6 Brokers

This chapters covers

- The role of brokers and their duties
- Evaluating options for certain broker configuration values
- Explaining replicas and how they stay up to date

So far in our discussions, we have dealt with Kafka from the view of an application developer interacting from external applications and processes. However, Kafka is a distributed system that deserves attention in its own right. In this chapter, let's look at the parts that make the Kafka brokers work.

6.1 Introducing the broker

Although we have focused on the client side of Kafka so far, our focus will now shift to another powerful component of the ecosystem: brokers. Brokers work together with other brokers to form the core of the system.

As we start to discover Kafka, those who are familiar with big data concepts or who have worked with Hadoop before might see familiar terminologies such as *rack awareness* (knowing which physical server rack a machine is hosted on) and *partitions*. Kafka has a rack awareness feature that makes replicas for a partition exist physically on separate racks [1]. Using familiar data terms should make us feel at home as we draw new parallels between what we've worked with before and what Kafka can do for us. When setting up our own Kafka cluster, it is important to know that we have another cluster to be aware of: Apache ZooKeeper. This then is where we'll begin.

6.2 Role of ZooKeeper

ZooKeeper is a key part of how the brokers work and is a requirement to run Kafka. Because Kafka needs to be running and exist before the brokers do, we will start our discussion there.

> **NOTE** As mentioned in chapter 2, to simplify the requirements of running Kafka, there was a proposal for the replacement of ZooKeeper with its own managed quorum [2]. Because this work was not yet complete at the time of publication, ZooKeeper is discussed in this work. But look for an early access release of the managed quorum, arriving in version 2.8.0.

As ZooKeeper needs to have a minimum number in order to elect leaders and reach a decision, this cluster is indeed important for our brokers [3]. ZooKeeper itself holds information such as topics in our cluster [4]. ZooKeeper helps the brokers by coordinating assignments and notifications [5].

With all of this interaction with the brokers, it is important that we have ZooKeeper running before starting our brokers. The health of the ZooKeeper cluster impacts the health of our Kafka brokers. For instance, if our ZooKeeper instances are damaged, topic metadata and configuration could be lost.

Usually, we won't need to expose the details (IP addresses and ports) of our ZooKeeper cluster to our producer and consumer applications. Certain legacy frameworks we use might also provide a means of connecting our client application with our ZooKeeper cluster. One example of this is version 3.1.*x* of Spring Cloud Stream, which allowed us to set the `zkNodes` property [6]. The value defaulted to `localhost` and should be left alone in most cases to avoid a ZooKeeper dependency. The `zkNodes` property is marked as deprecated, but you never know if you will encounter older code for maintenance, so you want to keep an eye out for it. Why is this not needed currently and in the future? Besides the fact that Kafka will not always require ZooKeeper, it is also important for us to avoid unnecessary external dependencies in our applications. In addition, it gives us fewer ports to expose if we are working with firewalls for Kafka and our client to communicate directly.

Using the Kafka tool `zookeeper-shell.sh`, which is located in the bin folder of our Kafka installation, we can connect to a ZooKeeper host in our cluster and look at how the data is stored [7]. One way to find the paths that Kafka uses is to look at the

class ZkData.scala [8]. In this file, you will find paths like /controller, /controller_epoch, /config, and /brokers, for example. If we look at the /brokers/topics path, we will see a list of the topics that we have created. At this point, we should, hopefully, at least have the kinaction_helloworld topic in the list.

> **NOTE** We can also use a different Kafka tool, kafka-topics.sh, to see the list of topics, getting the same results! Commands in the following listings connect to ZooKeeper and Kafka, respectively, for their data but do so with a different command interface. The output should include the topic we created in chapter 2, [kinaction_helloworld].

Listing 6.1 Listing our topics

```
bin/zookeeper-shell.sh localhost:2181
ls /brokers/topics

# OR
bin/kafka-topics.sh --list \
➥ --bootstrap-server localhost:9094
```

Connects to our local ZooKeeper instance

Lists all the topics with the ls command

Using kafka-topics, connects to ZooKeeper and lists the topics

Even when ZooKeeper no longer helps to power Kafka, we might need to work with clusters that have not migrated yet, and we will likely see ZooKeeper in documentation and reference material for quite a while. Overall, being aware of the tasks that Kafka used to rely on ZooKeeper to perform and the shift to handling those inside a Kafka cluster with internal metadata nodes provides insight into the moving pieces of the entire system.

Being a Kafka broker means being able to coordinate with the other brokers as well as talking to ZooKeeper. In testing or working with proof-of-concept clusters, we might have only one broker node. However, in production, we will almost always have multiple brokers.

Turning away from ZooKeeper for now, figure 6.1 shows how brokers exist in a cluster and how they are home to Kafka's data logs. Clients will be writing to and reading from brokers to get information into and out of Kafka, and they will demand broker attention [9].

6.3 Options at the broker level

Configuration is an important part of working with Kafka clients, topics, and brokers. If you looked at the setup steps to create our first brokers in appendix A, we modified the server.properties file there, which we then passed as a command line argument to the broker startup shell script. This file is a common way to pass a specific configuration to a broker instance. For example, the log.dirs configuration property in that file should always be set to a log location that makes sense for your setup.

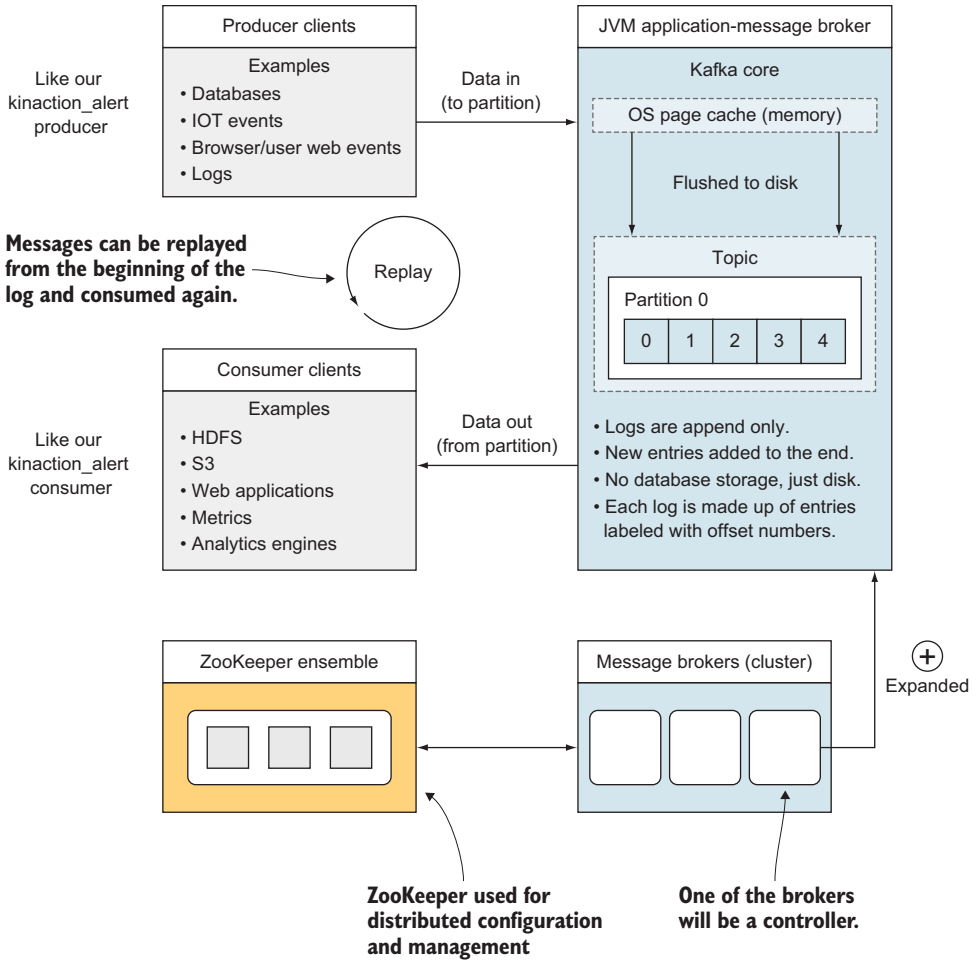

Figure 6.1 Brokers

This file also deals with configurations related to listeners, log locations, log retention, ZooKeeper, and group coordinator settings [10]. As with the producer and consumer configurations, look for the Importance label of "high" in the documentation at http://mng.bz/p9p2.

The following listing provides an example of what happens when we have only one copy of our data and the broker it is on goes down. This can happen when we allow the broker defaults and do not pick them with purpose. To begin, make sure that your local test Kafka cluster is running with three nodes, and create a topic like listing 6.2 presents.

Listing 6.2 Listing our topics

```
bin/kafka-topics.sh --create \
  --bootstrap-server localhost:9094 \
  --topic kinaction_one_replica

bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \
  --topic kinaction_one_replica

Topic: one-replica  PartitionCount: 1  ReplicationFactor: 1    Configs:
    Topic: kinaction_one_replica  Partition: 0
Leader: 2  Replicas: 2  Isr: 2
```

Creates a topic with only one partition and one replica

Describes the kinaction_one_replica topic with all the data located on the broker with ID 2

When we run the commands in listing 6.2 to create and describe the topic `kinaction_one_replica`, we'll see that there is only one value in the fields `Partition`, `Leader`, `Replicas`, and `Isr` (in-sync replicas). Further, the broker uses the same ID value. This means that the entire topic depends on that one broker being up and working.

If we terminate the broker with ID 2 in this example and then try to consume a message for that topic, we would get a message such as "1 partitions have leader brokers without a matching listener." Because there are no replica copies for the topic's partition, there is no easy way to keep producing or consuming that topic without recovering that broker. Although this is just one example, it illustrates the importance that broker configuration can have when users create their topics manually as in listing 6.2.

Another important configuration property to define sets the location for our application logs and errors during normal operation. Let's look at this next.

6.3.1 Kafka's other logs: Application logs

As with most applications, Kafka provides logs for letting us know what is going on inside the application. In the discussion that follows, the term *application logs* refers to the logs that we usually think of when working with any application, whether debugging or auditing. These application logs are not related to the record logs that form the backbone of Kafka's feature set.

The location where these application logs are stored is also entirely different than those for records. When we start a broker, we will find the application log directory in the Kafka base installation directory under the folder logs/. We can change this location by editing the config/log4j.properties file and the value for `kafka.logs.dir` [11].

6.3.2 Server log

Many errors and unexpected behaviors can be traced back to configuration issues on startup. The server log file, server.log, is where we would look if there is a startup error or an exception that terminates the broker. It seems to be the most natural place to

check first for any issues. Look (or use the `grep` command) for the heading `Kafka-Config values`.

If you are overwhelmed when you first look at the directory that holds this file, note that you will likely see other files like controller.log (if the broker was ever in that role) and older dated files with the same name. One tool that you can use for log rotation and compression is `logrotate` (https://linux.die.net/man/8/logrotate), but there are many other tools available as well to manage older server logs.

Something else to mention in regard to these logs is that they are located on each broker. They are not aggregated by default into one location. Various platforms might do this on our behalf, or we can gather them with a tool like Splunk™ (https://www.splunk.com/). It is especially important to know when we are trying to analyze logs to gather them when using something like a cloud environment in which the broker instance might not exist.

6.3.3 Managing state

As we discussed in chapter 2, each partition has a single leader replica. A leader replica resides on a single broker at any given time. A broker can host the leader replica of multiple partitions, and any broker in a cluster can host leader replicas. Only one broker in the cluster, however, acts as the controller. The role of the controller is to handle cluster management [12]. The controller also performs other administrative actions like partition reassignment [13].

When we consider a rolling upgrade of a cluster, shutting down and restarting one broker at a time, it is best to do the controller last [14]. Otherwise, we might end up restarting the controller multiple times.

To figure out which broker is the current controller, we can use the zookeeper-shell script to look up the ID of the broker, as listing 6.3 shows. The path /controller exists in ZooKeeper, and in the listing, we run one command to look at the current value. Running that command for my cluster showed my broker with ID 0 as the controller.

Listing 6.3 Listing the current controller

```
bin/zookeeper-shell.sh localhost:2181
get /controller
```

Connects to your ZooKeeper instance

Uses get against the controller path

Figure 6.2 shows all of the output from ZooKeeper, including the `brokerid` value, `"brokerid":0`. If we migrate or upgrade this cluster, we would upgrade this broker last due to this role.

We will also find a controller log file with the name controller.log that serves as an application log on broker 0 in this case. This log file can be important when we look at broker actions and failures.

```
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
get /controller
{"version":1,"brokerid":0,"timestamp":"1540874053577"}
cZxid = 0x2f
ctime = Mon Oct 29 23:34:13 CDT 2018
mZxid = 0x2f
mtime = Mon Oct 29 23:34:13 CDT 2018
pZxid = 0x2f
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x166c33ffa650000
dataLength = 54
numChildren = 0
```

Figure 6.2 Example controller output

6.4 Partition replica leaders and their role

As a quick refresher, topics are made up of partitions, and partitions can have replicas for fault tolerance. Also, partitions are written on the disks of the Kafka brokers. One of the replicas of the partition will have the job of being the leader. The leader is in charge of handling writes from external producer clients for that partition. Because the leader is the only one with newly written data, it also has the job of being the source of data for the replica followers [15]. And because the ISR list is maintained by the leader, it knows which replicas are up to date and have seen all the current messages. Replicas act as consumers of the leader partition and will fetch the messages [15].

Figure 6.3 shows a three-node cluster with broker 3 as its leader and broker 2 and broker 1 as its followers, using `kinaction_helloworld` as a topic that might have been

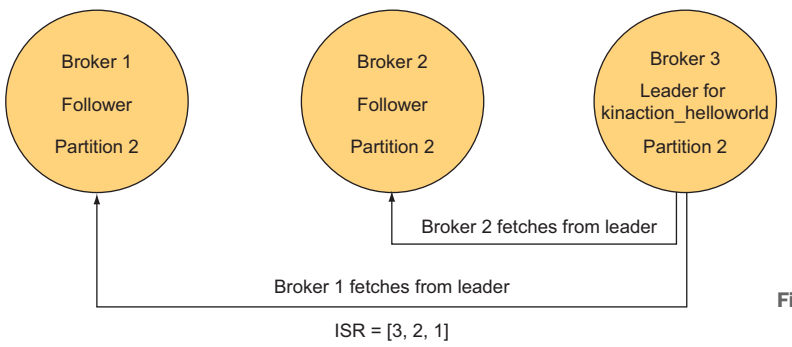

Figure 6.3 Leader

created in this manner. Broker 3 holds the leader replica for partition 2. As the leader, broker 3 handles all of the reads and writes from external producers and consumers. It also handles requests it receives from broker 2 and broker 1 as they pull new messages into their copies. The ISR list `[3,2,1]` includes the leader in the first position (`3`) and then the remaining followers (`2,1`), who stay current with their copies of the messages from the leader.

In some cases, a broker that fails may have hosted the leader replica for a partition. In figure 6.4, the previous example in figure 6.3 experiences a failure. Because broker 3 is not available, a new leader is elected. Figure 6.4 shows the new leader broker 2. Once a follower, it was elected as a leader replica to keep Kafka serving and receiving data for that partition. The ISR list is now `[2,1]` with the first position reflecting the new leader replica hosted on broker 2.

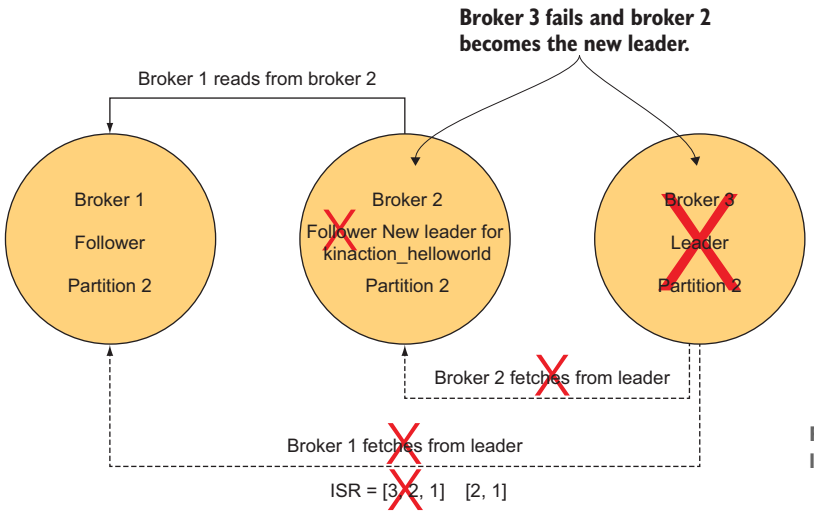

Figure 6.4 New leader elected

NOTE In chapter 5 we discussed a Kafka Improvement Proposal, KIP-392, which allows consumer clients to fetch from the closest replica [16]. Reading from a preferred follower rather than the leader replica is something that might make sense if our brokers span physical data centers. However, when discussing leaders and followers in this book, unless stated otherwise, we will focus on the default leader read and write behaviors.

In-sync replicas (ISRs) are a key piece to really understanding Kafka. For a new topic, a specific number of replicas are created and added to the initial ISR list [17]. This number can be either from a parameter or, as a default, from the broker configuration.

One of the details to note with Kafka is that replicas do not heal themselves by default. If you lose a broker on which one of your copies of a partition exists, Kafka does not (currently) create a new copy. We mention this because some users are used

to filesystems like HDFS that maintain their replication number (self-heal) if a block is seen as corrupted or failed. An important item to look at when monitoring the health of our systems is how many of our ISRs are indeed matching our desired number.

Why is watching this number so important? It is good to keep aware of how many copies you have before it hits 0! Let's say that we have a topic that is only one partition and that partition is replicated three times. In the best-case scenario, we would have two copies of the data that is in our lead partition replica. This, of course, means that the follower replicas are caught up with the leader. But what if we lose another ISR?

It is also important to note that if a replica starts to get too far behind in copying messages from the leader, it can be removed from the ISR list. The leader notices if a follower is taking too long and drops it from its list of followers [17]. Then the leader continues to operate with a new ISR list. The result of this "slowness" to the ISR list is the same as in figure 6.4, in which a broker failed.

6.4.1 Losing data

What if we have no ISRs and lose our lead replica due to a failure? When `unclean.leader.election.enable` is `true`, the controller selects a leader for a partition even if it is not up to date so that the system keeps running [15]. The problem with this is that data could be lost because none of the replicas have all the data at the time of the leader's failure.

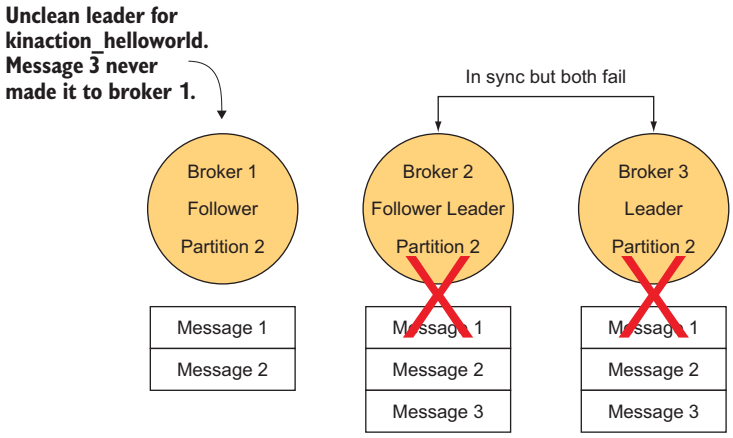

Figure 6.5 Unclean leader election

Figure 6.5 shows data loss in the case of a partition with three replicas. In this case, both brokers 3 and 2 failed and are not online. Because unclean leader election was enabled, broker 1 is made the new leader even though it is not in sync with the other brokers. Broker 1 never sees message 3, so it cannot present that data to clients. At the cost of missing data, this option allows us to keep serving clients.

6.5 Peeking into Kafka

There are many tools we can use to capture and view data from our applications. We will look at Grafana® (https://grafana.com/) and Prometheus® (https://prometheus.io/) as examples of tools that can be used to help set up a simple monitoring stack that can be used for Confluent Cloud [18].[1] We'll use Prometheus to extract and store Kafka's metrics data. Then we'll send that data to Grafana to produce helpful graphical views. To fully understand why we are setting up all of the following tools, let's quickly review the components and the work each one does (figure 6.6).

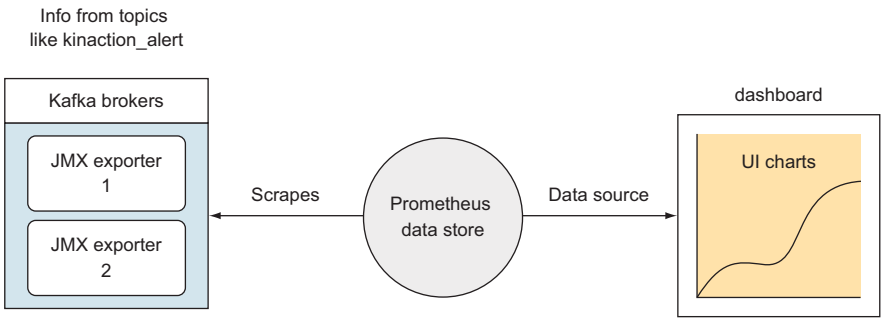

Figure 6.6 Graph flow

In figure 6.6, we use JMX to look inside the Kafka applications. The Kafka exporter takes the JMX notifications and exports them into the Prometheus format. Prometheus scrapes the exporter data and stores the metrics data. Various tools can then take the information from Prometheus and display that information in a visual dashboard.

There are many Docker™ images and Docker Compose files that bundle all of these tools, or you can install each tool to a local machine in order to explore this process in greater detail.

For the Kafka exporter, an outstanding option is available at https://github.com/danielqsj/kafka_exporter. We prefer the simplicity of this tool because we can just run it and give it one or a list of Kafka servers to watch. It might work well for your use cases as well. Notice that we will get many client and broker-specific metrics because there are quite a few options that we might want to monitor. Even so, this is not a complete list of the metrics available to us.

Figure 6.7 shows a query against a local data store, such as a local instance of Prometheus, that gathers metrics from our Kafka exporter tool. As we discussed about partitions, Kafka replicas do not heal themselves automatically, so one of the things we

[1] The Grafana Labs Marks are trademarks of Grafana Labs, and are used with Grafana Labs' permission. We are not affiliated with, endorsed or sponsored by Grafana Labs or its affiliates.

want to monitor is under-replicated partitions. If this number is greater than 0, we might want to look at what is going on in the cluster to determine why there is a replica issue. We might display the data from this query in a chart or dashboard, or we can, potentially, send an alert.

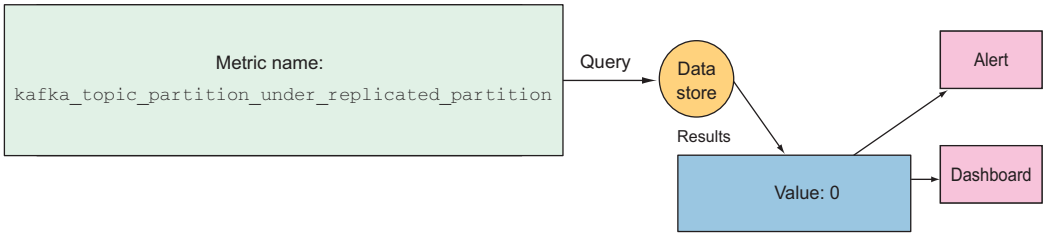

Figure 6.7 Metric query example

As noted, the Kafka exporter does not expose every JMX metric. To get more JMX metrics, we can set the `JMX_PORT` environment variable when starting our Kafka processes [19]. Other tools are available that use a Java agent to produce the metrics to an endpoint or port, which Prometheus can scrape.

Listing 6.4 shows how we would set the variable `JMX_PORT` when starting a broker [19]. If we already have a broker running and do not have this port exposed, we will need to restart the broker to affect this change. We may also want to automate the setting of this variable to ensure that it is enabled on all future broker restarts.

Listing 6.4 Starting a broker with a JMX port

```
JMX_PORT=$JMX_PORT bin/kafka-server-start.sh \
➥ config/server0.properties
```

Adds the JMX_PORT variable when starting the cluster

6.5.1 Cluster maintenance

As we consider moving to production, we will want to configure more than one server. Another item to note is that various pieces of the ecosystem such as Kafka and Connect clients, Schema Registry, and the REST Proxy do not usually run on the same servers as the brokers themselves. Although we might run all of these on a laptop for testing (and we can run this software on one server), for safety and efficiency, we definitely don't want all of these processes running on a single server when we handle production workloads. To draw a parallel to similarities with tools from the Hadoop ecosystem, Kafka scales well horizontally with more servers. Let's look at adding a server to a cluster.

6.5.2 Adding a broker

Beginning with a small cluster is a great way to start, as we can always add brokers to grow our footprint. To add a Kafka broker to our cluster, we just start a new Kafka broker with a unique ID. This ID can either be created with the configuration `broker.id` or with `broker.id.generation.enable` set to `true` [10]. That is pretty much it. But, there is something to be aware of in this situation—the new broker will not be assigned to any partitions! Any topic partitions that we create before adding a new broker still persist on the brokers that existed at the time of their creation [20]. If we are okay with the new broker only handling new topics, then we don't need to do anything else.

6.5.3 Upgrading your cluster

As with all software, updates and upgrades are a part of life. Not all systems can be brought down simultaneously and upgraded due to production workloads or business impact. One technique that can be used to avoid downtime for our Kafka applications is the *rolling restart* [14]. This means just upgrading one broker at a time. Figure 6.8 shows each broker being upgraded one at a time before moving on to the next broker for our cluster.

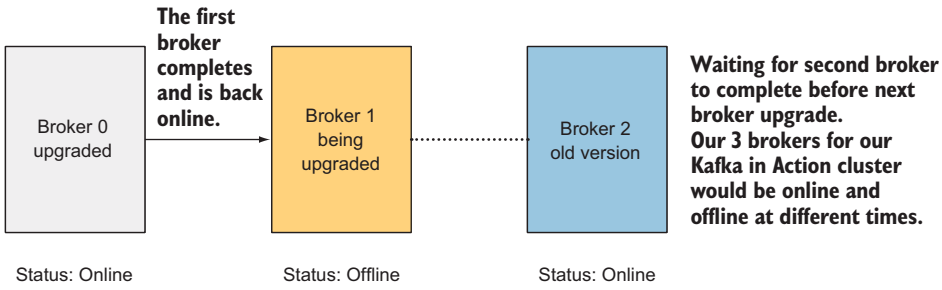

Figure 6.8 Rolling restart

An important broker configuration property for rolling restarts is `controlled.shutdown.enable`. Setting this to `true` enables the transfer of partition leadership before a broker shuts down [21].

6.5.4 Upgrading your clients

As mentioned in chapter 4, although Kafka does its best to decouple the clients from the broker, it's beneficial to know the versions of clients with respect to brokers. This bidirectional client compatibility feature was new in Kafka 0.10.2, and brokers version 0.10.0 or later support this feature [22]. Clients can usually be upgraded *after* all of the Kafka brokers in a cluster are upgraded. As with any upgrade, though, take a peek at the version notes to make sure newer versions are compatible.

6.5.5 Backups

Kafka does not have a backup strategy like one would use for a database; we don't take a snapshot or disk backup per se. Because Kafka logs exist on disk, why not just copy the entire partition directories? Although nothing is stopping us from doing that, one concern is making a copy of all of the data directories across all locations. Rather than performing manual copies and coordinating across brokers, one preferred option is for a cluster to be backed by a second cluster [23]. Between the two clusters, events are then replicated between topics. One of the earliest tools that you might have seen in production settings is MirrorMaker. A newer version of this tool (called MirrorMaker 2.0) was released with Kafka version 2.4.0 [24]. In the bin subdirectory of the Kafka install directory, we will find a shell script named kafka-mirror-maker as well as a new MirrorMaker 2.0 script, connect-mirror-maker.

There are also some other open source as well as enterprise offerings for mirroring data between clusters. Confluent Replicator (http://mng.bz/Yw7K) and Cluster Linking (http://mng.bz/OQZo) are also options to be aware of [25].

6.6 A note on stateful systems

Kafka is an application that definitely works with stateful data stores. In this book, we will work on our own nodes and not with any cloud deployments. There are some great resources, including Confluent's site on using the Kubernetes Confluent Operator API (https://www.confluent.io/confluent-operator/) as well as Docker images available to do what you need done. Another interesting option is Strimzi™ (https://github.com/strimzi/strimzi-kafka-operator), if you are looking at running your cluster on Kubernetes. At the time of this writing, Strimzi is a Cloud Native Computing Foundation® (https://www.cncf.io/) sandbox project. If you are familiar with these tools, it might be a quick way for you to kick the tires on a proof of concept (PoC) setup if you find some interesting projects out in the Docker Hub. There is not, however, a one-size-fits-all mandate for our infrastructure.

One benefit of Kubernetes that stands out is its ability to create new clusters quickly and with different storage and service communication options that Gwen Shapira explores further in her paper, "Recommendations for Deploying Apache Kafka on Kubernetes" [26]. For some companies, giving each product its own cluster might be easier to manage than having one huge cluster for the entire enterprise. The ability to spin up a cluster quickly rather than adding physical servers can provide the quick turnaround products need.

Figure 6.9 shows a general outline of how Kafka brokers can be set up in Kubernetes with an operator pod, similar to how the Confluent and Strimzi operators might work. The terms in the figure are Kubernetes-specific, and we do not provide much explanation here because we do not want to shift the focus away from learning about Kafka itself. We, rather, provide a general overview. Note that this is how a cluster *could* work, not a specific setup description.

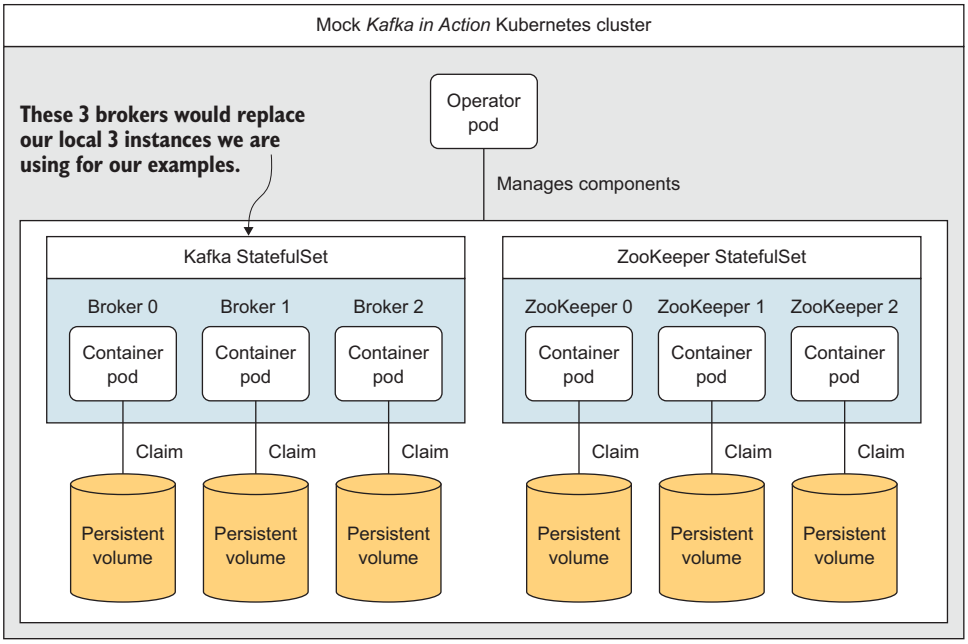

Figure 6.9 Kafka on Kubernetes

The Kubernetes operator is its own pod that lives inside of the Kubernetes cluster. As well, each broker is in its own pod as a part of a logical group called a StatefulSet. The purpose of the StatefulSet is to manage the Kafka pods and help guarantee ordering and an identity for each pod. If the pod that hosts a broker (the JVM process) with ID 0 fails, for example, a new pod is created with that identity (and not a random ID) and attaches to the same persistent storage volume as before. Because these volumes hold the messages of the Kafka partitions, the data is maintained. This statefulness helps overcome the sometimes short lives of containers. Each ZooKeeper node would also be in its own pod and part of its own StatefulSet.

For those who are new to Kubernetes or are anxious about the transition to such a platform, one migration strategy that can be helpful is to run Kafka clients and applications on a Kubernetes cluster before the Kafka brokers. Besides being stateless, running our clients in this manner can help us get a feel for Kubernetes at the start of our learning path. However, we should not neglect the need to understand Kubernetes well in order to run Kafka on top of this platform.

One developer team of four that one of the authors worked with recently focused half of the team on Kubernetes and half on running Kafka. Of course, this ratio might not be what every team encounters. The developer time required to focus on Kubernetes depends on your team and overall experience.

6.7 Exercise

Because it can be hard to apply some of our new learning in a hands-on manner and because this chapter is heavier on commands than code, it might be helpful to have a quick exercise to explore a different way to discover the metric under-replicated partitions rather than the exporter we saw earlier. Besides using something like a dashboard to see this data, what command line options can we use to discover this information?

Let's say that we want to confirm the health of one of our topics named `kinaction_replica_test`. We created this topic with each partition having three replicas. We want to make sure we have three brokers listed in the ISR list in case there is ever a broker failure. What command should we run to look at that topic and see its current status? Listing 6.5 shows an example describing that topic [27]. Notice that the `ReplicationFactor` is `3` and the `Replicas` list shows three broker IDs as well. However, the ISR list only shows two values when it should show three!

Listing 6.5 Describing the topic replica: a test for ISR count

```
$ bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \
  --topic kinaction_replica_test

Topic:kinaction_replica_test  PartitionCount:1  ReplicationFactor:3    Configs:
    Topic: kinaction_replica_test  Partition: 0

 Leader: 0  Replicas: 1,0,2  Isr: 0,2
```

Note the topic parameter and the describe flag in use.

Topic-specific information about leader, partition, and replicas

Although we can notice the under-replicated partitions issue by looking at the details of the command output, we could have also used the `--under-replicated-partitions` flag to see any problems quickly [27]. Listing 6.6 shows how to use this flag, which quickly filters out the hard-to-see ISR data and only outputs under-replicated partitions to the terminal.

Listing 6.6 Using the `under-replicated-partitions` flag

```
bin/kafka-topics.sh --describe --bootstrap-server localhost:9094 \
  --under-replicated-partitions

Topic: kinaction_replica_test  Partition: 0
➥ Leader: 0  Replicas: 1,0,2  Isr: 0,2
```

Note the under-replicated-partition flag in use.

The ISR only lists two brokers!

Listing 6.6 shows that when using the `--describe` flag, we do not have to limit the check for under-replicated partitions to a specific topic. We can run this command to display issues across topics and to quickly find issues on our cluster. We will explore

more of the out-of-the-box tools included with Kafka when we talk about administration tools in chapter 9.

> **TIP** When using any of the commands in this chapter, it is always a good idea to run the command without any parameters and read the command options that are available for troubleshooting.

As we examined more about Kafka in this chapter, we've come to realize we are running a complex system. However, there are various command line tools as well as metrics to help us monitor the health of our cluster. In our next chapter, we will continue to use commands to complete specific tasks for this dynamic system throughout its lifetime.

Summary

- Brokers are the centerpiece of Kafka and provide the logic with which external clients interface with our applications. Clusters provide not only scale but also reliability.
- We can use ZooKeeper to provide agreement in a distributed cluster. One example is to elect a new controller between multiple available brokers.
- To help manage our cluster, we can set configurations at the broker level, which our clients can override for specific options.
- Replicas allow for a number of copies of data to span across a cluster. This helps in the event a broker fails and cannot be reached.
- In-sync replicas (ISRs) are current with the leader's data and that can take over leadership for a partition without data loss.
- We can use metrics to help produce graphs to visually monitor a cluster or alert on potential issues.

References

1. "Post Kafka Deployment." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#balancing-replicas-across-racks (accessed September 15, 2019).
2. "KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum." Wiki for Apache Kafka. Apache Software Foundation (July 09, 2020). https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum (accessed August 22, 2020).
3. F. Junqueira and N. Narkhede. "Distributed Consensus Reloaded: Apache ZooKeeper and Replication in Apache Kafka." Confluent blog (August 27, 2015). https://www.confluent.io/blog/distributed-consensus-reloaded-apache-zookeeper-and-replication-in-kafka/ (accessed September 15, 2019).
4. "Kafka data structures in Zookeeper [sic]." Wiki for Apache Kafka. Apache Software Foundation (February 10, 2017). https://cwiki.apache.org/confluence/display/KAFKA/Kafka+data+structures+in+Zookeeper (accessed January 19, 2020).

5 C. McCabe. "Apache Kafka Needs No Keeper: Removing the Apache ZooKeeper Dependency." Confluent blog. (May 15, 2020). https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier (accessed August 20, 2021).

6 Apache Kafka Binder (n.d.). https://docs.spring.io/spring-cloud-stream-binder-kafka/docs/3.1.3/reference/html/spring-cloud-stream-binder-kafka.html#_apache_kafka_binder (accessed July 18, 2021).

7 "CLI Tools for Confluent Platform." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/cli-reference.html (accessed August 25, 2021).

8 "ZkData.scala." Apache Kafka GitHub. https://github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/core/src/main/scala/kafka/zk/ZkData.scala (accessed August 27, 2021).

9 "A Guide To The Kafka Protocol." Wiki for Apache Kafka. Apache Software Foundation (June 14, 2017). https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol (accessed September 15, 2019).

10 "Kafka Broker Configurations." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html (accessed August 21, 2021).

11 "Logging." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#logging (accessed August 21, 2021).

12 "Controller." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#controller (accessed August 21, 2021).

13 "Kafka Controller Internals." Wiki for Apache Kafka. Apache Software Foundation (January 26, 2014). https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals (accessed September 15, 2019).

14 "Post Kafka Deployment." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#rolling-restart (accessed July 10, 2019).

15 "Replication." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html#replication (accessed August 21, 2021).

16 "KIP-392: Allow consumers to fetch from closest replica." Wiki for Apache Kafka. Apache Software Foundation (November 5, 2019). https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica (accessed December 10, 2019).

17 N. Narkhede. "Hands-free Kafka Replication: A lesson in operational simplicity." Confluent blog (July 1, 2015). https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/ (accessed October 02, 2019).

18 "Observability Overview and Setup." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/tutorials/examples/ccloud-observability/docs/observability-overview.html (accessed August 26, 2021).

19 "Kafka Monitoring and Metrics Using JMX". Confluent documentation. (n.d.). https://docs.confluent.io/platform/current/installation/docker/operations/monitoring.html (accessed June 12, 2020).

20 "Scaling the Cluster (Adding a node to a Kafka cluster)." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#scaling-the-cluster-adding-a-node-to-a-ak-cluster (accessed August 21, 2021).

21 "Graceful shutdown." Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_restarting (accessed May 11, 2018).

22 C. McCabe. "Upgrading Apache Kafka Clients Just Got Easier." Confluent blog. (July 18, 2017). https://www.confluent.io/blog/upgrading-apache-kafka-clients-just-got-easier (accessed October 02, 2019).

23 "Backup and Restoration." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/post-deployment.html#backup-and-restoration (accessed August 21, 2021).

24 Release Notes, Kafka Version 2.4.0. Apache Software Foundation (n.d.). https://archive.apache.org/dist/kafka/2.4.0/RELEASE_NOTES.html (accessed May 12, 2020).

25 "Multi-DC Solutions." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/multi-dc-deployments/index.html#multi-dc-solutions (accessed August 21, 2021).

26 G. Shapira. "Recommendations_for_Deploying_Apache_Kafka_on_Kubernetes." White paper (2018). https://www.confluent.io/resources/recommendations-for-deploying-apache-kafka-on-kubernetes (accessed December 15, 2019).

27 "Replication tools." Wiki for Apache Kafka. Apache Software Foundation (February 4, 2019). https://cwiki.apache.org/confluence/display/kafka/replication+tools (accessed January 19, 2019).