

Kafka uses a configurable pool of threads for handling log segments in three situations:

- When starting normally, to open each partition's log segments
- When starting after a failure, to check and truncate each partition's log segments
- When shutting down, to cleanly close log segments

By default, only one thread per log directory is used. As these threads are only used during startup and shutdown, it is reasonable to set a larger number of threads in order to parallelize operations. Specifically, when recovering from an unclean shutdown this can mean the difference of several hours when restarting a broker with a large number of partitions! When setting this parameter, remember that the number configured is per log directory specified with log.dirs. This means that if `num.recovery.threads.per.data.dir` is set to 8, and there are 3 paths specified in `log.dirs`, this is a total of 24 threads.

auto.create.topics.enable

The default Kafka configuration specifies that the broker should automatically create a topic under the following circumstances

- When a producer starts writing messages to the topic
- When a consumer starts reading messages from the topic
- When any client requests metadata for the topic

In many situations, this can be undesirable behavior, especially as there is no way to validate the existence of a topic through the Kafka protocol without causing it to be created. If you are managing topic creation explicitly, whether manually or through a provisioning system, you can set the `auto.create.topics.enable` configuration to `false`.

Topic Defaults

The Kafka server configuration specifies many default configurations for topics that are created. Several of these parameters, including partition counts and message retention, can be set per-topic using the administrative tools (covered in Chapter 8). The defaults in the server configuration should be set to baseline values that are appropriate for the majority of the topics in the cluster.

> In previous versions of Kafka, it was possible to specify per-topic overrides for these configurations in the broker configuration using parameters named `log.retention.hours.per.topic`, `log.retention.bytes.per.topic`, and `log.segment.bytes.per.topic`. These parameters are no longer supported, and overrides must be specified using the administrative tools.

num.partitions

The `num.partitions` parameter determines how many partitions a new topic is created with, primarily when automatic topic creation is enabled (which is the default

setting). This parameter defaults to 1 partition. Keep in mind that the number of partitions for a topic can only be increased, never decreased. This means that if a topic needs to have fewer partitions than `num.partitions`, care will need to be taken to manually create the topic (discussed in Chapter 8).

As described in Chapter 1, partitions are the way a topic is scaled within a Kafka cluster, which makes it important to use partition counts that will balance the message load across the entire cluster as brokers are added. This does not mean that all topics must have a partition count higher than the number of brokers so that they span all brokers, provided there are multiple topics (which will also be spread out over the brokers). However, in order to spread out the load for a topic with a high message volume, the topic will need to have a larger number of partitions.

log.retention.ms

The most common configuration for how long Kafka will retain messages is by time. The default is specified in the configuration file using the `log.retention.hours` parameter, and it is set to 168 hours, or one week. However, there are two other parameters allowed, `log.retention.minutes` and `log.retention.ms`. All three of these specify the same configuration, the amount of time after which messages may be deleted, but the recommended parameter to use is `log.retention.ms`. If more than one is specified, the smaller unit size will take precedence.

> Retention by time is performed by examining the last modified time (mtime) on each log segment file on disk. Under normal cluster operations, this is the time that the log segment was closed, and represents the timestamp of the last message in the file. However, when using administrative tools to move partitions between brokers, this time is not accurate. This will result in excess retention for these partitions. More information on this is provided in Chapter 8 when discussing partition moves.

log.retention.bytes

Another way to expire messages is based on the total number of bytes of messages retained. This value is set using the `log.retention.bytes` parameter, and it is applied per-partition. This means that if you have a topic with 8 partitions, and `log.retention.bytes` is set to 1 gigabyte, the amount of data retained for the topic will be 8 gigabytes at most. Note that all retention is performed for an individual par-

tition, not the topic. This means that should the number of partitions for a topic be expanded, the retention will increase as well if `log.retention.bytes` is used.

> If you have specified a value for both `log.retention.bytes` and `log.retention.ms` (or another parameter for retention by time), messages may be removed when either criteria is met. For example, if `log.retention.ms` is set to 86400000 (1 day), and `log.retention.bytes` is set to 1000000000 (1 gigabyte), it is possible for messages that are less than 1 day old to get deleted if the total volume of messages over the course of the day is greater than 1 gigabyte. Conversely, if the volume is less than 1 gigabyte, messages can be deleted after 1 day even if the total size of the partition is less than 1 gigabyte.

log.segment.bytes

The log retention settings above operate on log segments, not individual messages. As messages are produced to the Kafka broker, they are appended to the current log segment for the partition. Once the log segment has reached the size specified by the `log.segment.bytes` parameter, which defaults to 1 gibibyte, the log segment is closed and a new one is opened. Once a log segment has been closed, it can be considered for expiration. A smaller size means that files must be closed and allocated more often, which reduces the overall efficiency of disk writes.

Adjusting the size of the log segments can be important if topics have a low produce rate. For example, if a topic receives only 100 megabytes per day of messages, and `log.segment.bytes` is set to the default, it will take 10 days to fill one segment. As messages cannot be expired until the log segment is closed, if `log.retention.ms` is set to 604800000 (1 week), there will actually be up to 17 days of messages retained until the closed log segment is expired. This is because once the log segment is closed with the current 10 days of messages, that log segment must be retained for 7 days before it can be expired based on the time policy (as the segment can not be removed until the last message in the segment can be expired).

> The size of the log segments also affects the behavior of fetching offsets by timestamp. When requesting offsets for a partition at a specific timestamp, Kafka fulfills the request by looking for the log segment in the partition where the last modified time of the file is (and therefore closed) after the timestamp and the immediately previous segment was last modified before the timestamp. Kafka then returns the offset at the beginning of that log segment (which is also the filename). This means that smaller log segments will provide more accurate answers for offset requests by timestamp.

log.segment.ms

Another way to control when log segments are closed is by using the `log.segment.ms` parameter, which specifies the amount of time after which a log segment should be closed. As with the `log.retention.bytes` and `log.retention.ms` parameters, `log.segment.bytes` and `log.segment.ms` are not mutually exclusive properties. Kafka will close a log segment either when the size limit is reached, or when the time limit is reached, whichever comes first. By default, there is no setting for `log.segment.ms`, which results in only closing log segments by size.

> When using a time-based log segment limit, it is important to consider the impact on disk performance when multiple log segments are closed simultaneously. This can happen when there are many partitions which never reach the size limit for log segments, as the clock for the time limit will start when the broker starts and will always execute at the same time for these low-volume partitions.

message.max.bytes

The Kafka broker limits the maximum size of a message that can be produced, configured by the `message.max.bytes` parameter which defaults to 1000000, or 1 megabyte. A producer which tries to send a message larger than this will receive an error back from the broker and the message will not be accepted. As with all byte sizes specified on the broker, this configuration deals with compressed message size, which means that producers can send messages that are much larger than this value uncompressed, provided they compress down to under the configured `message.max.bytes` size.

There are noticeable performance impacts from increasing the allowable message size. Larger messages will mean that the broker threads that deal with processing network connections and requests will be working longer on each request. It also increases the size of disk writes, which will impact I/O throughput.

> The message size configured on the Kafka broker must be coordinated with the `fetch.message.max.bytes` configuration on consumer clients. If this value is smaller than `message.max.bytes`, then consumers which encounter larger messages will fail to fetch those messages, resulting in a situation where the consumer gets stuck and cannot proceed. The same rule applies to the `replica.fetch.max.bytes` configuration on the brokers when configured in a cluster.

Hardware Selection

Selecting an appropriate hardware configuration for a Kafka broker can be more art than science. Kafka itself has no strict requirement on a specific hardware configuration, and will run without issue on any system. Once performance becomes a concern, however, there are several factors that must be considered that will contribute to the overall performance. Once you have determined which types of performance are the most critical for your environment, you will be able to select an optimized hardware configuration that fits within your budget.

Disk Throughput

The performance of producer clients will be most directly influenced by the throughput of the broker disk that is used for storing log segments. Kafka messages must be committed to local storage when they are produced, and most clients will wait until at least one broker has confirmed that messages have been committed before considering the send successful. This means that faster disk writes will equal lower produce latency.

The obvious decision when it comes to disk throughput is whether to use traditional spinning hard drives (HDD) or solid state disks (SSD). Solid state disks have drastically lower seek and access times and will provide the best performance. Spinning disks, on the other hand, are more economical and provide more capacity per unit. You can also improve the performance of spinning disks by using more of them in a broker, whether by having multiple data directories or by setting up the drives in a RAID configuration. Other factors, such as the specific drive technology (e.g. Serial Attached Storage or Serial ATA), as well as the quality of the drive controller, will affect throughput.

Disk Capacity

Capacity is the other side of the storage discussion. The amount of disk capacity that is needed is driven by how many messages need to be retained at any time. If the broker is expected to receive 1 terabyte of traffic each day, with 7 days of retention, then the broker will need a minimum of 7 terabytes of useable storage for log segments. You should also factor in at least 10% overhead for other files, in addition to any buffer that you wish to maintain for fluctuations in traffic or growth over time.

Storage capacity will be one of the factors to consider when sizing a Kafka cluster, and determining when to expand it. The total traffic for a cluster can be balanced across it by having multiple partitions per topic, and this will allow additional brokers to shore up the available capacity if the density on a single broker will not suffice. The decision on how much disk capacity is needed will also be informed by the replication strategy chosen for the cluster (which is discussed in more detail in Chapter 5).

Memory

Aside from disk performance, the amount of memory available to the broker is the primary factor in client performance. Where disk performance primarily affects producers of messages, the memory available mostly affects consumers. The normal mode of operation for a Kafka consumer is reading from the end of the partitions, where it is caught up and lagging behind the producers very little, if at all. In this situation, the messages that the consumer is reading are optimally stored in the systems page cache, resulting in faster reads than if the broker must reread the messages from disk.

Kafka itself does not need very much heap memory configured for the Java Virtual Machine (JVM). Even a broker that is handling X messages per second and a data rate of X megabits per second can run with a 5 gigabyte heap. The rest of the system memory will be used by the page cache and will benefit Kafka. This is the main reason why it is not recommended to have Kafka colocated on a system with any other significant application, as this allows the page cache to continually be polluted, which will decrease performance.

Networking

The available network throughput will specify the maximum amount of traffic that Kafka can handle. This is often the governing factor, combined with disk storage, for cluster sizing. This is complicated by the inherent imbalance between inbound and outbound network usage that is created by Kafka's support for multiple consumers. A producer may write 1 MB per second in for a given topic, but there could be any number of consumers which creates a multiplier on the outbound network usage. Other operations, such as cluster replication (covered in Chapter 5) and mirroring (discussed in Chapter 7) will also increase requirements. Should the network interface become saturated, it is not uncommon for cluster replication to fall behind, which can leave the cluster in a vulnerable state.

CPU

Processing power is a lesser concern when compared to disk and memory, but it will affect overall performance of the broker to some extent. Ideally, clients should compress message to optimize network and disk usage. This does require that the Kafka broker decompress every message batch in order to assign offsets, and then recompress the message batch to store it on disk. This is where the majority of Kafka's requirement for processing power comes from. This should not be the primary factor in selecting hardware, however.

Kafka in the Cloud

A common installation for Kafka is within cloud computing environments, such as Amazon Web Services. Due to the different types of instances available, the various performance characteristics of Kafka must be prioritized in order to select the correct instance configuration to use. A good place to start is with the amount of data retention required, followed by the performance needed from the producers. If very low latency is necessary, I/O optimized instances that have local solid state disk storage may be required. Otherwise, ephemeral storage (such as the AWS Elastic Block Store) may be sufficient. Once these decisions are made, the CPU and memory options available will be appropriate for the performance.

In real terms, this will mean that for AWS either the m4 or r3 instance types are a common choice. The m4 instance will allow for greater retention periods, but the throughput to the disk will be less as it is on Elastic Block Storage. The r3 instance will have much better throughput, with local SSD drives, but those drives will limit the amount of data that can be retained. For the best of both worlds, it is necessary to move up to either the i2 or d2 instance types, which are significantly more expensive.

Kafka Clusters

A single Kafka server works well for local development work, or for a proof of concept system, but there are significant benefits to having multiple brokers configured as a cluster. The biggest benefit is the ability to scale the load across multiple servers. A close second is using replication to guard against data loss due to single system failures. This will also allow for performing maintenance work on Kafka, or the underlying systems, while still maintaining availability for clients. This section focuses on just configuring a Kafka cluster. Chapter 5 contains more more information on replication of data.

How Many Brokers

The appropriate size for a Kafka cluster is determined by several factors. The first concern is how much disk capacity is required for retaining messages and how much storage is available on a single broker. If the cluster is required to retain 10 terabytes of data, and a single broker can store 2 TB, then the minimum cluster size is 5 brokers. In addition, using replication will increase the storage requirements by at least 100%, depending on the replication factor chosen (Chapter 5). This means that this same cluster, configured with replication, now needs to contain at least 10 brokers.

The other consideration is the capacity of the cluster to handle requests. This is often required due to the capacity of the network interfaces to handle the client traffic, specifically if there are multiple consumers of the data or if the traffic is not consistent over the retention period of the data (e.g. bursts of traffic during peak times). If

the network interface on a single broker is used to 80% capacity at peak, and there are two consumers of that data, the consumers will not be able to keep up with peak traffic unless there are two brokers. If replication is being used in the cluster, this is an additional consumer of the data that must be taken into account. It may also be desirable to scale out to more brokers in a cluster in order to handle performance concerns caused by lesser disk throughput or system memory available.

Broker Configuration

There are only two requirements in the broker configuration to allow multiple Kafka brokers to join a single cluster. The first is that all brokers must have the same configuration for the `zookeeper.connect` parameter. This specifies the Zookeeper ensemble and path where the cluster stores metadata. The second requirement is that all brokers in the cluster must have a unique value for the `broker.id` parameter. If two brokers attempt to join the same cluster with the same `broker.id`, the second broker will log an error and fail to start. There are other configuration parameters used when running a cluster, specifically parameters that control replication, which are covered in later chapters.

Operating System Tuning

While most Linux distributions have an out-of-the-box configuration for the kernel tuning parameters that will work fairly well for most applications, there are a few changes that can be made for a Kafka broker that will improve performance. These primarily revolve around the virtual memory and networking subsystems, as well as specific concerns for the disk mount point that is used for storing log segments. These parameters are typically configured in the /etc/sysctl.conf file, but you should refer to your Linux distribution's documentation for specific details regarding how to adjust the kernel configuration.

Virtual Memory

In general, the Linux virtual memory system will automatically adjust itself for the work load of the system. The most impact that can be made here is to adjust how swap space is handled. As with most applications, specifically ones where throughput is a concern, the advice is to avoid swapping at (almost) all costs. The cost incurred by having pages of memory swapped to disk will show up as a noticeable impact in all aspects of performance in Kafka. In addition, Kafka makes heavy use of the system page cache, and if the VM system is swapping to disk, this shows that there is certainly not enough memory being allocated to page cache.

One way to avoid swapping is just to not configure any swap space at all. Having swap is not a requirement, but it does provide a safety net if something catastrophic happens on the system. Having swap can prevent the operating system from abruptly

killing a process due to an out of memory condition. For this reason the recommendation is to set the `vm.swappiness` parameter to a very low value, such as 1. The parameter is a percentage of how likely the VM subsystem is to use swap space, rather than dropping pages from the page cache. It is preferable to reduce the size of the page cache rather than swap.

> Previously, the recommendation for `vm.swappiness` was always to set it to 0. This value used to have the meaning "do not swap unless there is an out of memory condition". However, the meaning of this value changed as of Linux kernel version 3.5-rc1, and that change was back ported into many distributions, including Red Hat Enterprise Linux kernels as of version 2.6.32-303. This changed the meaning of the value 0 to "never swap under any circumstances". It is for this reason that a value of 1 is now recommended.

There is also a benefit to adjusting how the kernel handles dirty pages that must be flushed to disk. Kafka relies on disk I/O performance to provide a good response time to producers. This is also the reason that the log segments are usually put on a fast disk, whether that is an individual disk with a fast response time (e.g. SSD) or a disk subsystem with signficant NVRAM for caching (e.g. RAID). This allows the amount of dirty pages allowed before the flush background process starts writing them to disk to be reduced, by setting the =vm.dirty_background_ratio+ value lower than the default of 10. This value is a percentage of the total amount of system memory, and setting this value to 5 is appropriate in many situations. This setting should not be set to zero, however, as that would cause the kernel to continually flush pages, which would eliminate the ability for the kernel to buffer disk writes against temporary spikes in the underlying device performance.

The total number of dirty pages that are allowed before the kernel forces synchronous operations to flush them to disk can also be increased by changing the value of `vm.dirty_ratio`, increasing it above the default of 20 (also a percentage of total system memory). There is a wide range of possible values for this setting, but between 60 and 80 is a reasonable number. This setting does introduce a small amount of risk, both with regards to the amount of unflushed disk activity as well as the potential for long I/O pauses if synchronous flushes are forced. If a higher setting for `vm.dirty_ratio` is chosen, it is highly recommended that replication be used in the Kafka cluster to guard against system failures.

When choosing values for these parameters, it is wise to review the number of dirty pages over time while the Kafka cluster is running under load, whether production or simulated. The current number of dirty pages can be determined by checking the `/proc/vmstat` file:

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 3875
nr_writeback 29
nr_writeback_temp 0
#
```

Disk

Outside of selecting the disk device hardware, as well as the configuration of RAID if it is used, the choice of filesystem used for this disk can have the next largest impact on performance. There are many different filesystems available, but the most common choices for local filesystems are either EXT4 (Fourth extended file system) or XFS. Recently XFS has become the default filesystem for many Linux distributions, and this is with good reason - it outperforms EXT4 for most workloads with minimal tuning required. EXT4 can perform well, but it requires using tuning parameters that can be considered less safe. This includes setting the commit interval to a longer time than the default of 5 to force less frequent flushes. EXT4 also introduced delayed allocation of blocks, which brings with it a greater chance of data loss and filesystem corruption in the case of a system failure. The XFS filesystem also uses a delayed allocation algorithm, but it is generally safer than that used by EXT4. XFS also has better performance for Kafka's work load without requiring tuning beyond the automatic tuning performed by the filesystem. It is more efficient when batching disk writes, all of which combines to give better overall I/O throughput.

Regardless of which filesystem is chosen for the mount which holds the log segments, it is advisable to set the `noatime` mount option for the mount point. File metadata contains three timestamps: creation time (ctime), last modified time (mtime), and last access time (atime). By default, the atime is updated every time a file is read. This generates a large number of disk writes and the atime attribute is generally considered to be of little use, unless an application needs to know if a file has been accessed since it was last modified (in which case the `relatime` option can be used). It is not used by Kafka at all, which means disabiling setting of the last access time entirely is safe. Setting `noatime` on the mount will prevent these timestamp updates from happening, but this does not affect the proper handling of the ctime and mtime attributes.

Networking

Adjusting the default tuning of the Linux networking stack is common for any application which generates a high amount of network traffic, as the kernel is not tuned by default for large, high speed data transfers. In fact, the recommended changes for Kafka are the same as are suggested for most web servers and other networking applications. The first adjustment is to change the default and maximum amount of memory allocated for the send and receive buffers for each socket. This will increase performance significantly for large transfers. The relevant parameters for the send

and receive buffer default size per socket are `net.core.wmem_default` and `net.core.rmem_default`, and a reasonable setting for these parameters is 131072, or 128 kibibytes. The parameters for the send and receive buffer maximum sizes are `net.core.wmem_max` and `net.core.rmem_max`, and a reasonable setting is 2097152, or 2 mebibytes. Keep in mind that the maximum size does not indicate that every socket will have this much buffer space allocated, it only allows up to that much if needed.

In addition to the socket settings, the send and receive buffer sizes for TCP sockets must be set separately using the `net.ipv4.tcp_wmem` and `net.ipv4.tcp_rmem` parameters. These are set using 3 space-separated integers that specify the minimum, default, and maximum sizes respectively. The maximum size cannot be larger than the values specified for all sockets using `net.core.wmem_max` and `net.core.rmem_max`. An example setting for each of these parameters is "4096 65536 2048000", which is a 4 kibibyte minimum buffer, 64 kibibyte default, and 2 mebibyte maximum. Based upon the actual workload that your Kafka brokers receive, you may want to increase the maximum sizes higher to allow for greater buffering of the network connections.

There are several other network tuning parameters that are useful to set. Enabling TCP window scaling by setting `net.ipv4.tcp_window_scaling` to 1 will allow clients to transfer data more efficiently, and allow it to be buffered on the broker side. Increasing the value of `net.ipv4.tcp_max_syn_backlog` above the default of 1024 will allow a greater number of simultaenous connections to be accepted. Increasing the value of `net.core.netdev_max_backlog` to greater than the default of 1000 can assist with bursts of network traffic, specifically when using multi-gigabit network connection speeds, by allowing more packets to be queued up for the kernel to process them.

Production Concerns

Once you are ready to move your Kafka environment out of testing and into your production operations, there are a few more things to think about that will assist with setting up a reliable messaging service.

Garbage Collector Options

Tuning the Java garbage collection options for an application has always been something of an art, requiring detailed information about how the application uses memory and a significant amount of observation and trial and error. Thankfully this has changed with Java 7 and the introduction of the Garbage First (or G1) garbage collector. G1 is designed to automatically adjust to different workloads and provide consistent pause times for garbage collection over the lifetime of the application. It also

handles large heap sizes with ease, by segmenting the heap into smaller zones and not collecting over the entire heap in each pause.

G1 does all of this with a minimal amount of configuration in normal operation. There are two configuration options for G1 that are often used to adjust its performance. These are:

- `MaxGCPauseMillis`: This option specifies the preferred pause time for each garbage collection cycle. It is not a fixed maximum - G1 can and will exceed this time if it is required. This value defaults to 200 milliseconds. This means that G1 will attempt to schedule the frequency of GC cycles, as well as the number of zones that are collected in each cycle, such that each cycle will take approximately 200ms.
- `InitiatingHeapOccupancyPercent`: This option specifies the percentage of the total heap that may be in use before G1 will start a collection cycle. The default value is 45. This means that G1 will not start a collection cycle until after 45% of the heap is in use. This includes both the new (eden) and old zone usage in total.

The Kafka broker is fairly efficient with the way it utilizes heap memory and creates garbage objects, so it is possible to set these options lower. The particular tuning used here has been found to be appropriate for a server with 64 gigabytes of memory, running Kafka in a 5 gigabyte heap. For `MaxGCPauseMillis`, this broker can be configured with a value of 20 milliseconds. The value for `InitiatingHeapOccupancyPercent` is set to 35, which causes garbage collection to run slightly earlier than with the default value.

The start script for Kafka does not use the G1 collector, instead defaulting to using Parallel New and Concurrent Mark and Sweep garbage collection. The change is easy to make via environment variables. Using the start command from earlier in the chapter, modify it as such:

```
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+DisableExplicitGC -Djava.awt.headless=true"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Datacenter Layout

For development systems, the physical location of the Kafka brokers within a datacenter is not as much of a concern, as there is not as severe an impact if the cluster is partially or completely unavailable for short periods of time. When serving production traffic, however, downtime means dollars lost, whether through loss of services

to users or loss of telemetry on what the users are doing. This is when it becomes critical to configure replication within the Kafka cluster (see Chapter 5), which is also when it is important to consider the physical location of brokers in their racks in the datacenter. If not addressed prior to deploying Kafka, it can mean expensive maintenances to move servers around.

The Kafka broker has no rack-awareness when assigning new partitions to brokers. This means that it cannot take into account that two brokers may be located in the same physical rack, or in the same availability zone (if running in a cloud service like AWS), and therefore can easily assign all replicas for a partition to brokers that share the same power and network connections in the same rack. Should that rack have a failure, these partitions would be offline and inaccessible to clients. In addition, it can result in additional lost data on recovery due to an unclean leader election (more about this in Chapter 5).

The best practice is to have each Kafka broker in a cluster installed in a different rack, or at the very least not sharing single points of failure for infrastructure services such as power and network. This typically means at least deploying the servers that will run brokers with dual power connections (to two different circuits) and dual network switches (with a bonded interface on the servers themselves to fail over seamlessly). Even with dual connections, there is a benefit to having brokers in completely separate racks. From time to time, it may be neccessary to perform physical maintenance on a rack or cabinet that requires it to be offline (such as moving servers around, or rewiring power connections).

Colocating Applications on Zookeeper

Kafka utilizes Zookeeper for storing metadata information about the brokers, topics, and partitions. The Kafka consumer client stores information about the composition of the consumer group and the topics it consumers there (see Chapter X regarding consumers for more information about how the client uses Zookeeper). This does not represent enough traffic for a Zookeeper ensemble with multiple nodes for the ensemble to be dedicated to just that Kafka cluster, as the writes are only on changes to the makeup of the consumer group or the Kafka cluster itself. In fact, many deployments will use a single Zookeeper ensemble for multiple Kafka clusters (using a chroot Zookeeper path for each cluster, as described earlier in this chapter).

There is a concern with consumers and Zookeeper under certain configurations, however. Consumers have a configurable choice to use either Zookeeper or Kafka for committing offsets, as well as the interval between commits. If the consumer uses Zookeeper for offsets (which has been the default and is still the best choice, due to continuing development on Kafka-committed offsets), each consumer will perform a Zookeeper write every interval for every partition it consumes. A reasonable interval for offset commits is one minute, as this is the period of time over which a consumer

group will read duplicate messages in the case of a consumer failure. These commits can be a significant amount of Zookeeper traffic, especially in a cluster with many consumers, and will need to be taken into account. It may be neccessary to use a longer commit interval if the Zookeeper ensemble is not able to handle the traffic.

Outside of using a single ensemble for multiple Kafka clusters, it is not recommended to share the ensemble with other applications, if it can be avoided. Kafka is sensitive to Zookeeper latency and timeouts, and an interruption in communications with the ensemble will cause the brokers to behave unpredictably. This can easily cause multiple brokers to go offline at the same time, should they lose Zookeeper connections, which will result in offline partitions. It also puts stress on the cluster controller, which can show up as subtle errors long after the interruption has passed, such as when trying to perform a controlled shutdown of a broker. Other applications which can put stress on the Zookeeper ensemble, either through heavy usage or improper operations, should be segregated to their own ensemble.