Chapter 10. Monitoring Kafka

The Apache Kafka applications maintain numerous measurements regarding their operation, so many that it can easily become confusing as to what is important to watch and what can be safely set aside. These range from simple metrics about the overall rate of traffic, to detailed timing metrics for every request type, to per-topic and per-partition metrics. They provide a detailed view into every operation in the broker, but they can also make you the bane of whomever is responsible for managing your monitoring system.

This section will detail the most critical metrics to monitor all the time, and how to respond to them. We'll also detail some of the more important metrics to have on hand when debugging problems. This is not to be considered an exhaustive list of the metrics that are available, however. The list changes frequently, and many will only be informative for a hardcode Kafka developer.

Metric Basics

Before getting into the specific metrics provided by the Kafka broker and clients, let us discuss the basics of how to monitor Java applications, and some best practices around monitoring and alerting. This will provide a basis for understanding how to monitor the applications, and why the specific metrics described later in this chapter are the ones that have been chosen as the most important.

Where Are the Metrics?

All of the metrics that are exposed by Kafka can be accessed via the Java Management Extensions (JMX) interface. The easiest way to use them in an external monitoring system is to use a collection agent provided by your monitoring system and attach it to the Kafka process. This may be a separate process that runs on the system and connects to the JMX interface, such as with the Nagios XI check_jmx plugin, or jmxtrans. You may also utilize a JMX agent that runs directly in the Kafka process to access metrics via an HTTP connection, such as Jolokia or MX4J.

An in-depth discussion of how to set up monitoring agents is outside the scope of this chapter, and there are far too many choices to do justice to all of them. If your organization does not currently have experience with monitoring Java applications, it may be worthwhile to instead consider monitoring as a service. Using this model, you would purchase services from one of many companies that provide monitoring agents, metrics collection points, storage, graphing, and alerting as a package. They can assist you further with setting up the monitoring agents required.

FINDING THE JMX PORT

For programmatic access to JMX on the Kafka broker, such as for administrative tooling that needs to discover how to connect without having a configured port, the broker sets the configured JMX port as part of the broker information stored in Zookeeper. The `/brokers/ids/<ID>` znode contains JSON-formatted data for the broker that includes `hostname` and `jmx_port` keys.

Internal or External Measurements

Metrics provide via an interface such as JMX are internal metrics: they are created and provided by the application that is being monitored. For many internal measurements, such as timing of individual request stages, this is the best choice. Nothing other than the application itself has that level of detail. There are other metrics, such as the overall time for a request or the availability of a particular request type, that can be measured externally. This would mean that the client application, or some other 3rd party application, provides the metrics for the server (the broker, in our case). This provides an external view of the application that is often more informative.

A familiar example for the value of external measurements would be monitoring the health of a website. The web server itself may be running properly and able to serve requests. It is reporting metrics back to your monitoring system and everything looks good. However, there is a problem with firewalls, whether on the web server itself or in the networking leading to the web server, and no clients can connect to it. External monitoring that checks the accessibility of the website would detect this and alert you to the situation.

Application Health Checks

Whichever way you collect monitoring from Kafka, you should make sure that you have a way to also monitor the overall health of the application process via a simple healthcheck. You may be able to detect the lack of monitoring coming from your application (also known as “stale” metrics). It is still helpful to have an independent check that the process is running correctly. This can help to differentiate between a failure of the monitoring system and a failure of Kafka itself.

For the Kafka broker, this can simply be connecting to the external port (the same port that clients use to connect to the broker) to check that it responds. For client applications it can be more complex, ranging from a simple check of whether or not the process is running to an internal method that determines application health.

Metric Coverage

Especially when considering the number of measurements exposed by Kafka, it is important to pick and choose what you look at. This becomes even more important when defining alerts based on those measurements. It is far to easy to succumb to “alert fatigue”, where there are so many different alerts going off that it is difficult to know how severe the problem is. It is also hard to properly define thresholds for every metric and keep them up to date. In this situation, we reduce trust in the alerts.

It is more advantageous to have a few alerts that have a high level coverage. That is, when there is one alert that indicates that there is definitely a big problem, but where you may have to gather additional data to determine the exact nature of that problem. Think of this like the “Check Engine” light on a car. It would be confusing to have 100 different indicators on the dashboard that show individual problems with the air filter, the oil, the exhaust, and so on. Instead, there is one indicator that tells you that there is a problem, and there is a way to find out more detailed information to tell you exactly what the problem is. Throughout this chapter, we will identify the metrics that will provide the highest amount of coverage to keep your alerting simple.

Kafka Broker Metrics

For the Kafka broker itself, there are many metrics that are available. Many of them are low-level measurements, added by developers when investigating a specific issue or in anticipating of needing the information for debugging purposes later, or requested by those running Kafka to provide insight into every day operations. There are metrics providing information about nearly every function within the broker. This can easily overwhelm us with information, but there are several metrics that will provide the information needed to run Kafka on a daily basis.

Example 10-1. Who Watches the Watchers?

One of the use cases for Kafka, implemented in many organizations, is to use it for collecting application and system metrics and logs for consumption by a central monitoring system. This is an excellent way to decouple the applications from the monitoring system, but it presents a specific concern for Kafka itself. If you use this same system for monitoring Kafka itself, it is very likely that you will never know when Kafka is broken, because the data flow for your monitoring system will be broken as well.

There are many ways that this can be addressed. One way is to use a separate monitoring system for Kafka that does not have a dependency on Kafka. Another way, if you have multiple datacenters, is to make sure that the metrics for the Kafka cluster in datacenter A are produced to datacenter B, and vice versa. However you decide to handle it, make sure that the monitoring and alerting for Kafka does not depend on Kafka working.

In this section, we'll start by discussing the under-replicated partitions metric as an overall performance measurement, as well as how to respond to it. The other metrics discussed will round out the view of the broker at a high level. This is by no means an exhaustive list of broker metrics, but rather several "must have" numbers for checking on the health of the broker and the cluster. We'll wrap up with a discussion on logging before moving on to client metrics.

Under Replicated Partitions

Metric Name	Under-replicated Partitions
JMX MBean	`kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`
Value Range	Integer, zero or greater

If there is only one metric that you are able to monitor from the Kafka broker, it should be the number of under-replicated partitions. This measurement, provided on each broker in a cluster, gives a count of the number of partitions for which that broker is the leader where the replicas are not caught up. This single measurement provides insight into a number of problems with the Kafka cluster, from a broker being down to resource exhaustion. With the wide variety of problems that this metric can indicate, it is worthy of an in-depth look at how to respond to a value other than zero. Please note that many of the metrics used in diagnosing these types of problems will be described later in this chapter.

A steady (unchanging) number of under-replicated partitions reported by many of the brokers in a cluster will normally indicate that one of the brokers in the cluster is offline. The count of under-replicated partitions across the entire cluster will equal the number of partitions that are assigned to that broker, and the broker that is down will not be reporting a metric. In this case, you will need to investigate what has happened to that broker and resolve that situation. This is often a hardware failure, but could also be an operating system or Java issue as well that has caused the process to go away or pause.

PREFERRED REPLICA ELECTIONS

Before going any further trying to diagnose a problem, have you run a preferred replica election (see Chapter 9) recently? Kafka brokers do not automatically take partition leadership back (unless auto leader rebalance is enabled, but this configuration is not recommended) after they have released leadership (for example, when the broker has failed or been shut down). This means that it's very easy for leader replicas to become unbalanced in a cluster. The preferred replica election is safe and easy to run, so it's a good idea to do that first and see if the problem goes away.

If the number of under-replicated partitions is fluctuating, or if the number is steady but there are no brokers offline, this typically indicates a performance issue in the cluster. These types of problems are much harder to diagnose due to their variety, but there are

several steps you can worth through to narrow it down to the most likely causes. The first item to try and determine is if the problem relates to a single broker, or to the entire cluster. This can sometimes be a difficult question to answer. If the under-replicated partitions are on a single broker, then that broker is typically the problem. It shows that other brokers are having a problem replicating messages from that one.

If several brokers have under-replicated partitions, it could be a cluster problem, but it might still be a single broker. In that case, it would be because a single broker is having problems replicating messages from everywhere. You will need to isolate which broker that is. One way to do this is to get a list of under-replicated partitions for the cluster and see if there is a commonality among them. Using the *kafka-topics.sh* tool (discussed in detail in Chapter 9), you can get a list of under-replicated partitions to look for a common thread.

Example: List under-replicated partitions in a cluster

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe --under-replicated
    Topic: topicOne   Partition: 5    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicOne   Partition: 6    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicTwo   Partition: 3    Leader: 4    Replicas: 2,4 Isr: 4
    Topic: topicTwo   Partition: 7    Leader: 5    Replicas: 5,2 Isr: 5
    Topic: topicSix   Partition: 1    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicSix   Partition: 2    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicSix   Partition: 5    Leader: 6    Replicas: 2,6 Isr: 6
    Topic: topicSix   Partition: 7    Leader: 7    Replicas: 7,2 Isr: 7
    Topic: topicNine  Partition: 1    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicNine  Partition: 3    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 4    Leader: 3    Replicas: 3,2 Isr: 3
    Topic: topicNine  Partition: 7    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 0    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 5    Leader: 6    Replicas: 6,2 Isr: 6
#
```

In this example, the common broker is number 2. It is present in all the replica lists, however it is missing from all the in-sync replica (ISR) lists. This will lead us to focus our investigation on that one broker. If instead we were to find no common broker, a cluster-wide problem is the more likely answer.

CLUSTER-LEVEL PROBLEMS

Cluster problems usually fall into two categories:

- Unbalanced Load
- Resource Exhaustion

The first problem, unbalanced partitions or leadership, is the easiest to find even though fixing it can be an involved process. In order to diagnose this problem, you will need several metrics from the brokers in the cluster:

- Partition Count
- Leader Partition Count
- All Topics Bytes In Rate
- All Topics Messages In Rate

Examine these metrics. In a perfectly balanced cluster, the numbers will be even (within reason) across all brokers in the cluster. This indicates that all the brokers are taking approximately the same amount of traffic. Assuming you have already run a preferred replica election, a large deviation indicates that the traffic is not balanced within the cluster. To resolve this, you will need to move partitions from the heavily loaded brokers to the less loaded brokers. This is done using the *kafka-reassign-partitions.sh* tool described in the previous chapter.

HELPERS FOR BALANCING CLUSTERS

The Kafka broker itself does not provide for automatic reassignment of partitions in a cluster. This means that balancing traffic within a Kafka cluster can be a mind-numbing process of manually reviewing long lists of metrics and trying to come up with a replica assignment that works. In order to help with this, some organizations have developed automated tools for performing this task. One example is the *kafka-assigner* tool that LinkedIn has released in the open source kafka-tools repository. Some enterprise offerings for Kafka support also provide this feature.

Another common cluster performance issue is just that you are just out of capacity. There are many possible bottlenecks that could slow things down: CPU, disk IO, and network throughput are a few of the most common. Disk utilization is not one of them, as the brokers will operate properly right up until the disk is filled, and then it will fail abruptly. In order to diagnose a capacity problem, there are many metrics you can track at the operating system level, including:

- CPU Utilization
- Inbound Network Throughput
- Outbound Network Throughput
- Disk Average Wait Time
- Disk Percent Utilization

Exhausting any of these resources will end up presenting as under-replicated partitions, and it's critical to remember that the broker replication process is just another Kafka client. If your cluster is having problems with replication, then your customers are having problems with producing and consuming messages as well. It makes sense to develop a baseline for these metrics when your cluster is operating correctly, and then set thresholds that indicate a developing problem long before you run out of capacity. You will also want to review the trend for these metrics as the traffic to your cluster increases over time. As far as Kafka broker metrics are concerned, the `All Topics Bytes In Rate` is a good guideline to show cluster usage.

HOST-LEVEL PROBLEMS

If the performance problem with Kafka is not present in the entire cluster, and can be isolated to one or two brokers, it's time to examine that server and see what makes it different from the rest of the cluster. These types of problems fall into several general categories:

- Hardware failures
- Conflicts with another process
- Local configuration differences

TYPICAL SERVERS AND TYPICAL PROBLEMS

A server and its operating system are a complex machine with thousands of components, any of which could have problems and cause either a complete failure or just a performance degradation. It's impossible for us to cover everything that can fail in this book - numerous volumes have been written, and will continue to be, on this subject. We can discuss some of the most common problems that are seen, however. In specific, this section will focus on issues with a typical server running a Linux operating system.

Hardware failures are sometimes obvious, as the server just stops working, but it's the less obvious problems that are the ones that cause performance issues. These are usually soft failures that allow the system to keep running, but degrade operation. This could be

a bad bit of memory, where the system has detected the problem and bypassed that segment (reducing the overall available memory). The same can happen with a CPU failure. For items such as these, you should be using the facilities that your hardware provides, such as an Intelligent Platform Management Interface (IPMI) to monitor hardware health. When there's an active problem, looking at the kernel ring buffer using `dmesg` will help you to see log messages that are getting thrown to the system console.

The more common type of hardware failure that leads to a performance degradation in Kafka is a disk failure. Apache Kafka is dependent on the disk for persistence of messages, and producer performance is directly tied to how fast your disks commit those writes. Any deviation in this will show up as problems with the performance of the producers and the replica fetchers. The latter is what leads to under-replicated partitions. As such, it is important to monitor the health of the disks at all times and address any problems quickly.

> **ONE BAD EGG**
>
> A single disk failure on a single broker can destroy the performance of an entire cluster. This is because the producer clients will connect to all brokers that lead partitions for a topic, and if you have followed best practices those partitions are evenly spread over the entire cluster. If one broker starts performing poorly and slowing down produce requests, this will cause backpressure in the producers, slowing down requests to all brokers.

To begin with, assure you are monitoring hardware status information for the disks from IPMI, or the interface provided by your hardware. In addition, within the operating system you should be running SMART (Self-Monitoring, Analysis and Reporting Technology) tools to both monitor and test the disks on a regular basis. This will alert you to a failure that is about to happen. It is also important to keep an eye on the disk controller, especially if it has RAID functionality, whether you are using hardware RAID or not. Many controllers have on-board cache which is only used when the controller is healthy and the battery backup unit is working. A failure of the BBU can result in the cache being disabled, degrading disk performance.

Networking is another area where partial failures will cause problems. Some of these problems are hardware issues, such as a bad network cable or connector. Some are configuration issues, which is usually a change in the speed or duplex settings for the connection, either on the server side or upstream on the networking hardware. Network configuration problems could also be operating system issues, such as having the network buffers undersized, or too many network connections taking up too much of the overall memory footprint. One of the key indicators of problems in this area will be the number of errors detected on the network interfaces. If the error count is increasing, there is probably an unaddressed issue.

If there are no hardware problems, another common problem to look for is another application running on the system that is consuming resources and putting pressure on the Kafka broker. This could be something that was installed in error, or it could be a process that is supposed to be running, such as a monitoring agent, but is having problems. Utilize the tools on your system, such as `top`, to identify if there is a process that is using more CPU or memory than expected.

If the other options have been exhausted and you have not yet found the source of the discrepancy on the host, it is likely a configuration difference that has crept in, either with the broker or the system itself. Given the number of applications that are running on any single server, and the number of configuration options for each of them, it can be a daunting task to find a discrepancy. This is why it is cruical that you utilize a configuration management system, such as Chef or Puppet, in order to maintain consistent configurations across both your operating systems and your applications (including Kafka).

Broker Metrics

In addition to under-replicated partitions, there are other metrics that are present at the overall broker level that should be monitored. While you may not be inclined to set alert thresholds for all of them, they provide valuable information about your broker and your cluster. They should be present in any monitoring dashboard you create.

ACTIVE CONTROLLER COUNT

Metric Name	Active Controller Count
JMX MBean	`kafka.controller:type=KafkaController,name=ActiveControllerCount`
Value Range	Zero or one

The active controller count metric indicates whether or not the broker is currently the controller for the cluster. The metric will either be zero or one, with one showing that the broker is currently the controller. At all times, only one broker should be the controller, and one broker must always be the controller in the cluster. If two brokers say that they are currently the controller, this means that you have a problem where a controller thread that should have exited has become stuck. This can cause problems with not being able to execute administrative tasks, such as partition moves, properly. To remedy this, you will need to restart both brokers at the very least, and they may not shut down in a controlled manner because of the extra controller.

If no broker claims to be the controller in the cluster, the cluster will fail to respond properly in the face of state changes, including topic or partition creation, or broker failures. In this situation you must investigate further to find out why the controller threads are not working properly. For example, a network partition from the Zookeeper cluster could result in a problem like this. Once that underlying problem is fixed, it is wise to restart all the brokers in the cluster in order to reset state for the controller threads.

REQUEST HANDLER IDLE RATIO

Metric Name	Request Handler Average Idle Percent
JMX MBean	`kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent`
Value Range	Float, betweeen zero and one inclusive

Kafka uses two thread pools for handling all client requests: network handlers and request handlers. The network handler threads are responsible for reading and writing data to the clients across the network. This does not require a lot of work, which means that we do not have to be as concerned about the utilization of these threads. The request handler threads, however, are responsible for servicing the client request itself, which includes reading or writing the messages to disk. As such, as the brokers get more heavily loaded there is a significant impact on this thread pool.

> **INTELLIGENT THREAD USAGE**
>
> While it may seem like you will need hundreds of request handler threads, in reality you do not need to configure any more threads than you have CPUs in the broker. Apache Kafka is very smart about the way it uses the request handlers, making sure to offload requests that will take a long time to process to purgatory. This is used, for example, when requests are being quotaed or when more than one acknowledgement of produce requests is required.

The request handler idle ratio metric indicates the percentage of time the request handlers are not in use. The lower this number, the more loaded the broker is. Experience tells us that idle ratios lower than 20% indicate a potential problem, and lower than 10% is usually an active performance problem. Besides the cluster being undersized, there are two reasons you will see for high thread

utilization in this pool. The first is that there are not enough threads in the pool. In general, you should set the number of request handler threads equal to the number of processors in the system (including hyperthreaded processors).

The other common reason for high request handler thread utilization is that the threads are doing unnecessary work for each request. Prior to Kafka 0.10, the request handler thread was responsible for decompressing every incoming message batch, validating the messages and assigning offsets, and then recompressing the message batch with offsets before writing it to disk. To make matters worse, the compression methods were all behind a synchronous lock. As of version 0.10, a new message format was introduced that allows for relative offsets in a message batch. This means that newer producers will set relative offsets prior to sending the message batch, which allows the broker to skip recompression of the message batch. One of the single largest performance improvements you can make is to assure all producer and consumer clients support the 0.10 message format, and to change the message format version on the brokers to 0.10 as well. This will greatly reduce the utilization of the request handler threads.

ALL TOPICS BYTES IN

Metric Name	Bytes In Per Second
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec`
Value Range	Rates as doubles, count as integer

The all topics bytes in rate, expressed in bytes per second, is useful as a measurement of how much message traffic your brokers are receiving from producing clients. This is a good metric to trend over time to help you determine when you need to expand the cluster or do other growth-related work. It is also useful for evaluating if one broker in a cluster is receiving more traffic than the others, which would indicate that it is necessary to rebalance the partitions in the cluster.

As this is the first rate metric discussed, it is worth a short discussion of the attributes that are provided. All of the rate metrics have seven attributes that are provided, and choosing which ones to use is going to depend on what type of measurement you want. They can provide a discrete count or an average over varying periods of time. Make sure to use the metrics appropriately, or you will end up with a flawed view of the broker.

The first two attributes are not measurements, but they will help you understand the metric you are looking at:

- `EventType` - This is the unit of measurement for all the attributes. In this case it is “bytes”
- `RateUnit` - For the rate attributes, this is the time period for the rate. In this case it is “SECONDS”

These two descriptive attributes tell us that the rates, regardless of the period of time they average over, are presented as a value of bytes per second. There are four rate attributes that are provided with different granularities:

- `OneMinuteRate` - An average over the previous one minute
- `FiveMinuteRate` - An average over the previous five minutes
- `FifteenMinuteRate` - An average over the previous fifteen minutes
- `MeanRate` - An average since the broker was started

The `OneMinuteRate` will fluctuate quickly and provides more of a “point in time” view of the measurement. This is useful for seeing short spikes in traffic. The `MeanRate` will not vary much at all and provides an overall trend. While this can have its uses, it is probably not the metric you want to be alerted on. The `FiveMinuteRate` and `FifteenMinuteRate` provide a compromise in the middle.

In addition to the rate attributes, there is a `Count` attribute as well. This is a constantly increasing value for the metric since the time the broker was started. For this metric, all topics bytes in, the `Count` represents the total number of bytes produced to the broker since the process was started. Utilized with a metrics system that supports counter metrics, this can give you an absolute view of the measurement, rather than an averaged rate.

ALL TOPICS BYTES OUT

Metric Name	Bytes Out Per Second
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec`
Value Range	Rates as doubles, count as integer

The all topics bytes out rate, similar to the bytes in rate, is another overall growth metric. In this case, the bytes out rate shows the rate at which consumers are reading messages out. The outbound bytes rate may scale differently than the inbound bytes rate, thanks to Kafka's capacity to handle multiple consumers with ease. There are many deployments of Kafka where the outbound rate can easily be 6 times the inbound rate! This is why it is important to observe and trend the outbound bytes rate separately.

> **REPLICA FETCHERS INCLUDED**
>
> The outbound bytes rate *also* includes the replica traffic. This means that if all of the topics are configured with a replication factor of 2, then you will see a bytes out rate equal to the bytes in rate when there are no consumer clients. If you have one consumer client that is reading all the messages in the cluster, then the bytes out rate will be twice the bytes in rate. This can be confusing when looking at the metrics if you're not aware of what is counted.

ALL TOPICS MESSAGES IN

Metric Name	Messages In Per Second
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec`
Value Range	Rates as doubles, count as integer

While the bytes rates described previously show the broker traffic in absolute terms of bytes, the messages in rate shows the number of individual messages, regardless of their size, produced per second. This is useful as a growth metric as a different measure of producer traffic. It can also be used in conjunction with the bytes in rate to determine an average message size. You may also see an imbalance in the brokers, just like with the bytes in rate, that will alert you to maintenance work that is needed.

WHY NO MESSAGES OUT?

The question is often raised as to why there is no "Messages Out" metric for the Kafka broker. The reason is that when messages are consumed, the broker just sends the next batch to the consumer without expanding it to find out how many messages are inside. Therefore, the broker doesn't really know how many messages were sent out. The only metric that can be provided is the number of fetches per second, which is a request rate, not a messages count.

PARTITION COUNT

Metric Name	Partition Count
JMX MBean	`kafka.server:type=ReplicaManager,name=PartitionCount`
Value Range	Integer, zero or greater

The partition count for a broker generally doesn't change that much, as it is the total number of partitions assigned to that broker. This includes every replica the broker has, regardless of whether it is a leader or follower for that partition. Monitoring this is often more interesting in a cluster that has automatic topic creation enabled, as that can leave the creation of topics outside of the control of the person who is running the cluster.

LEADER COUNT

Metric Name	Leader Count
JMX MBean	`kafka.server:type=ReplicaManager,name=LeaderCount`
Value Range	Integer, zero or greater

The leader count metric shows the number of partitions that the broker is currently the leader for. As with most other measurements in the brokers, this one should be generally even across the brokers in the cluster. It is much more important to check the leader count on a regular basis, possibly alerting on it, as it will indicate when the cluster is imbalanced even if the number of replicas are perfectly balanced in count and size across the cluster. This is because a broker can drop leadership for a partition for many reasons, such as a Zookeeper session expiration, and it will not automatically take leadership back once it recovers (except if you have enabled automatic leader rebalancing). In these cases, this metric will show fewer leaders, or often zero. This indicates that you need to run a preferred replica election to rebalance leadership in the cluster.

A useful way to consume this metric is to use it along with the partition count to show a percentage of partitions that the broker is the leader for. In a well-balanced cluster that is using a replication factor of two, all brokers should be leaders for approximately 50% of their partitions. If the replication factor in use is three, this percentage drops to 33%.

OFFLINE PARTITIONS

Metric Name	Offline Partitions Count
JMX MBean	`kafka.controller:type=KafkaController,name=OfflinePartitionsCount`
Value Range	Integer, zero or greater

Along with the under-replicated partitions count, the offline partitions count is a critical metric for monitoring. This measurement is only provided by the broker that is the controller for the cluster (all other brokers will report zero), and shows the number of partitions in the cluster that currently have no leader. This can happen for two main reasons:

- All brokers hosting replicas for this partition are down
- No in-sync replica can take leadership due to message count mismatches (with unclean leader election disabled)

In a production Kafka cluster, an offline partition is a bad situation that may be impacting the producer clients, losing messages or causing back-pressure in the application. This is most often a "site down" type of problem and will need to be addressed immediately.

REQUEST METRICS

The Kafka protocol, described in Chapter 5, has many different requests and there are metrics provided for how each of those requests performs. The following requests have metrics provided:

- `ApiVersions`
- `ControlledShutdown`
- `CreateTopics`
- `DeleteTopics`
- `DescribeGroups`
- `Fetch`
- `FetchConsumer`
- `FetchFollower`
- `GroupCoordinator`
- `Heartbeat`
- `JoinGroup`
- `LeaderAndIsr`
- `LeaveGroup`
- `ListGroups`
- `Metadata`

- `OffsetCommit`
- `OffsetFetch`
- `Offsets`
- `Produce`
- `SaslHandshake`
- `StopReplica`
- `SyncGroup`
- `UpdateMetadata`

For each of these requests, there are eight metrics provided, providing insight into each of the phases of the request processing. For example, for the `Fetch` request, the following metrics are available:

Table 10-1. Request Metrics

Name	JMX MBean
Total Time	`kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Fetch`
Request Queue Time	`kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=Fetch`
Local Time	`kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Fetch`
Remote Time	`kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Fetch`
Throttle Time	`kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=Fetch`
Response Queue Time	`kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Fetch`
Response Send Time	`kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=Fetch`
Requests Per Second	`kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Fetch`

The *Requests Per Second* metric is a rate metric, for which we have previously describe the attributes, which shows the total number of that type of request that has been received and processed over the time unit. This provides a view into the frequency of each request time, though it should be noted that many of the requests, such as `StopReplica` and `UpdateMetadata`, can be infrequent.

The seven *Time* metrics each provide a set of percentiles for requests, as well as a discrete `Count` attribute, similar to rate metrics. The metrics are all calculated since the broker was started, so keep that in mind when looking at metrics that do not change for long

periods of time; the longer your broker has been running, the more stable the numbers will be. The parts of request processing they represent are:

- *Total Time* measures the total amount of time the broker spends processing the request, from receiving it to sending the response back to the requestor.
- *Request Queue Time* is the amount of time the request spends in queue after it has been received, but before processing starts
- *Local Time* is the amount of time the partition leader spends processing a request, including sending it to disk (but not necessarily flushing it)
- *Remote Time* is the amount of time spent waiting for the followers before request processing can complete
- *Throttle Time* is the amount of time the response must be held in order to slow the requestor down to satisfy client quota settings
- *Response Queue Time* is the amount of time the response to the request spends in queue before it can be sent to the requestor
- *Response Send Time* is the amount of time spent actually sending the response

The attributes provided for each metric are:

- Percentiles - `50thPercentile`, `75thPercentile`, `95thPercentile`, `98thPercentile`, `99thPercentile`, `999thPercentile`
- `Count` - Absolute count of number of requests since process start
- `Min` - Minimum value for all requests
- `Max` - Maximum value for all requests
- `Mean` - Average value for all requests
- `StdDev` - The standard deviation of the request timing measurements as a whole

WHAT IS A PERCENTILE?

Percentiles are a common way of looking at timing measurement, even though many misunderstand what a percentile measures. A 99th percentile measurement tells us that 99% of all values in the sample group (request timings, in this case) are less than the value of the metric. This means that 1% of the values are greater than the value specified. A common pattern is to view the average value and the 99% or 99.9% value. In this way you can understand how the average request performs and what the outliers are.

Out of all of these metrics and attributes for requests, which are the important ones to monitor? At a minimum, you should collect at least the average and one of the higher percentiles (either 99% or 99.9%) for the *Total Time* metric, as well as the *Requests Per Second* metric, for every request type. This gives a view into the overall performance of requests to the Kafka broker. If you can, you should also collect those measurements for the other six timing metrics for each request type, as this will allow you to narrow down any performance problems to a specific phase of request processing.

For setting alert thresholds, the timing metrics can be difficult. The timing for a `Fetch` request, for example, can vary wildly depending on many factors, including settings on the client for how long it will wait for messages, how busy the particular topic being fetched is, and the speed of the network connection between the client and the broker. It can be very useful, however, to develop a baseline value for the 99.9th percentile measurement for at least the *Total Time*, especially for `Produce` requests, and alert on this. Much like the under-replicated partitions metric, a sharp increase in the 99.9th percentile for `Produce` requests can alert you to a wide range of performance problems.

Topic and Partition Metrics

In addition to the many metrics available on the broker that describe the operation of the Kafka broker in general, there are topic and partition specific metrics that are provided. In larger clusters these can be numerous, and it may not be possible to collect all of them into a metrics system as a matter of normal operations. However, they are quite useful for debugging specific issues with a client. For example, the topic metrics can be used to identify a specific topic that is causing a large increase in traffic to the cluster. It also may be important to provide these metrics so that users of Kafka (the producer and consumer clients) are able to access them. Regardless of whether or not you are able to collect these metrics regularly, you should be aware of what is useful.

> **EXAMPLE TOPIC NAMES**
>
> For all the examples below, we will be using the example topic name "TOPICNAME", as well as partition 0. When accessing the metrics described, make sure to substitute the topic name and partition number that are appropriate for your cluster.

PER-TOPIC METRICS

Table 10-2. Metrics for Each Topic

Name	JMX MBean
Bytes In Rate	`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=TOPICNAME`
Bytes Out Rate	`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=TOPICNAME`
Failed Fetch Rate	`kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=TOPICNAME`
Failed Produce Rate	`kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=TOPICNAME`
Messages In Rate	`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=TOPICNAME`
Fetch Request Rate	`kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=TOPICNAME`
Produce Request Rate	`kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=TOPICNAME`

For all the per-topic metrics, the measurements are very similar to the broker metrics described previously. In fact, the only difference is the provided topic name, and that the metrics will be specific to the named topic. Given the sheer number of metrics available, depending on the number of topics present in your cluster, these will almost certainly be metrics that you will not want to set up monitoring and alerts for. They are useful to provide to clients, however, so that they are able to evaluate and debug their own usage of Kafka.

PER-PARTITION METRICS

Table 10-3. Metrics for Each Topic

Name	JMX MBean
Partition Size	`kafka.log:type=Log,name=Size,topic=TOPICNAME,partition=0`
Log Segment Count	`kafka.log:type=Log,name=NumLogSegments,topic=TOPICNAME,partition=0`
Log End Offset	`kafka.log:type=Log,name=LogEndOffset,topic=TOPICNAME,partition=0`
Log Start Offset	`kafka.log:type=Log,name=LogStartOffset,topic=TOPICNAME,partition=0`

The per-partition metrics tend to be less useful on an ongoing basis than the per-topic metrics. Additionally, they are quite numerous as hundreds of topics can easily be thousands of partitions. Nevertheless, they can be useful in some limited situations. In particular, the partition size metric indicates the amount of data (in bytes) that is currently being retained on disk for the partition. Combined, these will indicate the amount of data retained for a single topic, which can be useful in allocating costs for Kafka to individual clients. A discrepancy between the size of two partitions for the same topic can indicate a problem where the messages are not evenly distributed across the key that is being used when producing. The log segment count metric shows the number of log segment files on disk for the partition. This may be useful along with the partition size for resource tracking.

The log end offset and log start offset metrics are the highest and lowest offsets for messages in that partition, respectively. It should be noted, however, that the difference between these two numbers does not necessarily indicate the number of messages in the partition, as log compaction can result in "missing" offsets that have been removed from the partition due to newer messages with the same key. In some environments it could be useful to track these offsets for a partition. One such use case was to provide a more granular mapping of timestamp to offset, allowing for consumer clients to easily roll back offsets to a specific time (though this is less important with time-based index searching, introduced in Kafka 0.10.1).

> **UNDER-REPLICATED PARTITION METRICS**
>
> There is a per-partition metric provided to indicate whether or not the partition is under-replicated. In general, this is not very useful in day-to-day operations, as there are too many metrics to gather and watch. It is much easier to monitor the broker-wide under-replicated partition count, and then use the command line tools (described in Chapter 9) to determine the specific partitions that are under-replicated.

Java Virtual Machine Monitoring

In addition to the metrics provided by the Kafka broker, you should be monitoring a standard suite of measurements for all of your servers, as well as the Java Virtual Machine (JVM) itself. These will be useful to alert you to a situation, such as increasing garbage collection activity, that will degrade the performance of the broker. They will also provide insight into why you see changes in metrics downstream in the broker.

GARBAGE COLLECTION

For the Java JVM, the critical thing to monitor is the status of garbage collection (GC). The particular beans that you must monitor for this information will vary depending on the particular Java Runtime Environment (JRE) that you are using, as well as the specific garbage collection settings in use. For an Oracle Java 1.8 JRE running with G1 garbage collection, the beans to use are:

Table 10-4. G1 Garbage Collection Metrics

Name	JMX MBean
Full GC Cycles	`java.lang:type=GarbageCollector,name=G1 Old Generation`
Young GC Cycles	`java.lang:type=GarbageCollector,name=G1 Young Generation`

Note that in the semantics of garbage collection, "Old" and "Full" are the same thing. For each of these metrics, the two attributes to watch are `CollectionCount` and `CollectionTime`. The `CollectionCount` is the number of garbage collection cycles of that type (full or young) since the JVM was started. The `CollectionTime` is the amount of time, in milliseconds, spent in that type of GC cycle since the JVM was started. As these measurements are counters, they can be used by a metrics system to tell you an absolute number of GC cycles and time spent in GC per unit of time. They can also be used to provide an average amount of time per GC cycle, though this is of less use in normal operations.

Each of these metrics also has a `LastGcInfo` attribute. This is a composite value, made up of 5 fields that give you information on the last GC cycle of that type. The important value to look at is the `duration` value, as this tells you how long, in milliseconds, the last GC cycle took. The other values in the composite (`GcThreadCount`, `id`, `startTime`, and `endTime`) are informational and do not prove to be very useful. It's important to note that you will not be able to see the timing of every GC cycle using this Attribute, as young GC cycles especially can happen frequently.

JAVA OS MONITORING

The JVM can provide you with some information on the OS through the `java.lang:type=OperatingSystem` bean. However, this information is limited and does not represent everything you need to know about the system running your broker. The two attributes that can be collected here that are of use, which are difficult to collect in the OS, are the `MaxFileDescriptorCount` and `OpenFileDescriptorCount` attributes. `MaxFileDescriptorCount` will tell you the maximum number of file descriptors (FDs) that the JVM is allowed to have open. The `OpenFileDescriptorCount` attribute tells you the number of FDs that are currently open. There will be FDs open for every log segment and network connection, and they can add up quickly. A problem closing network connections properly could cause the broker to rapidly exhaust the number allowed.

Operating System Monitoring

The JVM cannot provide us with all the information that we need to know about the system that it is running on. For this reason we must not only collect metrics from the broker, but also from the operating system (OS) itself. Most monitoring systems will provide agents that will collect more OS information than you could possibly be interested in. The main areas that are necessary to watch are CPU usage, memory usage, disk usage, disk IO, and network usage.

For CPU utilization, you will want to look at the system load average at the very least. This provides a single number that will indicate the relative utilization of the processors. In addition, it may also be useful to capture the percent usage of the CPU broken down by type. Depending on the method of collection, and your particular operating system, you may have some or all of the following CPU percentage breakdowns (provided with the abbreviation used):

- `us` - time spent in user space
- `sy` - time spent in kernel space
- `ni` - time spent on low priority processes
- `id` - time spent idle
- `wa` - time spent in wait (on disk)

- `hi` - time spent handling hardware interrupts
- `si` - time spent handling software interrupts
- `st` - time waiting for the hypervisor

> **WHAT IS SYSTEM LOAD?**
>
> While many know that system load is a measure of CPU usage on a system, most people misunderstand how it is measured. The load average is a count of the number of processes that are runnable and are waiting for a processor to execute on. Linux also includes threads that are in an uninterruptable sleep state, such as waiting for the disk. The load is presented as three numbers, which is the count averaged over the last minute, five minutes, and fifteen minutes. In a single CPU system, a value of one would mean the system is 100% loaded, with a thread always waiting to execute. This means that on a multiple CPU system, the load average number that indicates 100% is equal to the number of CPUs in the system.

Processor usage is important to keep track of for the Kafka broker, as it uses a significant amount of processing for handling requests. Memory is less important to track for the broker itself, as Kafka will normally be run with a relatively small JVM heap size. It will use a small amount of memory outside of the heap for compression functions, but most of the system memory will be left to be used for cache. All the same, you should keep track of memory utilization to make sure other applications do not infringe on the broker. You will also want to assure that swap memory is not being used by monitoring the amount of total and free swap memory.

Disk is by far the most important subsystem when it comes to Kafka. All messages are persisted to disk, so the performance of Kafka depends heavily on the performance of the disks. Monitoring usage of both disk space and inodes is important, as you need to assure that you are not running out of space. This is especially true for the partitions where Kafka data is being stored. It is also necessary to monitor the disk IO statistics, as this will tell us that the disk is being used efficiently. For at least the disks where Kafka data is stored, monitor the reads and writes per second, the average read and write queue sizes, the average wait time, and the percent utilization of the disk.

Finally, monitor the network utilization on the brokers. This is simply the amount of inbound and outbound network traffic, normally reported in bits per second. Keep in mind that every bit inbound to the Kafka broker will be a number of bits outbound equal to the replication factor of the topics, with no consumers. Depending on the number of consumers, inbound network traffic could easily become an order of magnitude larger on outbound traffic. Keep this in mind when setting thresholds for alerts.

Logging

No discussion of monitoring is complete without a word about logging. Like many applications, the Kafka broker will fill disks with log messages in minutes if you let it. In order to get useful information from logging, it is important to select the right loggers to enable, and the right levels to set them at. By simply logging all messages at the `INFO` level, you will capture a significant amount of important information about the state of the broker. It is useful to separate a couple loggers from this, however, in order to provide a cleaner set of log files.

There are two loggers to consider separating out to separate files. The first is `kafka.controller`, still at the `INFO` level. This logger is used to provide messages specifically regarding the cluster controller. At any time, only one broker will be the controller, and therefore only one broker will be writing to this logger. The information includes topic creation and modification, broker status changes, and cluster activities such as preferred replica elections and partition moves. The other logger to separate is `kafka.server.ClientQuotaManager`, also at the `INFO` level. This logger is used to show messages related to produce and consume quota activities. While this is useful information, it is better to not have it in the main broker log file.

There is also some logging that may be useful to turn on when debugging issues with Kafka. One such logger is `kafka.request.logger`, turned on at either `DEBUG` or `TRACE` levels. This logs information about every request sent to the broker. At `DEBUG` level, the log includes connection endpoints, request timings, and summary information. At the `TRACE` level, it will also

include topic and partition information - nearly all request information short of the message payload itself. At either level, this logger generates a significant amount of data, and it is not recommended to enable it unless necessary for debugging.

Other useful information to have is regarding the status of the log compaction threads. There is no single metric to show the health of these threads, and it is possible for failure in compaction of a single partition to halt the log compaction threads entirely, and silently. Enabling the `kafka.log.LogCleaner`, `kafka.log.Cleaner`, and `kafka.log.LogCleanerManager` loggers at the `DEBUG` level will output information about the status of these threads. This will include information about each partition being compacted, including the size and number of messages in each. Under normal operations, this is not a lot of logging. This means that it can be enabled by default without overwhelming you.

Client Monitoring

All applications need monitoring and those that instantiate a Kafka client, either a producer or consumer, have metrics specific to the client that should be captured. This section specifically covers the official Java client libraries, though other implementations should have their own measurements available.

Producer Metrics

The new Kafka producer client has greatly compacted the metrics available by making them available as attributes on a small number of mbeans. In contrast, the previous version of the producer client (which is no longer supported) used a larger number of mbeans, but had more detail in many of the metrics (providing a greater number of percentile measurements and different moving averages). As a result, the overall number of metrics provided covers a wider surface area, but it can be more difficult to track outliers.

All of the producer metrics have the client ID of the producer client in the bean names. In the examples provided, this has been replaced with `CLIENTID`. Where a bean name contains a broker ID, this has been replaced with `BROKERID`. Topic names have been replaced with `TOPICNAME`.

Table 10-5. Kafka Producer Metric MBeans

Name	JMX MBean
Overall Producer	`kafka.producer:type=producer-metrics,client-id=CLIENTID`
Per-Broker	`kafka.producer:type=producer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID`
Per-Topic	`kafka.producer:type=producer-topic-metrics,client-id=CLIENTID,topic=TOPICNAME`

Each of these metric beans have multiple attributes available to describe the state of the producer. The particular attributes that are of the most use are described below. Before proceeding, it is worthwhile to understand the semantics of how the producer works as described in Chapter 3.

OVERALL PRODUCER METRICS

The overall producer metrics bean provides attributes describing everything from the sizes of the message batches to the memory buffer utilization. While all of these measurements have their uses in debugging, there are only a handful that are needed on a regular basis, and only a couple of those that should be monitored and have alerts set for. Note that while we will discuss several metrics that are averages (ending in `-avg`), there are equivalent maximum values (ending in `-max`) that have limited usefulness.

The `record-error-rate` is one attribute that you will definitely want to set an alert for. This metric should always be zero, and if it is anything greater than that it means that the producer is dropping messages that it is trying to send to the Kafka brokers. The

producer has a configured number of retries and a backoff between those, and once that has been exhausted the messages (called records here) will be dropped. There is also a `record-retry-rate` attribute that can be tracked, but it is less critical than the error rate because retries are normal.

The other metric to alert on is the `request-latency-avg`. This is the average amount of time a produce request sent to the brokers takes. You should be able to establish a baseline value for what this number should be in normal operations, and set an alert threshold above that. An increase in the request latency means that produce requests are getting slower. This could be due to networking issues, or it could indicate problems on the brokers. Either way, it's a performance issue that will cause back-pressure and other problems in your producing application.

In addition to these critical metrics, it is always good to know how much message traffic your producer is sending. Three attributes will provide three different views of this. The `outgoing-byte-rate` describes the messages in a absolute size in bytes per second. The `record-send-rate` describes the traffic in terms of the number of messages produced per second. Finally, the `request-rate` provides the number of produce requests send to the brokers per second. A single request contains one or more batches. A single batch contains one or more messages. And, of course, each message is made up of some number of bytes. These metrics are all useful to have on an application dashboard to track how it uses Kafka.

WHY NOT PRODUCERREQUESTMETRICS?

There is a producer metric bean called `ProducerRequestMetrics` that provides both percentiles for request latency as well as several moving averages for the request rate. So why is it not one of the recommended metrics to use? The problem is that this metric is provided separately for each producer thread. In applications where there are multiple threads used for performance reasons it will be difficult to reconcile these metrics. It is normally sufficient to use the attributes provided by the single overall producer bean.

In order to help understand the relationship between bytes, records, requests, and batches, there are also metrics that describe the sizes of these entities. The `request-size-avg` metric provides the average size of the produce requests being sent to the brokers in bytes. The `batch-size-avg` provides the average size of a single message batch (which, by definition, is comprised of messages for a single topic partition) in bytes. The `record-size-avg` shows the average size of a single record in bytes. For a single-topic producer, this provides useful information about the messages being produced. For multiple-topic producers, such as Mirror Maker, it is less informative. Besides these three metrics, there is a `records-per-request-avg` metric which describes the average number of messages that are in a single produce request.

The last overall producer metric attribute that is recommended is `record-queue-time-avg`. This measurement is the average amount of time, in milliseconds, that a single message waits in the producer, after the application sends it, before it is actually produced to Kafka. After an application calls the producer client to send a message (by calling the `send` method), the producer waits until one of two things happens:

- The producer client has enough messages to fill a batch based on the `max.partition.bytes` configuration
- It has been long enough since the last batch was sent based on the `linger.ms` configuration

Either of these two will cause the producer client to close the current batch it is building and send it to the brokers. The easiest way to understand it is that for busy topics the first condition will apply, while for slow topics the second will apply. The `record-queue-time-avg` measurement will indicate how long messages take to be produced, and therefore is helpful when tuning these two configurations to meet the latency requirements that your application has.

PER-BROKER AND PER-TOPIC METRICS

In addition to the overall producer metrics, there are metric beans that provide a limited set of attributes for the connection to each Kafka broker, as well as for each topic that is being produced. These measurements are useful for debugging problems in some cases, but they are not metrics that you are going to want to review on an ongoing basis. All of the attributes on these beans are named as

the overall metrics attributes are named, and have the same meaning as described previously (excepting that they apply either to a specific broker or a specific topic).

For the metrics that are provided for each Kafka broker connection, the one that is more useful is the `request-latency-avg`. This is because this metric will be mostly stable (given stable batching of messages) and can still show a problem with connections to a specific broker. The other attributes, such as `outgoing-byte-rate` and `request-latency-avg`, will tend to vary depending on what partitions each broker is leading. This means that what these measurements "should" be at any point in time can quickly change, depending on the state of the Kafka cluster.

The topic metrics are a little more interesting than the per-broker metrics, but they will only be useful for producers that are working with more than one topic. They will also only be useable on a regular basis if the producer is not working with a lot of topics. For example, a mirror maker could be producing hundreds, or thousands, of topics. It is difficult to review all of those metrics, and nearly impossible to set reasonable alert thresholds on them. As with the per-broker metrics, the per-topic measurements are best used when investigating a specific problem. The `record-send-rate` and `record-error-rate` attributes, for example, can be used to isolate dropped messages to a specific topic (or validated to be across all topics). In addition, there is a `byte-rate` metric that provides the overall messages rate in bytes per second for the topic.

Consumer Metrics

Similar to the new producer client, the new consumer in Kafka consolidates many of the metrics into attributes on just a few metric beans. This has the same tradeoffs described for the producer client as well, where the percentiles for latencies and the moving averages for rates have been removed. In the consumer, because the logic around consuming messages is a little more complex than just firing messages into the Kafka brokers, there are a few more metrics to deal with as well.

Table 10-6. Kafka Consumer Metric MBeans

Name	JMX MBean
Overall Consumer	`kafka.consumer:type=consumer-metrics,client-id=CLIENTID`
Fetch Manager	`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID`
Per-Topic	`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID,topic=TOPICNAME`
Per-Broker	`kafka.consumer:type=consumer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID`
Coordinator	`kafka.consumer:type=consumer-coordinator-metrics,client-id=CLIENTID`

FETCH MANAGER METRICS

In the consumer client, the overall consumer metric bean is less useful for us because the metrics of interest are located in the fetch manager beans instead. The overall consumer bean has metrics regarding the lower-level network operations, but the fetch manager bean has the metrics regarding bytes, request, and record rates. Unlike the producer client, the metrics provided by the consumer are useful to look at but not useful for setting up alerts on.

For the fetch manager, the one attribute you may want to set up monitoring and alerts for is `fetch-latency-avg`. As with the equivalent `request-latency-avg` in the producer client, this metric tells us how long fetch requests to the brokers take. The problem with alerting on this metric is that the latency is governed by the consumer configurations `fetch.min.bytes` and

`fetch.max.wait.ms`. A slow topic will have erratic latencies, as sometimes the broker will respond quickly (when there are messages available), and sometimes it will not respond for `fetch.max.wait.ms` (when there are no messages available). When consuming topics that have more regular, and abundant, message traffic, this metric may be more useful to look at.

> **WAIT! NO LAG?**
>
> The best advice for all consumers is that you must monitor the consumer lag. So why do we not recommend the `records-lag-max` attribute on the fetch manager bean for monitoring? This metric shows the current lag (number of messages behind the broker) for the partition that is the most behind.
>
> The problem with this is twofold: it only shows the lag for one partition, and it relies on proper functioning of the consumer. If you have no other option, use this attribute for lag and set up alerting for it. But the best practice is to use external lag monitoring, as will be described later in this chapter.

In order to know how much message traffic your consumer client is handling, you should capture the `bytes-consumed-rate` or the `records-consumed-rate`, or preferably both. These metrics describe the message traffic consumed by this client instance in bytes per second and messages per second, respectively. Some users set minimum thresholds on these metrics for alerting, so that they are notified if the consumer is not doing enough work. You should be careful when doing this, however, as part of Kafka is that consumers and producers are decoupled. This means that the consumer should be wary of making assumptions about the producer patterns.

It is also good to understand the relationship between bytes, messages, and requests, and the fetch manager provides metrics to help with this. The `fetch-rate` measurement tells us the number of fetch requests per second that the consumer is performing. The `fetch-size-avg` metric gives the average size of those fetch requests in bytes. Finally, the `records-per-request-avg` metric gives us the average number of messages in each fetch request. Note that the consumer does not provide an equivalent to the producer `record-size-avg` metric to let us know what the average size of a message is. If this is important, you will need to infer it from the other metrics available, or capture it in your application after receiving messages from the consumer client library.

PER-BROKER AND PER-TOPIC METRICS

The metrics that are provided by the consumer client for each of the broker connections and each of the topics being consumed, as with the producer client, are useful for debugging issues with consumption, but will probably not be measurements that you review daily. As with the fetch manager, the `request-latency-avg` attribute provided by the per-broker metrics bean has limited usefulness, depending on the message traffic in the topics you are consuming. The `incoming-byte-rate` and `request-rate` metrics break down the consumed message metrics provided by the fetch manager into per-broker bytes per second and requests per second measurements, respectively. These can be used to help isolate problems that the consumer is having to the connection to a specific broker.

Per-topic metrics provided by the consumer client are useful if more than one topic is being consumed. Otherwise, these metrics will be the same as the fetch manager's metrics and redundant to collect. On the other end of the spectrum, if the client is consuming many topics, such as a mirror maker may be, these metrics will be difficult to review. If you plan on collecting them, the most important metrics to gather are the `bytes-consumed-rate`, the `records-consumed-rate`, and the `fetch-size-avg`. The `bytes-consumed-rate` shows the absolute size in bytes consumed per second for the specific topic, while the `records-consumed-rate` shows the same information in terms of the number of messages. The `fetch-size-avg` provides the average size of each fetch request for the topic in bytes.

CONSUMER COORDINATOR METRICS

As described in Chapter 4, consumer clients generally work together as part of a consumer group. This group has coordination activities, such as group members joining and heartbeat messages to the brokers to maintain group membership. The consumer coordinator is the part of the consumer client that is responsible for handling this work, and it maintains its own set of metrics. As with all metrics, there are many numbers provided, but only a few key ones that you should monitor regularly.

The biggest problem that consumers can run into due to coordinator activities is a pause in consumption while the consumer group synchronizes. This is when the consumer instances in a group negotiate which partitions which be consumed by which individual client instances. Depending on the number partitions that are being consumed, this can take some time. The coordinator provides the metric attribute `sync-time-avg`, which is the average amount of time, in milliseconds, that the sync activity takes. It is also useful to capture the `sync-rate` attribute, which is the number of group syncs that happen every second. For a stable consumer group, this number should be zero most of the time.

The consumer needs to commit offsets to checkpoint it's progress in consuming messages, either automatically on a regular interval, or by manual checkpoints triggered in the application code. These commits are essentially just produce requests (though they have their own request type), in that the offset commit is a message produced to a special topic. The consumer coordinator provides the `commit-latency-avg` attribute, which measures the average amount of time that offset commits take. You should monitor this value just as you would the request latency in the producer. It should be possible to establish a baseline expected value for this metric, and set reasonable thresholds for alerting above that value.

One final coordinator metric that can be useful to collect is `assigned-partitions`. This is a count of the number of partitions that the consumer client (as a single instance in the consumer group) has been assigned to consume. This is helpful because, when compared to this metric from other consumer clients in the group, it is possible to see the balance of load across the entire consumer group. We can use this to identify imbalances that might be caused by problems in the algorithm used by the consumer coordinator for distributing partitions to group members.

Quotas

Apache Kafka has the ability to throttle client requests in order to prevent one client from overwhelming the entire cluster. This is configurable for both producer and consumer clients, and is expressed in terms of the permitted amount of traffic from an individual client ID to an individual broker in bytes per second. There is a broker-level default value, as well as per-client overrides. When the broker calculates that a client has exceeded its quota, it slows the client down by holding the response back to the client for enough time to keep the client under the quota.

The Kafka broker does not use error codes in the response to indicate that the client is being throttled. This means that it is not obvious to the application that throttling is happening without monitoring the metrics that are provided to show the amount of time that the client is being throttled. The metrics that must be monitored are:

- Consumer: bean `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID`, attribute `fetch-throttle-time-avg`
- Producer: bean `kafka.producer:type=producer-metrics,client-id=CLIENTID`, attribute `produce-throttle-time-avg`

Quotas are not enabled by default on the Kafka brokers, but it is safe to monitor these metrics irrespective of whether or not you are currently using quotas. Monitoring them is a good practice as they may be enabled at some point in the future, and it's easier to start with monitoring them as opposed to adding them later.

Lag Monitoring

For Kafka consumers, the most important thing to monitor is the consumer lag. Measured in number of messages, this is the difference between the last message produced in a specific partition and the last message processed by the consumer. While this topic would normally be covered in the previous section on consumer client monitoring, it is one of the cases where external monitoring far surpasses what is available from the client itself. As mentioned previously, there is a lag metric in the consumer client, but using it is problematic. It only represents a single partition, the one partition that has the most lag, so it does not accurately show how far behind the consumer is. In addition, it requires proper operation of the consumer, because the metric is calculated by the consumer on each fetch request. If the consumer is broken or offline, the metric is either inaccurate or not available.

The preferred method of consumer lag monitoring is to have an external process which is able to watch both the state of the partition on the broker, tracking the offset of the most recently produced message, and the state of the consumer, tracking the last offset the consumer group has committed for the partition. This provides an objective view that can be updated regardless of the status of the

consumer itself. This checking must be performed for every partition that the consumer group consumes. For a large consumer, like mirror maker, this may mean tens of thousands of partitions.

The Chapter 9 chapter provides information on using the command line utilities to get consumer group information, including committed offsets and lag. Monitoring lag like this, however, presents its own problems. First, you must understand for each partition what is a reasonable amount of lag. A topic that receives 100 messages an hour will need a different threshold from a topic that receives 100,000 messages per second. Then, you must be able to consume all of the lag metrics into a monitoring system and set alerts on them. If you have a consumer group that consumes 100,000 partitions over 1,500 topics, you may find this to be a daunting task.

One way to work with this is to use Burrow. This is an open source application, originally developed by LinkedIn, which provides consumer status monitoring by gathering this information for all consumer groups in a cluster and calculating a single status for each group saying whether the consumer group is working properly, falling behind, or is stalled or stopped entirely. It does this without requiring thresholds by monitoring the progress that the consumer group is making on processing messages, though you can also get the message lag as an absolute number. There is an in-depth discussion of the reasoning and methodology behind how Burrow works on the LinkedIn Engineering Blog. Deploying Burrow can be an easy way to provide monitoring for all consumers in a cluster, as well as in multiple clusters, and it can be easily integrated with your existing monitoring and alerting system.

If there is no other option, the `records-lag-max` metric from the consumer client will provide at least a partial view of the consumer status. It is strongly suggested, however, that you utilize an external monitoring system like Burrow.

End to End Monitoring

Another type of external monitoring that is recommended for knowing whether or not your Kafka clusters are working properly is an end-to-end monitoring system that provides a client point of view on the health of the Kafka cluster. Consumer and producer clients have metrics that can indicate that there might be a problem with the Kafka cluster, but this can be a guessing game as to whether increased latency is due to a problem with the client, the network, or Kafka itself. In addition, it means that if your job is running the Kafka cluster, and not the clients, you would now have to monitor all of the clients as well. What is really needed is a simple answer to two questions:

- Can I produce messages to the Kafka cluster?
- Can I consume messages from the Kafka cluster?

Ideally you would want to know this for each individual topic, but it is often not reasonable to inject synthetic traffic for every topic in order to do that. We can, however, at least provide those answers for every broker in the cluster, and that is what Kafka Monitor does. This tool, open sourced by the Kafka team at LinkedIn, continually produces and consumes data from a topic that is spread across all brokers in a cluster. It measures the availability of both produce and consume requests on each broker, as well as the total produce to consume latency. This type of monitoring is invaluable to be able to externally verify that the Kafka cluster is operating as intended, as just like consumer lag monitoring, the Kafka broker cannot always be trusted to properly say when it is broken.

Summary

Monitoring is a key aspect of running Apache Kafka properly, which explains why so many teams spend a significant amount of their time perfecting that part of operations. Many organizations use Kafka to handle petabyte-scale data flows. Assuring that the data does not stop, and that messages are not lost, is a critical business requirement. Our goal as operators of a Kafka cluster is to be the most knowledgable people about the state of the cluster. At the same time, we need to assist our users with properly monitoring their own applications that use the cluster.

In this chapter we've covered the basics of how to monitor Java applications, and specifically the Kafka applications. We reviewed a subset of the numerous metrics available in the Kafka broker, also touching on Java and operating system monitoring, as well as logging. We then detailed the monitoring available in the Kafka client libraries, including quota monitoring. Finally, we discussed the use of external monitoring systems for consumer lag monitoring and end to end cluster availability. While certainly not an exhaustive list of the metrics that are available, this chapter has reviewed the most critical ones to keep an eye on.

PREV
9. Administering Kafka

NEXT
11. Stream Processing