11 Understanding Kubernetes internals

This chapter covers

- What components make up a Kubernetes cluster
- What each component does and how it does it
- How creating a Deployment object results in a running pod
- What a running pod is
- How the network between pods works
- How Kubernetes Services work
- How high-availability is achieved

By reading this book up to this point, you've become familiar with what Kubernetes has to offer and what it does. But so far, I've intentionally not spent much time explaining exactly how it does all this because, in my opinion, it makes no sense to go into details of how a system works until you have a good understanding of what the system does. That's why we haven't talked about exactly how a pod is scheduled or how the various controllers running inside the Controller Manager make deployed resources come to life. Because you now know most resources that can be deployed in Kubernetes, it's time to dive into how they're implemented.

11.1 Understanding the architecture

Before you look at how Kubernetes does what it does, let's take a closer look at the components that make up a Kubernetes cluster. In chapter 1, you saw that a Kubernetes cluster is split into two parts:

- The Kubernetes Control Plane
- The (worker) nodes

Let's look more closely at what these two parts do and what's running inside them.

COMPONENTS OF THE CONTROL PLANE

The Control Plane is what controls and makes the whole cluster function. To refresh your memory, the components that make up the Control Plane are

- The etcd distributed persistent storage
- The API server
- The Scheduler
- The Controller Manager

These components store and manage the state of the cluster, but they aren't what runs the application containers.

COMPONENTS RUNNING ON THE WORKER NODES

The task of running your containers is up to the components running on each worker node:

- The Kubelet
- The Kubernetes Service Proxy (kube-proxy)
- The Container Runtime (Docker, rkt, or others)

ADD-ON COMPONENTS

Beside the Control Plane components and the components running on the nodes, a few add-on components are required for the cluster to provide everything discussed so far. This includes

- The Kubernetes DNS server
- The Dashboard
- An Ingress controller
- Heapster, which we'll talk about in chapter 14
- The Container Network Interface network plugin (we'll explain it later in this chapter)

11.1.1 The distributed nature of Kubernetes components

The previously mentioned components all run as individual processes. The components and their inter-dependencies are shown in figure 11.1.

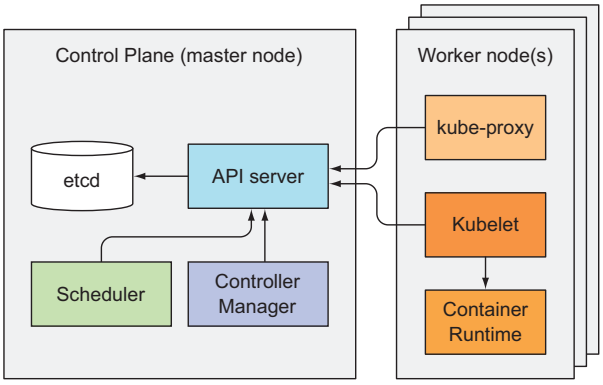

Figure 11.1 Kubernetes components of the Control Plane and the worker nodes

To get all the features Kubernetes provides, all these components need to be running. But several can also perform useful work individually without the other components. You'll see how as we examine each of them.

Checking the status of the Control Plane components

The API server exposes an API resource called ComponentStatus, which shows the health status of each Control Plane component. You can list the components and their statuses with `kubectl`:

```
$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE              ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

How these components communicate

Kubernetes system components communicate only with the API server. They don't talk to each other directly. The API server is the only component that communicates with etcd. None of the other components communicate with etcd directly, but instead modify the cluster state by talking to the API server.

Connections between the API server and the other components are almost always initiated by the components, as shown in figure 11.1. But the API server does connect to the Kubelet when you use `kubectl` to fetch logs, use `kubectl attach` to connect to a running container, or use the `kubectl port-forward` command.

NOTE The `kubectl attach` command is similar to `kubectl exec`, but it attaches to the main process running in the container instead of running an additional one.

Running multiple instances of individual components

Although the components on the worker nodes all need to run on the same node, the components of the Control Plane can easily be split across multiple servers. There

can be more than one instance of each Control Plane component running to ensure high availability. While multiple instances of etcd and API server can be active at the same time and do perform their jobs in parallel, only a single instance of the Scheduler and the Controller Manager may be active at a given time—with the others in standby mode.

HOW COMPONENTS ARE RUN

The Control Plane components, as well as kube-proxy, can either be deployed on the system directly or they can run as pods (as shown in listing 11.1). You may be surprised to hear this, but it will all make sense later when we talk about the Kubelet.

The Kubelet is the only component that always runs as a regular system component, and it's the Kubelet that then runs all the other components as pods. To run the Control Plane components as pods, the Kubelet is also deployed on the master. The next listing shows pods in the `kube-system` namespace in a cluster created with `kubeadm`, which is explained in appendix B.

Listing 11.1 Kubernetes components running as pods

```
$ kubectl get po -o custom-columns=POD:metadata.name,NODE:spec.nodeName
➥ --sort-by spec.nodeName -n kube-system
POD                                NODE
kube-controller-manager-master     master
kube-dns-2334855451-37d9k          master
etcd-master                        master
kube-apiserver-master              master
kube-scheduler-master              master
kube-flannel-ds-tgj9k              node1
kube-proxy-ny3xm                   node1
kube-flannel-ds-0eek8              node2
kube-proxy-sp362                   node2
kube-flannel-ds-r5yf4              node3
kube-proxy-og9ac                   node3
```

etcd, API server, Scheduler, Controller Manager, and the DNS server are running on the master.

The three nodes each run a Kube Proxy pod and a Flannel networking pod.

As you can see in the listing, all the Control Plane components are running as pods on the master node. There are three worker nodes, and each one runs the kube-proxy and a Flannel pod, which provides the overlay network for the pods (we'll talk about Flannel later).

> **TIP** As shown in the listing, you can tell `kubectl` to display custom columns with the `-o custom-columns` option and sort the resource list with `--sort-by`.

Now, let's look at each of the components up close, starting with the lowest level component of the Control Plane—the persistent storage.

11.1.2 How Kubernetes uses etcd

All the objects you've created throughout this book—Pods, ReplicationControllers, Services, Secrets, and so on—need to be stored somewhere in a persistent manner so their manifests survive API server restarts and failures. For this, Kubernetes uses etcd,

which is a fast, distributed, and consistent key-value store. Because it's distributed, you can run more than one etcd instance to provide both high availability and better performance.

The only component that talks to etcd directly is the Kubernetes API server. All other components read and write data to etcd indirectly through the API server. This brings a few benefits, among them a more robust optimistic locking system as well as validation; and, by abstracting away the actual storage mechanism from all the other components, it's much simpler to replace it in the future. It's worth emphasizing that etcd is the *only* place Kubernetes stores cluster state and metadata.

About optimistic concurrency control

Optimistic concurrency control (sometimes referred to as optimistic locking) is a method where instead of locking a piece of data and preventing it from being read or updated while the lock is in place, the piece of data includes a version number. Every time the data is updated, the version number increases. When updating the data, the version number is checked to see if it has increased between the time the client read the data and the time it submits the update. If this happens, the update is rejected and the client must re-read the new data and try to update it again.

The result is that when two clients try to update the same data entry, only the first one succeeds.

All Kubernetes resources include a `metadata.resourceVersion` field, which clients need to pass back to the API server when updating an object. If the version doesn't match the one stored in etcd, the API server rejects the update.

How resources are stored in etcd

As I'm writing this, Kubernetes can use either etcd version 2 or version 3, but version 3 is now recommended because of improved performance. etcd v2 stores keys in a hierarchical key space, which makes key-value pairs similar to files in a file system. Each key in etcd is either a directory, which contains other keys, or is a regular key with a corresponding value. etcd v3 doesn't support directories, but because the key format remains the same (keys can include slashes), you can still think of them as being grouped into directories. Kubernetes stores all its data in etcd under /registry. The following listing shows a list of keys stored under /registry.

Listing 11.2 Top-level entries stored in etcd by Kubernetes

```
$ etcdctl ls /registry
/registry/configmaps
/registry/daemonsets
/registry/deployments
/registry/events
/registry/namespaces
/registry/pods
...
```

You'll recognize that these keys correspond to the resource types you learned about in the previous chapters.

> **NOTE** If you're using v3 of the etcd API, you can't use the `ls` command to see the contents of a directory. Instead, you can list all keys that start with a given prefix with `etcdctl get /registry --prefix=true`.

The following listing shows the contents of the /registry/pods directory.

Listing 11.3 Keys in the /registry/pods directory

```
$ etcdctl ls /registry/pods
/registry/pods/default
/registry/pods/kube-system
```

As you can infer from the names, these two entries correspond to the `default` and the `kube-system` namespaces, which means pods are stored per namespace. The following listing shows the entries in the /registry/pods/default directory.

Listing 11.4 etcd entries for pods in the `default` namespace

```
$ etcdctl ls /registry/pods/default
/registry/pods/default/kubia-159041347-xk0vc
/registry/pods/default/kubia-159041347-wt6ga
/registry/pods/default/kubia-159041347-hp2o5
```

Each entry corresponds to an individual pod. These aren't directories, but key-value entries. The following listing shows what's stored in one of them.

Listing 11.5 An etcd entry representing a pod

```
$ etcdctl get /registry/pods/default/kubia-159041347-wt6ga
{"kind":"Pod","apiVersion":"v1","metadata":{"name":"kubia-159041347-wt6ga",
"generateName":"kubia-159041347-","namespace":"default","selfLink":...
```

You'll recognize that this is nothing other than a pod definition in JSON format. The API server stores the complete JSON representation of a resource in etcd. Because of etcd's hierarchical key space, you can think of all the stored resources as JSON files in a filesystem. Simple, right?

> **WARNING** Prior to Kubernetes version 1.7, the JSON manifest of a `Secret` resource was also stored like this (it wasn't encrypted). If someone got direct access to etcd, they knew all your Secrets. From version 1.7, Secrets are encrypted and thus stored much more securely.

ENSURING THE CONSISTENCY AND VALIDITY OF STORED OBJECTS

Remember Google's Borg and Omega systems mentioned in chapter 1, which are what Kubernetes is based on? Like Kubernetes, Omega also uses a centralized store to hold the state of the cluster, but in contrast, multiple Control Plane components access the store directly. All these components need to make sure they all adhere to

the same optimistic locking mechanism to handle conflicts properly. A single component not adhering fully to the mechanism may lead to inconsistent data.

Kubernetes improves this by requiring all other Control Plane components to go through the API server. This way updates to the cluster state are always consistent, because the optimistic locking mechanism is implemented in a single place, so less chance exists, if any, of error. The API server also makes sure that the data written to the store is always valid and that changes to the data are only performed by authorized clients.

ENSURING CONSISTENCY WHEN ETCD IS CLUSTERED

For ensuring high availability, you'll usually run more than a single instance of etcd. Multiple etcd instances will need to remain consistent. Such a distributed system needs to reach a consensus on what the actual state is. etcd uses the RAFT consensus algorithm to achieve this, which ensures that at any given moment, each node's state is either what the majority of the nodes agrees is the current state or is one of the previously agreed upon states.

Clients connecting to different nodes of an etcd cluster will either see the actual current state or one of the states from the past (in Kubernetes, the only etcd client is the API server, but there may be multiple instances).

The consensus algorithm requires a majority (or quorum) for the cluster to progress to the next state. As a result, if the cluster splits into two disconnected groups of nodes, the state in the two groups can never diverge, because to transition from the previous state to the new one, there needs to be more than half of the nodes taking part in the state change. If one group contains the majority of all nodes, the other one obviously doesn't. The first group can modify the cluster state, whereas the other one can't. When the two groups reconnect, the second group can catch up with the state in the first group (see figure 11.2).

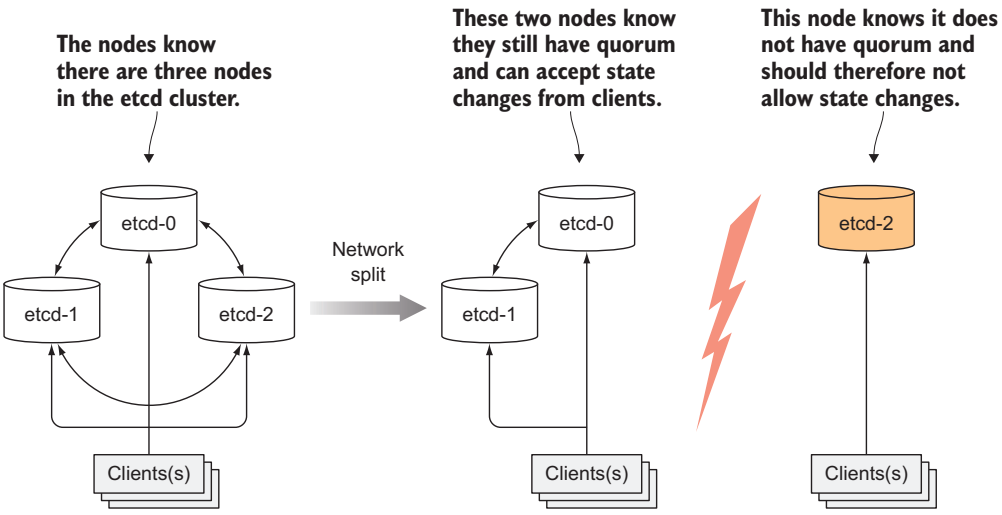

Figure 11.2 In a split-brain scenario, only the side which still has the majority (quorum) accepts state changes.

WHY THE NUMBER OF ETCD INSTANCES SHOULD BE AN ODD NUMBER

etcd is usually deployed with an odd number of instances. I'm sure you'd like to know why. Let's compare having two vs. having one instance. Having two instances requires both instances to be present to have a majority. If either of them fails, the etcd cluster can't transition to a new state because no majority exists. Having two instances is worse than having only a single instance. By having two, the chance of the whole cluster failing has increased by 100%, compared to that of a single-node cluster failing.

The same applies when comparing three vs. four etcd instances. With three instances, one instance can fail and a majority (of two) still exists. With four instances, you need three nodes for a majority (two aren't enough). In both three- and four-instance clusters, only a single instance may fail. But when running four instances, if one fails, a higher possibility exists of an additional instance of the three remaining instances failing (compared to a three-node cluster with one failed node and two remaining nodes).

Usually, for large clusters, an etcd cluster of five or seven nodes is sufficient. It can handle a two- or a three-node failure, respectively, which suffices in almost all situations.

11.1.3 What the API server does

The Kubernetes API server is the central component used by all other components and by clients, such as `kubectl`. It provides a CRUD (Create, Read, Update, Delete) interface for querying and modifying the cluster state over a RESTful API. It stores that state in etcd.

In addition to providing a consistent way of storing objects in etcd, it also performs validation of those objects, so clients can't store improperly configured objects (which they could if they were writing to the store directly). Along with validation, it also handles optimistic locking, so changes to an object are never overridden by other clients in the event of concurrent updates.

One of the API server's clients is the command-line tool `kubectl` you've been using from the beginning of the book. When creating a resource from a JSON file, for example, `kubectl` posts the file's contents to the API server through an HTTP POST request. Figure 11.3 shows what happens inside the API server when it receives the request. This is explained in more detail in the next few paragraphs.

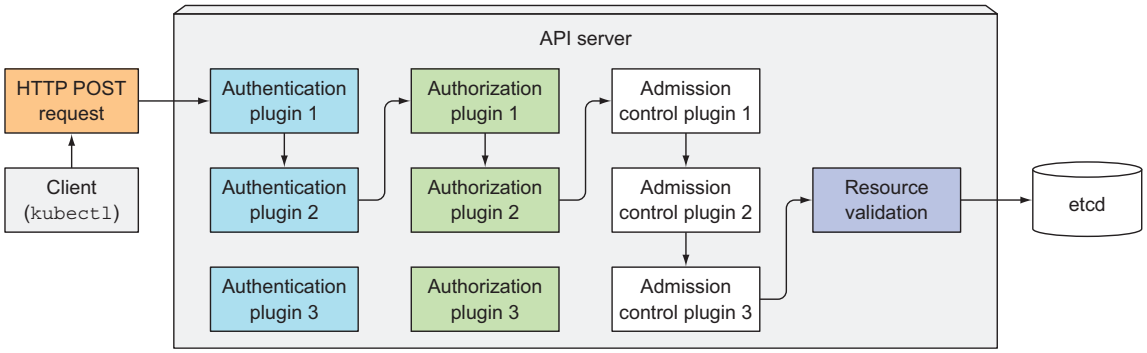

Figure 11.3 The operation of the API server

AUTHENTICATING THE CLIENT WITH AUTHENTICATION PLUGINS

First, the API server needs to authenticate the client sending the request. This is performed by one or more authentication plugins configured in the API server. The API server calls these plugins in turn, until one of them determines who is sending the request. It does this by inspecting the HTTP request.

Depending on the authentication method, the user can be extracted from the client's certificate or an HTTP header, such as `Authorization`, which you used in chapter 8. The plugin extracts the client's username, user ID, and groups the user belongs to. This data is then used in the next stage, which is authorization.

AUTHORIZING THE CLIENT WITH AUTHORIZATION PLUGINS

Besides authentication plugins, the API server is also configured to use one or more authorization plugins. Their job is to determine whether the authenticated user can perform the requested action on the requested resource. For example, when creating pods, the API server consults all authorization plugins in turn, to determine whether the user can create pods in the requested namespace. As soon as a plugin says the user can perform the action, the API server progresses to the next stage.

VALIDATING AND/OR MODIFYING THE RESOURCE IN THE REQUEST WITH ADMISSION CONTROL PLUGINS

If the request is trying to create, modify, or delete a resource, the request is sent through Admission Control. Again, the server is configured with multiple Admission Control plugins. These plugins can modify the resource for different reasons. They may initialize fields missing from the resource specification to the configured default values or even override them. They may even modify other related resources, which aren't in the request, and can also reject a request for whatever reason. The resource passes through all Admission Control plugins.

> **NOTE** When the request is only trying to read data, the request doesn't go through the Admission Control.

Examples of Admission Control plugins include

- `AlwaysPullImages`—Overrides the pod's `imagePullPolicy` to `Always`, forcing the image to be pulled every time the pod is deployed.
- `ServiceAccount`—Applies the default service account to pods that don't specify it explicitly.
- `NamespaceLifecycle`—Prevents creation of pods in namespaces that are in the process of being deleted, as well as in non-existing namespaces.
- `ResourceQuota`—Ensures pods in a certain namespace only use as much CPU and memory as has been allotted to the namespace. We'll learn more about this in chapter 14.

You'll find a list of additional Admission Control plugins in the Kubernetes documentation at https://kubernetes.io/docs/admin/admission-controllers/.

VALIDATING THE RESOURCE AND STORING IT PERSISTENTLY

After letting the request pass through all the Admission Control plugins, the API server then validates the object, stores it in etcd, and returns a response to the client.

11.1.4 Understanding how the API server notifies clients of resource changes

The API server doesn't do anything else except what we've discussed. For example, it doesn't create pods when you create a ReplicaSet resource and it doesn't manage the endpoints of a service. That's what controllers in the Controller Manager do.

But the API server doesn't even tell these controllers what to do. All it does is enable those controllers and other components to observe changes to deployed resources. A Control Plane component can request to be notified when a resource is created, modified, or deleted. This enables the component to perform whatever task it needs in response to a change of the cluster metadata.

Clients watch for changes by opening an HTTP connection to the API server. Through this connection, the client will then receive a stream of modifications to the watched objects. Every time an object is updated, the server sends the new version of the object to all connected clients watching the object. Figure 11.4 shows how clients can watch for changes to pods and how a change to one of the pods is stored into etcd and then relayed to all clients watching pods at that moment.

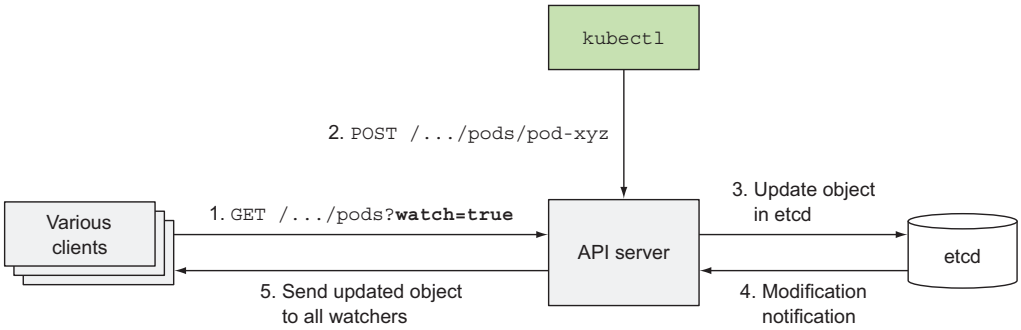

Figure 11.4 When an object is updated, the API server sends the updated object to all interested watchers.

One of the API server's clients is the `kubectl` tool, which also supports watching resources. For example, when deploying a pod, you don't need to constantly poll the list of pods by repeatedly executing `kubectl get pods`. Instead, you can use the `--watch` flag and be notified of each creation, modification, or deletion of a pod, as shown in the following listing.

Listing 11.6 Watching a pod being created and then deleted

```
$ kubectl get pods --watch
NAME                    READY     STATUS              RESTARTS    AGE
```

```
kubia-159041347-14j3i   0/1       Pending             0          0s
kubia-159041347-14j3i   0/1       Pending             0          0s
kubia-159041347-14j3i   0/1       ContainerCreating   0          1s
kubia-159041347-14j3i   0/1       Running             0          3s
kubia-159041347-14j3i   1/1       Running             0          5s
kubia-159041347-14j3i   1/1       Terminating         0          9s
kubia-159041347-14j3i   0/1       Terminating         0          17s
kubia-159041347-14j3i   0/1       Terminating         0          17s
kubia-159041347-14j3i   0/1       Terminating         0          17s
```

You can even have `kubectl` print out the whole YAML on each watch event like this:

```
$ kubectl get pods -o yaml --watch
```

The watch mechanism is also used by the Scheduler, which is the next Control Plane component you're going to learn more about.

11.1.5 Understanding the Scheduler

You've already learned that you don't usually specify which cluster node a pod should run on. This is left to the Scheduler. From afar, the operation of the Scheduler looks simple. All it does is wait for newly created pods through the API server's watch mechanism and assign a node to each new pod that doesn't already have the node set.

The Scheduler doesn't instruct the selected node (or the Kubelet running on that node) to run the pod. All the Scheduler does is update the pod definition through the API server. The API server then notifies the Kubelet (again, through the watch mechanism described previously) that the pod has been scheduled. As soon as the Kubelet on the target node sees the pod has been scheduled to its node, it creates and runs the pod's containers.

Although a coarse-grained view of the scheduling process seems trivial, the actual task of selecting the best node for the pod isn't that simple. Sure, the simplest Scheduler could pick a random node and not care about the pods already running on that node. On the other side of the spectrum, the Scheduler could use advanced techniques such as machine learning to anticipate what kind of pods are about to be scheduled in the next minutes or hours and schedule pods to maximize future hardware utilization without requiring any rescheduling of existing pods. Kubernetes' default Scheduler falls somewhere in between.

Understanding the default scheduling algorithm

The selection of a node can be broken down into two parts, as shown in figure 11.5:

- Filtering the list of all nodes to obtain a list of acceptable nodes the pod can be scheduled to.
- Prioritizing the acceptable nodes and choosing the best one. If multiple nodes have the highest score, round-robin is used to ensure pods are deployed across all of them evenly.

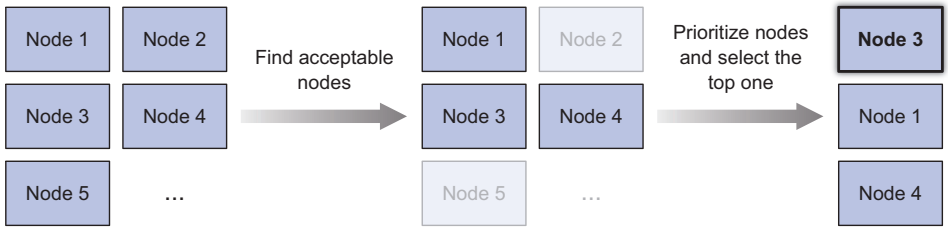

Figure 11.5 The Scheduler finds acceptable nodes for a pod and then selects the best node for the pod.

FINDING ACCEPTABLE NODES

To determine which nodes are acceptable for the pod, the Scheduler passes each node through a list of configured predicate functions. These check various things such as

- Can the node fulfill the pod's requests for hardware resources? You'll learn how to specify them in chapter 14.
- Is the node running out of resources (is it reporting a memory or a disk pressure condition)?
- If the pod requests to be scheduled to a specific node (by name), is this the node?
- Does the node have a label that matches the node selector in the pod specification (if one is defined)?
- If the pod requests to be bound to a specific host port (discussed in chapter 13), is that port already taken on this node or not?
- If the pod requests a certain type of volume, can this volume be mounted for this pod on this node, or is another pod on the node already using the same volume?
- Does the pod tolerate the taints of the node? Taints and tolerations are explained in chapter 16.
- Does the pod specify node and/or pod affinity or anti-affinity rules? If yes, would scheduling the pod to this node break those rules? This is also explained in chapter 16.

All these checks must pass for the node to be eligible to host the pod. After performing these checks on every node, the Scheduler ends up with a subset of the nodes. Any of these nodes could run the pod, because they have enough available resources for the pod and conform to all requirements you've specified in the pod definition.

SELECTING THE BEST NODE FOR THE POD

Even though all these nodes are acceptable and can run the pod, several may be a better choice than others. Suppose you have a two-node cluster. Both nodes are eligible, but one is already running 10 pods, while the other, for whatever reason, isn't running any pods right now. It's obvious the Scheduler should favor the second node in this case.

Or is it? If these two nodes are provided by the cloud infrastructure, it may be better to schedule the pod to the first node and relinquish the second node back to the cloud provider to save money.

Advanced scheduling of pods

Consider another example. Imagine having multiple replicas of a pod. Ideally, you'd want them spread across as many nodes as possible instead of having them all scheduled to a single one. Failure of that node would cause the service backed by those pods to become unavailable. But if the pods were spread across different nodes, a single node failure would barely leave a dent in the service's capacity.

Pods belonging to the same Service or ReplicaSet are spread across multiple nodes by default. It's not guaranteed that this is always the case. But you can force pods to be spread around the cluster or kept close together by defining pod affinity and anti-affinity rules, which are explained in chapter 16.

Even these two simple cases show how complex scheduling can be, because it depends on a multitude of factors. Because of this, the Scheduler can either be configured to suit your specific needs or infrastructure specifics, or it can even be replaced with a custom implementation altogether. You could also run a Kubernetes cluster without a Scheduler, but then you'd have to perform the scheduling manually.

Using multiple Schedulers

Instead of running a single Scheduler in the cluster, you can run multiple Schedulers. Then, for each pod, you specify the Scheduler that should schedule this particular pod by setting the `schedulerName` property in the pod spec.

Pods without this property set are scheduled using the default Scheduler, and so are pods with `schedulerName` set to `default-scheduler`. All other pods are ignored by the default Scheduler, so they need to be scheduled either manually or by another Scheduler watching for such pods.

You can implement your own Schedulers and deploy them in the cluster, or you can deploy an additional instance of Kubernetes' Scheduler with different configuration options.

11.1.6 Introducing the controllers running in the Controller Manager

As previously mentioned, the API server doesn't do anything except store resources in etcd and notify clients about the change. The Scheduler only assigns a node to the pod, so you need other active components to make sure the actual state of the system converges toward the desired state, as specified in the resources deployed through the API server. This work is done by controllers running inside the Controller Manager.

The single Controller Manager process currently combines a multitude of controllers performing various reconciliation tasks. Eventually those controllers will be split up into separate processes, enabling you to replace each one with a custom implementation if necessary. The list of these controllers includes the

- Replication Manager (a controller for ReplicationController resources)
- ReplicaSet, DaemonSet, and Job controllers

- Deployment controller
- StatefulSet controller
- Node controller
- Service controller
- Endpoints controller
- Namespace controller
- PersistentVolume controller
- Others

What each of these controllers does should be evident from its name. From the list, you can tell there's a controller for almost every resource you can create. Resources are descriptions of what should be running in the cluster, whereas the controllers are the active Kubernetes components that perform actual work as a result of the deployed resources.

UNDERSTANDING WHAT CONTROLLERS DO AND HOW THEY DO IT

Controllers do many different things, but they all watch the API server for changes to resources (Deployments, Services, and so on) and perform operations for each change, whether it's a creation of a new object or an update or deletion of an existing object. Most of the time, these operations include creating other resources or updating the watched resources themselves (to update the object's `status`, for example).

In general, controllers run a reconciliation loop, which reconciles the actual state with the desired state (specified in the resource's `spec` section) and writes the new actual state to the resource's `status` section. Controllers use the watch mechanism to be notified of changes, but because using watches doesn't guarantee the controller won't miss an event, they also perform a re-list operation periodically to make sure they haven't missed anything.

Controllers never talk to each other directly. They don't even know any other controllers exist. Each controller connects to the API server and, through the watch mechanism described in section 11.1.3, asks to be notified when a change occurs in the list of resources of any type the controller is responsible for.

We'll briefly look at what each of the controllers does, but if you'd like an in-depth view of what they do, I suggest you look at their source code directly. The sidebar explains how to get started.

A few pointers on exploring the controllers' source code

If you're interested in seeing exactly how these controllers operate, I strongly encourage you to browse through their source code. To make it easier, here are a few tips:

The source code for the controllers is available at https://github.com/kubernetes/kubernetes/blob/master/pkg/controller.

Each controller usually has a constructor in which it creates an `Informer`, which is basically a listener that gets called every time an API object gets updated. Usually,

an Informer listens for changes to a specific type of resource. Looking at the constructor will show you which resources the controller is watching.

Next, go look for the `worker()` method. In it, you'll find the method that gets invoked each time the controller needs to do something. The actual function is often stored in a field called `syncHandler` or something similar. This field is also initialized in the constructor, so that's where you'll find the name of the function that gets called. That function is the place where all the magic happens.

The Replication Manager

The controller that makes ReplicationController resources come to life is called the Replication Manager. We talked about how ReplicationControllers work in chapter 4. It's not the ReplicationControllers that do the actual work, but the Replication Manager. Let's quickly review what the controller does, because this will help you understand the rest of the controllers.

In chapter 4, we said that the operation of a ReplicationController could be thought of as an infinite loop, where in each iteration, the controller finds the number of pods matching its pod selector and compares the number to the desired replica count.

Now that you know how the API server can notify clients through the watch mechanism, it's clear that the controller doesn't poll the pods in every iteration, but is instead notified by the watch mechanism of each change that may affect the desired replica count or the number of matched pods (see figure 11.6). Any such changes trigger the controller to recheck the desired vs. actual replica count and act accordingly.

You already know that when too few pod instances are running, the Replication-Controller runs additional instances. But it doesn't actually run them itself. It creates

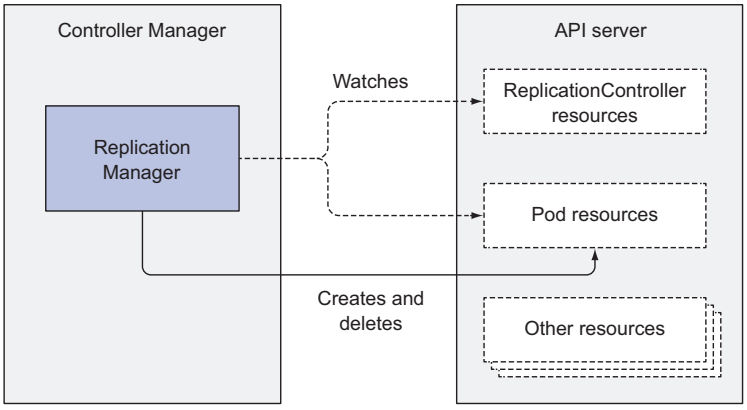

Figure 11.6 The Replication Manager watches for changes to API objects.

new Pod manifests, posts them to the API server, and lets the Scheduler and the Kubelet do their job of scheduling and running the pod.

The Replication Manager performs its work by manipulating Pod API objects through the API server. This is how all controllers operate.

THE REPLICASET, THE DAEMONSET, AND THE JOB CONTROLLERS

The ReplicaSet controller does almost the same thing as the Replication Manager described previously, so we don't have much to add here. The DaemonSet and Job controllers are similar. They create Pod resources from the pod template defined in their respective resources. Like the Replication Manager, these controllers don't run the pods, but post Pod definitions to the API server, letting the Kubelet create their containers and run them.

THE DEPLOYMENT CONTROLLER

The Deployment controller takes care of keeping the actual state of a deployment in sync with the desired state specified in the corresponding Deployment API object.

The Deployment controller performs a rollout of a new version each time a Deployment object is modified (if the modification should affect the deployed pods). It does this by creating a ReplicaSet and then appropriately scaling both the old and the new ReplicaSet based on the strategy specified in the Deployment, until all the old pods have been replaced with new ones. It doesn't create any pods directly.

THE STATEFULSET CONTROLLER

The StatefulSet controller, similarly to the ReplicaSet controller and other related controllers, creates, manages, and deletes Pods according to the spec of a StatefulSet resource. But while those other controllers only manage Pods, the StatefulSet controller also instantiates and manages PersistentVolumeClaims for each Pod instance.

THE NODE CONTROLLER

The Node controller manages the Node resources, which describe the cluster's worker nodes. Among other things, a Node controller keeps the list of Node objects in sync with the actual list of machines running in the cluster. It also monitors each node's health and evicts pods from unreachable nodes.

The Node controller isn't the only component making changes to Node objects. They're also changed by the Kubelet, and can obviously also be modified by users through REST API calls.

THE SERVICE CONTROLLER

In chapter 5, when we talked about Services, you learned that a few different types exist. One of them was the `LoadBalancer` service, which requests a load balancer from the infrastructure to make the service available externally. The Service controller is the one requesting and releasing a load balancer from the infrastructure, when a `LoadBalancer`-type Service is created or deleted.

THE ENDPOINTS CONTROLLER

You'll remember that Services aren't linked directly to pods, but instead contain a list of endpoints (IPs and ports), which is created and updated either manually or automatically according to the pod selector defined on the Service. The Endpoints controller is the active component that keeps the endpoint list constantly updated with the IPs and ports of pods matching the label selector.

As figure 11.7 shows, the controller watches both Services and Pods. When Services are added or updated or Pods are added, updated, or deleted, it selects Pods matching the Service's pod selector and adds their IPs and ports to the Endpoints resource. Remember, the Endpoints object is a standalone object, so the controller creates it if necessary. Likewise, it also deletes the Endpoints object when the Service is deleted.

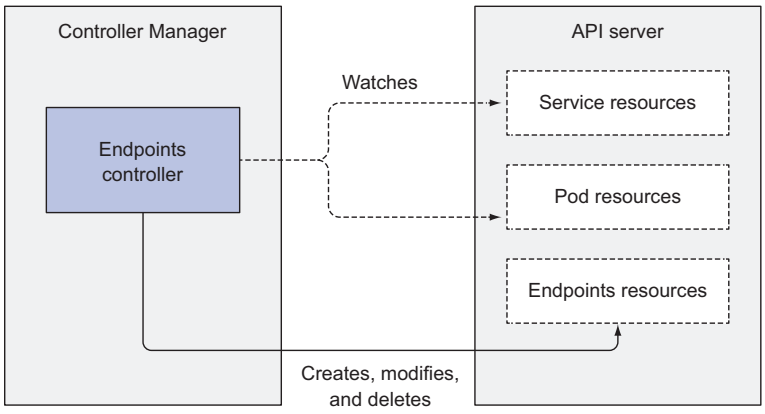

Figure 11.7 The Endpoints controller watches Service and Pod resources, and manages Endpoints.

THE NAMESPACE CONTROLLER

Remember namespaces (we talked about them in chapter 3)? Most resources belong to a specific namespace. When a Namespace resource is deleted, all the resources in that namespace must also be deleted. This is what the Namespace controller does. When it's notified of the deletion of a Namespace object, it deletes all the resources belonging to the namespace through the API server.

THE PERSISTENTVOLUME CONTROLLER

In chapter 6 you learned about PersistentVolumes and PersistentVolumeClaims. Once a user creates a PersistentVolumeClaim, Kubernetes must find an appropriate PersistentVolume and bind it to the claim. This is performed by the PersistentVolume controller.

When a PersistentVolumeClaim pops up, the controller finds the best match for the claim by selecting the smallest PersistentVolume with the access mode matching the one requested in the claim and the declared capacity above the capacity requested

in the claim. It does this by keeping an ordered list of PersistentVolumes for each access mode by ascending capacity and returning the first volume from the list.

Then, when the user deletes the PersistentVolumeClaim, the volume is unbound and reclaimed according to the volume's reclaim policy (left as is, deleted, or emptied).

CONTROLLER WRAP-UP

You should now have a good feel for what each controller does and how controllers work in general. Again, all these controllers operate on the API objects through the API server. They don't communicate with the Kubelets directly or issue any kind of instructions to them. In fact, they don't even know Kubelets exist. After a controller updates a resource in the API server, the Kubelets and Kubernetes Service Proxies, also oblivious of the controllers' existence, perform their work, such as spinning up a pod's containers and attaching network storage to them, or in the case of services, setting up the actual load balancing across pods.

The Control Plane handles one part of the operation of the whole system, so to fully understand how things unfold in a Kubernetes cluster, you also need to understand what the Kubelet and the Kubernetes Service Proxy do. We'll learn that next.

11.1.7 What the Kubelet does

In contrast to all the controllers, which are part of the Kubernetes Control Plane and run on the master node(s), the Kubelet and the Service Proxy both run on the worker nodes, where the actual pods containers run. What does the Kubelet do exactly?

UNDERSTANDING THE KUBELET'S JOB

In a nutshell, the Kubelet is the component responsible for everything running on a worker node. Its initial job is to register the node it's running on by creating a Node resource in the API server. Then it needs to continuously monitor the API server for Pods that have been scheduled to the node, and start the pod's containers. It does this by telling the configured container runtime (which is Docker, CoreOS' rkt, or something else) to run a container from a specific container image. The Kubelet then constantly monitors running containers and reports their status, events, and resource consumption to the API server.

The Kubelet is also the component that runs the container liveness probes, restarting containers when the probes fail. Lastly, it terminates containers when their Pod is deleted from the API server and notifies the server that the pod has terminated.

RUNNING STATIC PODS WITHOUT THE API SERVER

Although the Kubelet talks to the Kubernetes API server and gets the pod manifests from there, it can also run pods based on pod manifest files in a specific local directory as shown in figure 11.8. This feature is used to run the containerized versions of the Control Plane components as pods, as you saw in the beginning of the chapter.

Instead of running Kubernetes system components natively, you can put their pod manifests into the Kubelet's manifest directory and have the Kubelet run and manage

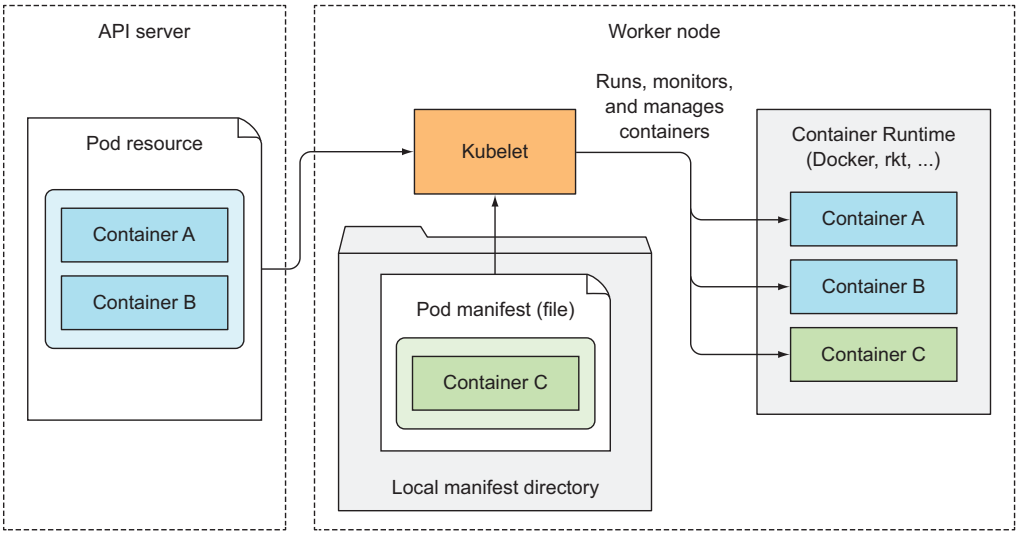

Figure 11.8 The Kubelet runs pods based on pod specs from the API server and a local file directory.

them. You can also use the same method to run your custom system containers, but doing it through a DaemonSet is the recommended method.

11.1.8 The role of the Kubernetes Service Proxy

Beside the Kubelet, every worker node also runs the kube-proxy, whose purpose is to make sure clients can connect to the services you define through the Kubernetes API. The kube-proxy makes sure connections to the service IP and port end up at one of the pods backing that service (or other, non-pod service endpoints). When a service is backed by more than one pod, the proxy performs load balancing across those pods.

WHY IT'S CALLED A PROXY

The initial implementation of the kube-proxy was the `userspace` proxy. It used an actual server process to accept connections and proxy them to the pods. To intercept connections destined to the service IPs, the proxy configured `iptables` rules (`iptables` is the tool for managing the Linux kernel's packet filtering features) to redirect the connections to the proxy server. A rough diagram of the `userspace` proxy mode is shown in figure 11.9.

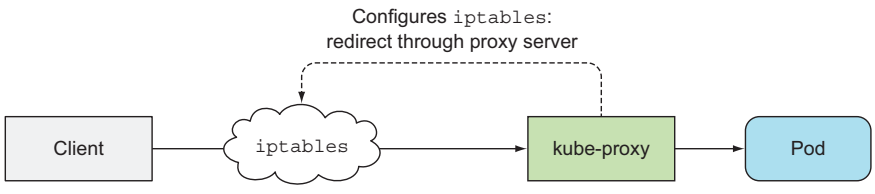

Figure 11.9 The `userspace` proxy mode

The kube-proxy got its name because it was an actual proxy, but the current, much better performing implementation only uses `iptables` rules to redirect packets to a randomly selected backend pod without passing them through an actual proxy server. This mode is called the `iptables` proxy mode and is shown in figure 11.10.

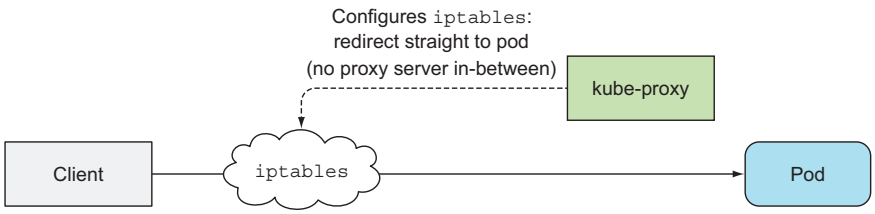

Figure 11.10 The `iptables` proxy mode

The major difference between these two modes is whether packets pass through the kube-proxy and must be handled in user space, or whether they're handled only by the Kernel (in kernel space). This has a major impact on performance.

Another smaller difference is that the `userspace` proxy mode balanced connections across pods in a true round-robin fashion, while the `iptables` proxy mode doesn't—it selects pods randomly. When only a few clients use a service, they may not be spread evenly across pods. For example, if a service has two backing pods but only five or so clients, don't be surprised if you see four clients connect to pod A and only one client connect to pod B. With a higher number of clients or pods, this problem isn't so apparent.

You'll learn exactly how `iptables` proxy mode works in section 11.5.

11.1.9 Introducing Kubernetes add-ons

We've now discussed the core components that make a Kubernetes cluster work. But in the beginning of the chapter, we also listed a few add-ons, which although not always required, enable features such as DNS lookup of Kubernetes services, exposing multiple HTTP services through a single external IP address, the Kubernetes web dashboard, and so on.

HOW ADD-ONS ARE DEPLOYED

These components are available as add-ons and are deployed as pods by submitting YAML manifests to the API server, the way you've been doing throughout the book. Some of these components are deployed through a Deployment resource or a ReplicationController resource, and some through a DaemonSet.

For example, as I'm writing this, in Minikube, the Ingress controller and the dashboard add-ons are deployed as ReplicationControllers, as shown in the following listing.

Listing 11.7 Add-ons deployed with ReplicationControllers in Minikube

```
$ kubectl get rc -n kube-system
NAME                        DESIRED   CURRENT   READY     AGE
default-http-backend        1         1         1         6d
kubernetes-dashboard        1         1         1         6d
nginx-ingress-controller    1         1         1         6d
```

The DNS add-on is deployed as a Deployment, as shown in the following listing.

Listing 11.8 The `kube-dns` Deployment

```
$ kubectl get deploy -n kube-system
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns   1         1         1            1           6d
```

Let's see how DNS and the Ingress controllers work.

How the DNS server works

All the pods in the cluster are configured to use the cluster's internal DNS server by default. This allows pods to easily look up services by name or even the pod's IP addresses in the case of headless services.

The DNS server pod is exposed through the `kube-dns` service, allowing the pod to be moved around the cluster, like any other pod. The service's IP address is specified as the `nameserver` in the /etc/resolv.conf file inside every container deployed in the cluster. The `kube-dns` pod uses the API server's watch mechanism to observe changes to Services and Endpoints and updates its DNS records with every change, allowing its clients to always get (fairly) up-to-date DNS information. I say fairly because during the time between the update of the Service or Endpoints resource and the time the DNS pod receives the watch notification, the DNS records may be invalid.

How (most) Ingress controllers work

Unlike the DNS add-on, you'll find a few different implementations of Ingress controllers, but most of them work in the same way. An Ingress controller runs a reverse proxy server (like Nginx, for example), and keeps it configured according to the Ingress, Service, and Endpoints resources defined in the cluster. The controller thus needs to observe those resources (again, through the watch mechanism) and change the proxy server's config every time one of them changes.

Although the Ingress resource's definition points to a Service, Ingress controllers forward traffic to the service's pod directly instead of going through the service IP. This affects the preservation of client IPs when external clients connect through the Ingress controller, which makes them preferred over Services in certain use cases.

Using other add-ons

You've seen how both the DNS server and the Ingress controller add-ons are similar to the controllers running in the Controller Manager, except that they also accept client connections instead of only observing and modifying resources through the API server.

Other add-ons are similar. They all need to observe the cluster state and perform the necessary actions when that changes. We'll introduce a few other add-ons in this and the remaining chapters.

11.1.10 Bringing it all together

You've now learned that the whole Kubernetes system is composed of relatively small, loosely coupled components with good separation of concerns. The API server, the Scheduler, the individual controllers running inside the Controller Manager, the Kubelet, and the kube-proxy all work together to keep the actual state of the system synchronized with what you specify as the desired state.

For example, submitting a pod manifest to the API server triggers a coordinated dance of various Kubernetes components, which eventually results in the pod's containers running. You'll learn how this dance unfolds in the next section.

11.2 How controllers cooperate

You now know about all the components that a Kubernetes cluster is comprised of. Now, to solidify your understanding of how Kubernetes works, let's go over what happens when a Pod resource is created. Because you normally don't create Pods directly, you're going to create a Deployment resource instead and see everything that must happen for the pod's containers to be started.

11.2.1 Understanding which components are involved

Even before you start the whole process, the controllers, the Scheduler, and the Kubelet are watching the API server for changes to their respective resource types. This is shown in figure 11.11. The components depicted in the figure will each play a part in the process you're about to trigger. The diagram doesn't include etcd, because it's hidden behind the API server, and you can think of the API server as the place where objects are stored.

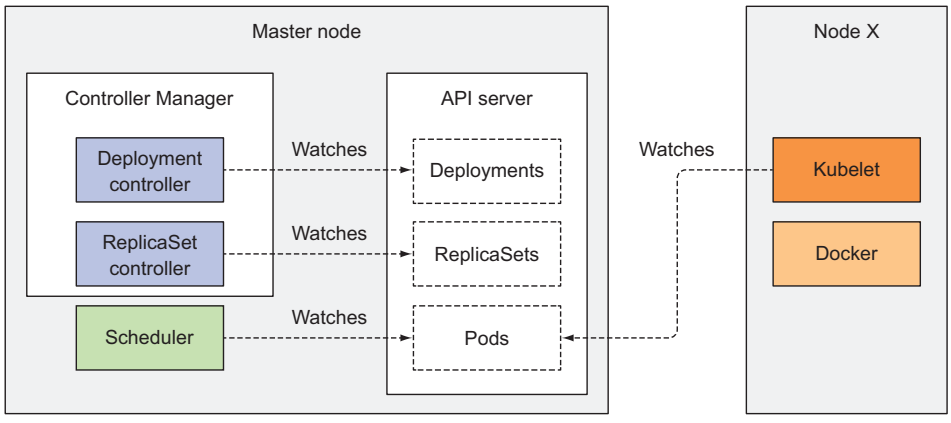

Figure 11.11 Kubernetes components watching API objects through the API server

11.2.2 The chain of events

Imagine you prepared the YAML file containing the Deployment manifest and you're about to submit it to Kubernetes through `kubectl`. `kubectl` sends the manifest to the Kubernetes API server in an HTTP POST request. The API server validates the Deployment specification, stores it in etcd, and returns a response to `kubectl`. Now a chain of events starts to unfold, as shown in figure 11.12.

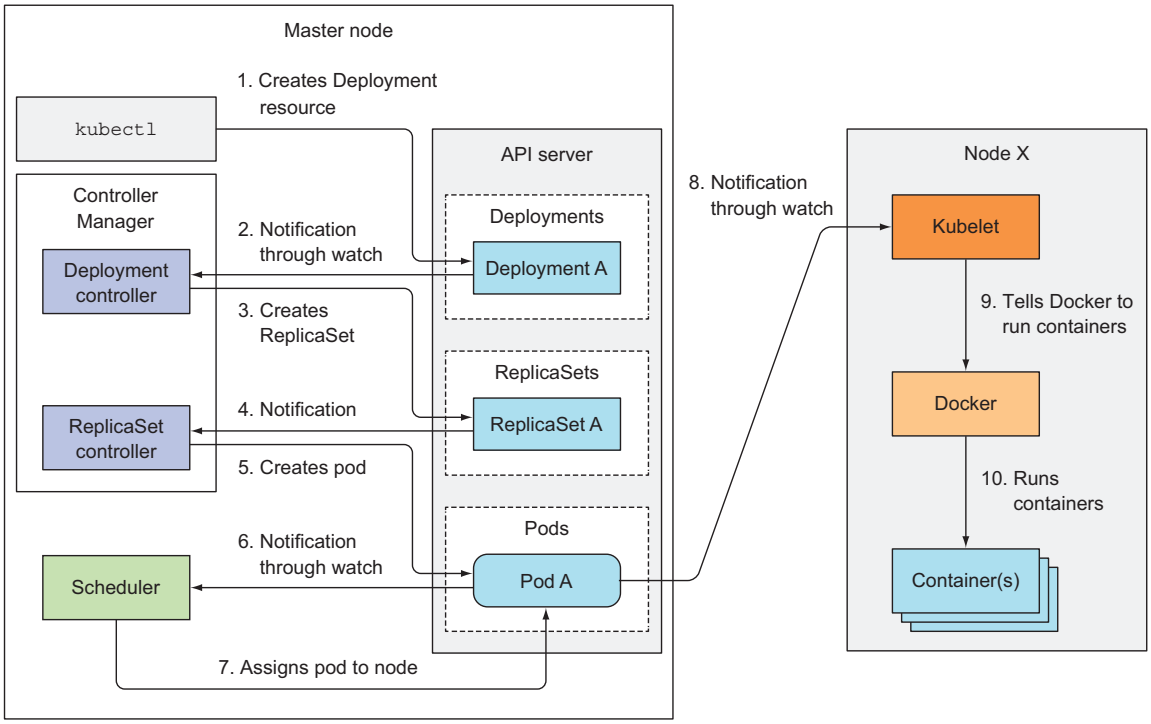

Figure 11.12 The chain of events that unfolds when a Deployment resource is posted to the API server

THE DEPLOYMENT CONTROLLER CREATES THE REPLICASET

All API server clients watching the list of Deployments through the API server's watch mechanism are notified of the newly created Deployment resource immediately after it's created. One of those clients is the Deployment controller, which, as we discussed earlier, is the active component responsible for handling Deployments.

As you may remember from chapter 9, a Deployment is backed by one or more ReplicaSets, which then create the actual pods. As a new Deployment object is detected by the Deployment controller, it creates a ReplicaSet for the current specification of the Deployment. This involves creating a new ReplicaSet resource through the Kubernetes API. The Deployment controller doesn't deal with individual pods at all.

THE REPLICASET CONTROLLER CREATES THE POD RESOURCES

The newly created ReplicaSet is then picked up by the ReplicaSet controller, which watches for creations, modifications, and deletions of ReplicaSet resources in the API server. The controller takes into consideration the replica count and pod selector defined in the ReplicaSet and verifies whether enough existing Pods match the selector.

The controller then creates the Pod resources based on the pod template in the ReplicaSet (the pod template was copied over from the Deployment when the Deployment controller created the ReplicaSet).

THE SCHEDULER ASSIGNS A NODE TO THE NEWLY CREATED PODS

These newly created Pods are now stored in etcd, but they each still lack one important thing—they don't have an associated node yet. Their `nodeName` attribute isn't set. The Scheduler watches for Pods like this, and when it encounters one, chooses the best node for the Pod and assigns the Pod to the node. The Pod's definition now includes the name of the node it should be running on.

Everything so far has been happening in the Kubernetes Control Plane. None of the controllers that have taken part in this whole process have done anything tangible except update the resources through the API server.

THE KUBELET RUNS THE POD'S CONTAINERS

The worker nodes haven't done anything up to this point. The pod's containers haven't been started yet. The images for the pod's containers haven't even been downloaded yet.

But with the Pod now scheduled to a specific node, the Kubelet on that node can finally get to work. The Kubelet, watching for changes to Pods on the API server, sees a new Pod scheduled to its node, so it inspects the Pod definition and instructs Docker, or whatever container runtime it's using, to start the pod's containers. The container runtime then runs the containers.

11.2.3 Observing cluster events

Both the Control Plane components and the Kubelet emit events to the API server as they perform these actions. They do this by creating Event resources, which are like any other Kubernetes resource. You've already seen events pertaining to specific resources every time you used `kubectl describe` to inspect those resources, but you can also retrieve events directly with `kubectl get events`.

Maybe it's me, but using `kubectl get` to inspect events is painful, because they're not shown in proper temporal order. Instead, if an event occurs multiple times, the event is displayed only once, showing when it was first seen, when it was last seen, and the number of times it occurred. Luckily, watching events with the `--watch` option is much easier on the eyes and useful for seeing what's happening in the cluster.

The following listing shows the events emitted in the process described previously (some columns have been removed and the output is edited heavily to make it legible in the limited space on the page).

Listing 11.9 Watching events emitted by the controllers

```
$ kubectl get events --watch
    NAME              KIND         REASON              SOURCE
... kubia             Deployment   ScalingReplicaSet   deployment-controller
                      ➥ Scaled up replica set kubia-193 to 3
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-w7ll2
... kubia-193-tpg6j   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-tpg6j to node1
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-39590
... kubia-193         ReplicaSet   SuccessfulCreate    replicaset-controller
                      ➥ Created pod: kubia-193-tpg6j
... kubia-193-39590   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-39590 to node2
... kubia-193-w7ll2   Pod          Scheduled           default-scheduler
                      ➥ Successfully assigned kubia-193-w7ll2 to node2
... kubia-193-tpg6j   Pod          Pulled              kubelet, node1
                      ➥ Container image already present on machine
... kubia-193-tpg6j   Pod          Created             kubelet, node1
                      ➥ Created container with id 13da752
... kubia-193-39590   Pod          Pulled              kubelet, node2
                      ➥ Container image already present on machine
... kubia-193-tpg6j   Pod          Started             kubelet, node1
                      ➥ Started container with id 13da752
... kubia-193-w7ll2   Pod          Pulled              kubelet, node2
                      ➥ Container image already present on machine
... kubia-193-39590   Pod          Created             kubelet, node2
                      ➥ Created container with id 8850184
...
```

As you can see, the `SOURCE` column shows the controller performing the action, and the `NAME` and `KIND` columns show the resource the controller is acting on. The `REASON` column and the `MESSAGE` column (shown in every second line) give more details about what the controller has done.

11.3 Understanding what a running pod is

With the pod now running, let's look more closely at what a running pod even is. If a pod contains a single container, do you think that the Kubelet just runs this single container, or is there more to it?

You've run several pods throughout this book. If you're the investigative type, you may have already snuck a peek at what exactly Docker ran when you created a pod. If not, let me explain what you'd see.

Imagine you run a single container pod. Let's say you create an Nginx pod:

```
$ kubectl run nginx --image=nginx
deployment "nginx" created
```

You can now `ssh` into the worker node running the pod and inspect the list of running Docker containers. I'm using Minikube to test this out, so to `ssh` into the single

node, I use `minikube ssh`. If you're using GKE, you can `ssh` into a node with `gcloud compute ssh <node name>`.

Once you're inside the node, you can list all the running containers with `docker ps`, as shown in the following listing.

Listing 11.10 Listing running Docker containers

```
docker@minikubeVM:~$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED
c917a6f3c3f7   nginx                  "nginx -g 'daemon off"   4 seconds ago
98b8bf797174   gcr.io/.../pause:3.0   "/pause"                 7 seconds ago
...
```

NOTE I've removed irrelevant information from the previous listing—this includes both columns and rows. I've also removed all the other running containers. If you're trying this out yourself, pay attention to the two containers that were created a few seconds ago.

As expected, you see the Nginx container, but also an additional container. Judging from the `COMMAND` column, this additional container isn't doing anything (the container's command is `"pause"`). If you look closely, you'll see that this container was created a few seconds before the Nginx container. What's its role?

This pause container is the container that holds all the containers of a pod together. Remember how all containers of a pod share the same network and other Linux namespaces? The pause container is an infrastructure container whose sole purpose is to hold all these namespaces. All other user-defined containers of the pod then use the namespaces of the pod infrastructure container (see figure 11.13).

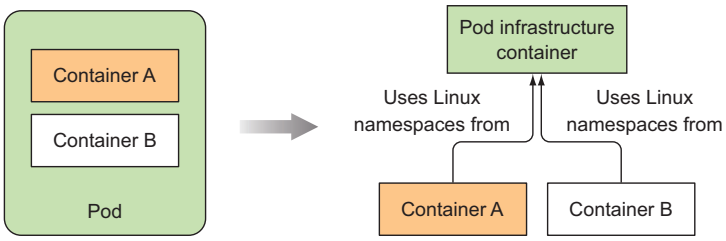

Figure 11.13 A two-container pod results in three running containers sharing the same Linux namespaces.

Actual application containers may die and get restarted. When such a container starts up again, it needs to become part of the same Linux namespaces as before. The infrastructure container makes this possible since its lifecycle is tied to that of the pod—the container runs from the time the pod is scheduled until the pod is deleted. If the infrastructure pod is killed in the meantime, the Kubelet recreates it and all the pod's containers.

11.4 Inter-pod networking

By now, you know that each pod gets its own unique IP address and can communicate with all other pods through a flat, NAT-less network. How exactly does Kubernetes achieve this? In short, it doesn't. The network is set up by the system administrator or by a Container Network Interface (CNI) plugin, not by Kubernetes itself.

11.4.1 What the network must be like

Kubernetes doesn't require you to use a specific networking technology, but it does mandate that the pods (or to be more precise, their containers) can communicate with each other, regardless if they're running on the same worker node or not. The network the pods use to communicate must be such that the IP address a pod sees as its own is the exact same address that all other pods see as the IP address of the pod in question.

Look at figure 11.14. When pod A connects to (sends a network packet to) pod B, the source IP pod B sees must be the same IP that pod A sees as its own. There should be no network address translation (NAT) performed in between—the packet sent by pod A must reach pod B with both the source and destination address unchanged.

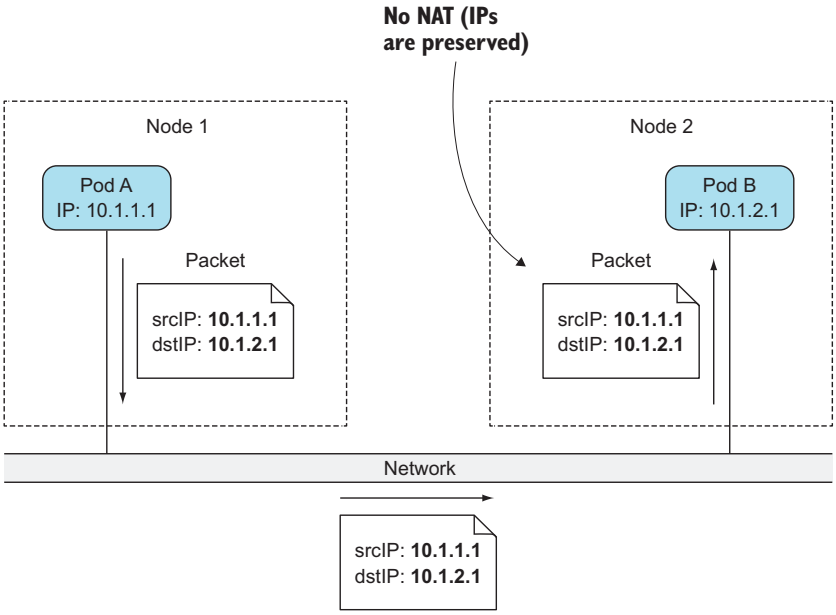

Figure 11.14 Kubernetes mandates pods are connected through a NAT-less network.

This is important, because it makes networking for applications running inside pods simple and exactly as if they were running on machines connected to the same network switch. The absence of NAT between pods enables applications running inside them to self-register in other pods.

For example, say you have a client pod X and pod Y, which provides a kind of notification service to all pods that register with it. Pod X connects to pod Y and tells it, "Hey, I'm pod X, available at IP 1.2.3.4; please send updates to me at this IP address." The pod providing the service can connect to the first pod by using the received IP address.

The requirement for NAT-less communication between pods also extends to pod-to-node and node-to-pod communication. But when a pod communicates with services out on the internet, the source IP of the packets the pod sends does need to be changed, because the pod's IP is private. The source IP of outbound packets is changed to the host worker node's IP address.

Building a proper Kubernetes cluster involves setting up the networking according to these requirements. There are various methods and technologies available to do this, each with its own benefits or drawbacks in a given scenario. Because of this, we're not going to go into specific technologies. Instead, let's explain how inter-pod networking works in general.

11.4.2 Diving deeper into how networking works

In section 11.3, we saw that a pod's IP address and network namespace are set up and held by the infrastructure container (the pause container). The pod's containers then use its network namespace. A pod's network interface is thus whatever is set up in the infrastructure container. Let's see how the interface is created and how it's connected to the interfaces in all the other pods. Look at figure 11.15. We'll discuss it next.

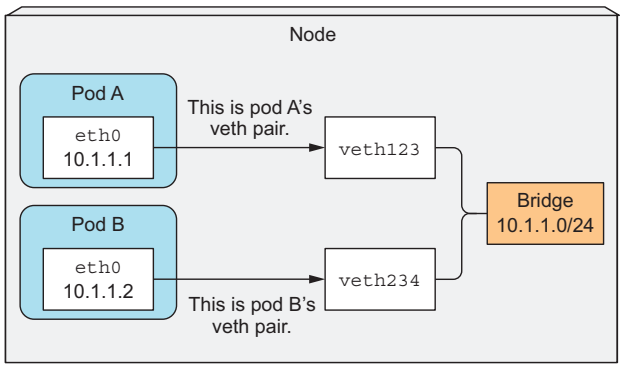

Figure 11.15 Pods on a node are connected to the same bridge through virtual Ethernet interface pairs.

ENABLING COMMUNICATION BETWEEN PODS ON THE SAME NODE

Before the infrastructure container is started, a virtual Ethernet interface pair (a veth pair) is created for the container. One interface of the pair remains in the host's namespace (you'll see it listed as `vethXXX` when you run `ifconfig` on the node), whereas the other is moved into the container's network namespace and renamed `eth0`. The two virtual interfaces are like two ends of a pipe (or like two network devices connected by an Ethernet cable)—what goes in on one side comes out on the other, and vice-versa.

The interface in the host's network namespace is attached to a network bridge that the container runtime is configured to use. The `eth0` interface in the container is assigned an IP address from the bridge's address range. Anything that an application running inside the container sends to the `eth0` network interface (the one in the container's namespace), comes out at the other veth interface in the host's namespace and is sent to the bridge. This means it can be received by any network interface that's connected to the bridge.

If pod A sends a network packet to pod B, the packet first goes through pod A's veth pair to the bridge and then through pod B's veth pair. All containers on a node are connected to the same bridge, which means they can all communicate with each other. But to enable communication between containers running on different nodes, the bridges on those nodes need to be connected somehow.

ENABLING COMMUNICATION BETWEEN PODS ON DIFFERENT NODES

You have many ways to connect bridges on different nodes. This can be done with overlay or underlay networks or by regular layer 3 routing, which we'll look at next.

You know pod IP addresses must be unique across the whole cluster, so the bridges across the nodes must use non-overlapping address ranges to prevent pods on different nodes from getting the same IP. In the example shown in figure 11.16, the bridge on node A is using the 10.1.1.0/24 IP range and the bridge on node B is using 10.1.2.0/24, which ensures no IP address conflicts exist.

Figure 11.16 shows that to enable communication between pods across two nodes with plain layer 3 networking, the node's physical network interface needs to be connected to the bridge as well. Routing tables on node A need to be configured so all packets destined for 10.1.2.0/24 are routed to node B, whereas node B's routing tables need to be configured so packets sent to 10.1.1.0/24 are routed to node A.

With this type of setup, when a packet is sent by a container on one of the nodes to a container on the other node, the packet first goes through the veth pair, then

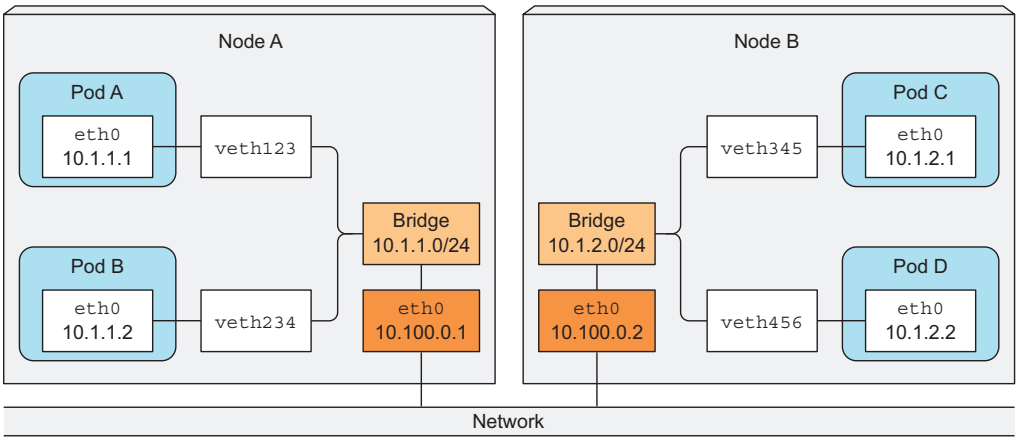

Figure 11.16 For pods on different nodes to communicate, the bridges need to be connected somehow.

through the bridge to the node's physical adapter, then over the wire to the other node's physical adapter, through the other node's bridge, and finally through the veth pair of the destination container.

This works only when nodes are connected to the same network switch, without any routers in between; otherwise those routers would drop the packets because they refer to pod IPs, which are private. Sure, the routers in between could be configured to route packets between the nodes, but this becomes increasingly difficult and error-prone as the number of routers between the nodes increases. Because of this, it's easier to use a Software Defined Network (SDN), which makes the nodes appear as though they're connected to the same network switch, regardless of the actual underlying network topology, no matter how complex it is. Packets sent from the pod are encapsulated and sent over the network to the node running the other pod, where they are de-encapsulated and delivered to the pod in their original form.

11.4.3 Introducing the Container Network Interface

To make it easier to connect containers into a network, a project called Container Network Interface (CNI) was started. The CNI allows Kubernetes to be configured to use any CNI plugin that's out there. These plugins include

- Calico
- Flannel
- Romana
- Weave Net
- And others

We're not going to go into the details of these plugins; if you want to learn more about them, refer to https://kubernetes.io/docs/concepts/cluster-administration/addons/.

Installing a network plugin isn't difficult. You only need to deploy a YAML containing a DaemonSet and a few other supporting resources. This YAML is provided on each plugin's project page. As you can imagine, the DaemonSet is used to deploy a network agent on all cluster nodes. It then ties into the CNI interface on the node, but be aware that the Kubelet needs to be started with `--network-plugin=cni` to use CNI.

11.5 How services are implemented

In chapter 5 you learned about Services, which allow exposing a set of pods at a long-lived, stable IP address and port. In order to focus on what Services are meant for and how they can be used, we intentionally didn't go into how they work. But to truly understand Services and have a better feel for where to look when things don't behave the way you expect, you need to understand how they are implemented.

11.5.1 Introducing the kube-proxy

Everything related to Services is handled by the kube-proxy process running on each node. Initially, the kube-proxy was an actual proxy waiting for connections and for each incoming connection, opening a new connection to one of the pods. This was called the `userspace` proxy mode. Later, a better-performing `iptables` proxy mode replaced it. This is now the default, but you can still configure Kubernetes to use the old mode if you want.

Before we continue, let's quickly review a few things about Services, which are relevant for understanding the next few paragraphs.

We've learned that each Service gets its own stable IP address and port. Clients (usually pods) use the service by connecting to this IP address and port. The IP address is virtual—it's not assigned to any network interfaces and is never listed as either the source or the destination IP address in a network packet when the packet leaves the node. A key detail of Services is that they consist of an IP and port pair (or multiple IP and port pairs in the case of multi-port Services), so the service IP by itself doesn't represent anything. That's why you can't ping them.

11.5.2 How kube-proxy uses iptables

When a service is created in the API server, the virtual IP address is assigned to it immediately. Soon afterward, the API server notifies all kube-proxy agents running on the worker nodes that a new Service has been created. Then, each kube-proxy makes that service addressable on the node it's running on. It does this by setting up a few `iptables` rules, which make sure each packet destined for the service IP/port pair is intercepted and its destination address modified, so the packet is redirected to one of the pods backing the service.

Besides watching the API server for changes to Services, kube-proxy also watches for changes to Endpoints objects. We talked about them in chapter 5, but let me refresh your memory, as it's easy to forget they even exist, because you rarely create them manually. An Endpoints object holds the IP/port pairs of all the pods that back the service (an IP/port pair can also point to something other than a pod). That's why the kube-proxy must also watch all Endpoints objects. After all, an Endpoints object changes every time a new backing pod is created or deleted, and when the pod's readiness status changes or the pod's labels change and it falls in or out of scope of the service.

Now let's see how kube-proxy enables clients to connect to those pods through the Service. This is shown in figure 11.17.

The figure shows what the `kube-proxy` does and how a packet sent by a client pod reaches one of the pods backing the Service. Let's examine what happens to the packet when it's sent by the client pod (pod A in the figure).

The packet's destination is initially set to the IP and port of the Service (in the example, the Service is at 172.30.0.1:80). Before being sent to the network, the

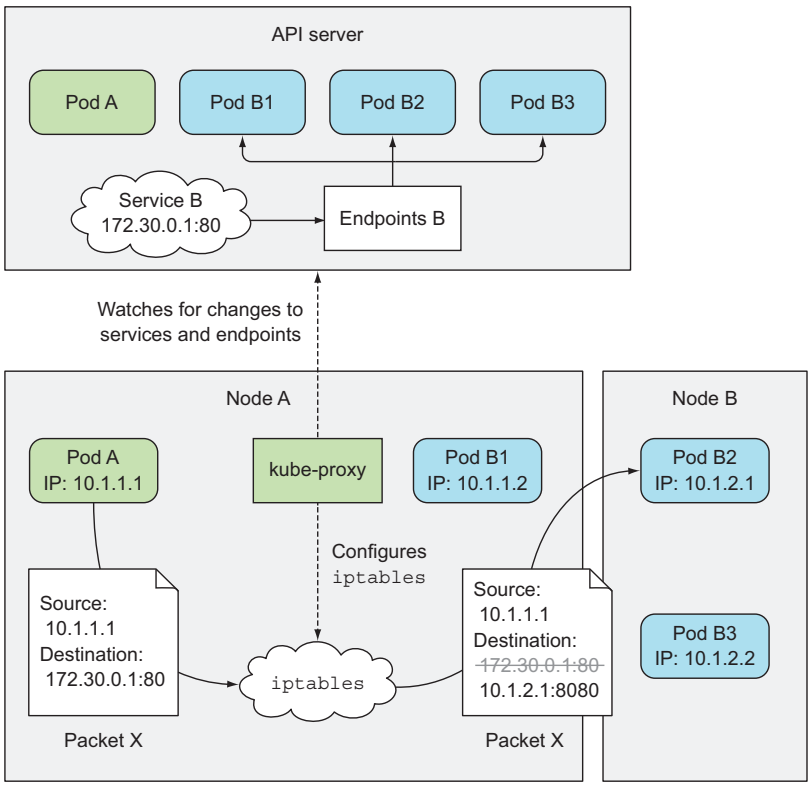

Figure 11.17 Network packets sent to a Service's virtual IP/port pair are modified and redirected to a randomly selected backend pod.

packet is first handled by node A's kernel according to the `iptables` rules set up on the node.

The kernel checks if the packet matches any of those `iptables` rules. One of them says that if any packet has the destination IP equal to 172.30.0.1 and destination port equal to 80, the packet's destination IP and port should be replaced with the IP and port of a randomly selected pod.

The packet in the example matches that rule and so its destination IP/port is changed. In the example, pod B2 was randomly selected, so the packet's destination IP is changed to 10.1.2.1 (pod B2's IP) and the port to 8080 (the target port specified in the Service spec). From here on, it's exactly as if the client pod had sent the packet to pod B directly instead of through the service.

It's slightly more complicated than that, but that's the most important part you need to understand.

11.6 Running highly available clusters

One of the reasons for running apps inside Kubernetes is to keep them running without interruption with no or limited manual intervention in case of infrastructure failures. For running services without interruption it's not only the apps that need to be up all the time, but also the Kubernetes Control Plane components. We'll look at what's involved in achieving high availability next.

11.6.1 Making your apps highly available

When running apps in Kubernetes, the various controllers make sure your app keeps running smoothly and at the specified scale even when nodes fail. To ensure your app is highly available, you only need to run them through a Deployment resource and configure an appropriate number of replicas; everything else is taken care of by Kubernetes.

Running multiple instances to reduce the likelihood of downtime

This requires your apps to be horizontally scalable, but even if that's not the case in your app, you should still use a Deployment with its replica count set to one. If the replica becomes unavailable, it will be replaced with a new one quickly, although that doesn't happen instantaneously. It takes time for all the involved controllers to notice that a node has failed, create the new pod replica, and start the pod's containers. There will inevitably be a short period of downtime in between.

Using leader-election for non-horizontally scalable apps

To avoid the downtime, you need to run additional inactive replicas along with the active one and use a fast-acting lease or leader-election mechanism to make sure only one is active. In case you're unfamiliar with leader election, it's a way for multiple app instances running in a distributed environment to come to an agreement on which is the leader. That leader is either the only one performing tasks, while all others are waiting for the leader to fail and then becoming leaders themselves, or they can all be active, with the leader being the only instance performing writes, while all the others are providing read-only access to their data, for example. This ensures two instances are never doing the same job, if that would lead to unpredictable system behavior due to race conditions.

The mechanism doesn't need to be incorporated into the app itself. You can use a sidecar container that performs all the leader-election operations and signals the main container when it should become active. You'll find an example of leader election in Kubernetes at https://github.com/kubernetes/contrib/tree/master/election.

Ensuring your apps are highly available is relatively simple, because Kubernetes takes care of almost everything. But what if Kubernetes itself fails? What if the servers running the Kubernetes Control Plane components go down? How are those components made highly available?

11.6.2 Making Kubernetes Control Plane components highly available

In the beginning of this chapter, you learned about the few components that make up a Kubernetes Control Plane. To make Kubernetes highly available, you need to run multiple master nodes, which run multiple instances of the following components:

- etcd, which is the distributed data store where all the API objects are kept
- API server
- Controller Manager, which is the process in which all the controllers run
- Scheduler

Without going into the actual details of how to install and run these components, let's see what's involved in making each of these components highly available. Figure 11.18 shows an overview of a highly available cluster.

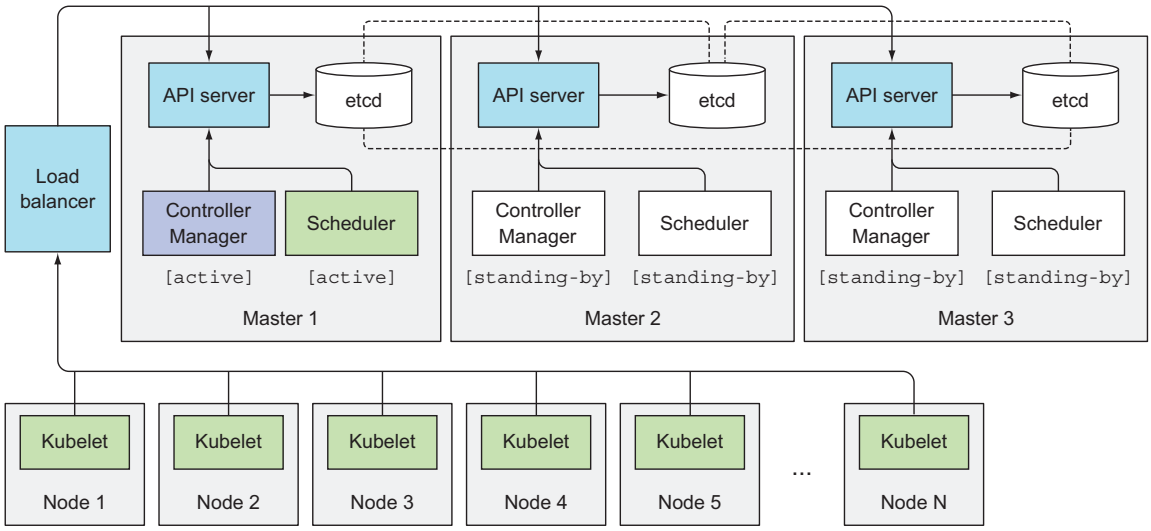

Figure 11.18 A highly-available cluster with three master nodes

RUNNING AN ETCD CLUSTER

Because etcd was designed as a distributed system, one of its key features is the ability to run multiple etcd instances, so making it highly available is no big deal. All you need to do is run it on an appropriate number of machines (three, five, or seven, as explained earlier in the chapter) and make them aware of each other. You do this by including the list of all the other instances in every instance's configuration. For example, when starting an instance, you specify the IPs and ports where the other etcd instances can be reached.

etcd will replicate data across all its instances, so a failure of one of the nodes when running a three-machine cluster will still allow the cluster to accept both read and write operations. To increase the fault tolerance to more than a single node, you need to run five or seven etcd nodes, which would allow the cluster to handle two or three

node failures, respectively. Having more than seven etcd instances is almost never necessary and begins impacting performance.

Running multiple instances of the API server

Making the API server highly available is even simpler. Because the API server is (almost completely) stateless (all the data is stored in etcd, but the API server does cache it), you can run as many API servers as you need, and they don't need to be aware of each other at all. Usually, one API server is collocated with every etcd instance. By doing this, the etcd instances don't need any kind of load balancer in front of them, because every API server instance only talks to the local etcd instance.

The API servers, on the other hand, do need to be fronted by a load balancer, so clients (`kubectl`, but also the Controller Manager, Scheduler, and all the Kubelets) always connect only to the healthy API server instances.

Ensuring high availability of the controllers and the Scheduler

Compared to the API server, where multiple replicas can run simultaneously, running multiple instances of the Controller Manager or the Scheduler isn't as simple. Because controllers and the Scheduler all actively watch the cluster state and act when it changes, possibly modifying the cluster state further (for example, when the desired replica count on a ReplicaSet is increased by one, the ReplicaSet controller creates an additional pod), running multiple instances of each of those components would result in all of them performing the same action. They'd be racing each other, which could cause undesired effects (creating two new pods instead of one, as mentioned in the previous example).

For this reason, when running multiple instances of these components, only one instance may be active at any given time. Luckily, this is all taken care of by the components themselves (this is controlled with the `--leader-elect` option, which defaults to true). Each individual component will only be active when it's the elected leader. Only the leader performs actual work, whereas all other instances are standing by and waiting for the current leader to fail. When it does, the remaining instances elect a new leader, which then takes over the work. This mechanism ensures that two components are never operating at the same time and doing the same work (see figure 11.19).

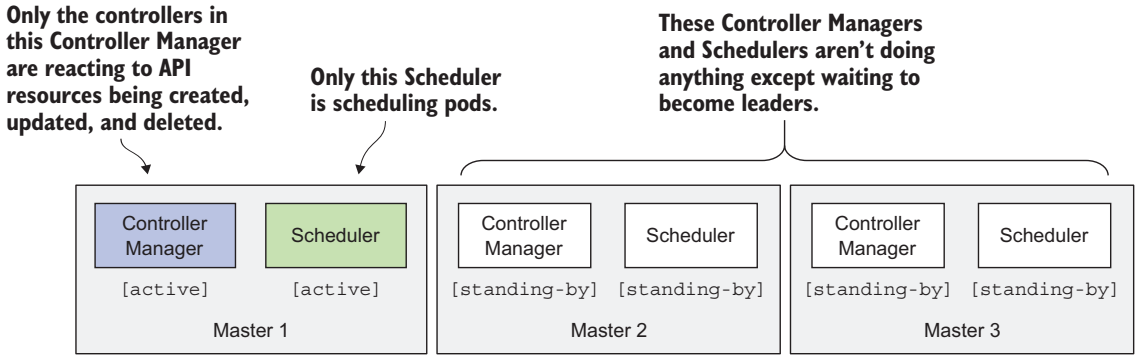

Figure 11.19 Only a single Controller Manager and a single Scheduler are active; others are standing by.

The Controller Manager and Scheduler can run collocated with the API server and etcd, or they can run on separate machines. When collocated, they can talk to the local API server directly; otherwise they connect to the API servers through the load balancer.

UNDERSTANDING THE LEADER ELECTION MECHANISM USED IN CONTROL PLANE COMPONENTS

What I find most interesting here is that these components don't need to talk to each other directly to elect a leader. The leader election mechanism works purely by creating a resource in the API server. And it's not even a special kind of resource—the Endpoints resource is used to achieve this (abused is probably a more appropriate term).

There's nothing special about using an Endpoints object to do this. It's used because it has no side effects as long as no Service with the same name exists. Any other resource could be used (in fact, the leader election mechanism will soon use ConfigMaps instead of Endpoints).

I'm sure you're interested in how a resource can be used for this purpose. Let's take the Scheduler, for example. All instances of the Scheduler try to create (and later update) an Endpoints resource called `kube-scheduler`. You'll find it in the `kube-system` namespace, as the following listing shows.

Listing 11.11 The `kube-scheduler` Endpoints resource used for leader-election

```
$ kubectl get endpoints kube-scheduler -n kube-system -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":
      ➥ "minikube","leaseDurationSeconds":15,"acquireTime":
      ➥ "2017-05-27T18:54:53Z","renewTime":"2017-05-28T13:07:49Z",
      ➥ "leaderTransitions":0}'
  creationTimestamp: 2017-05-27T18:54:53Z
  name: kube-scheduler
  namespace: kube-system
  resourceVersion: "654059"
  selfLink: /api/v1/namespaces/kube-system/endpoints/kube-scheduler
  uid: f847bd14-430d-11e7-9720-080027f8fa4e
subsets: []
```

The `control-plane.alpha.kubernetes.io/leader` annotation is the important part. As you can see, it contains a field called `holderIdentity`, which holds the name of the current leader. The first instance that succeeds in putting its name there becomes the leader. Instances race each other to do that, but there's always only one winner.

Remember the optimistic concurrency we explained earlier? That's what ensures that if multiple instances try to write their name into the resource only one of them succeeds. Based on whether the write succeeded or not, each instance knows whether it is or it isn't the leader.

Once becoming the leader, it must periodically update the resource (every two seconds by default), so all other instances know that it's still alive. When the leader fails,

other instances see that the resource hasn't been updated for a while, and try to become the leader by writing their own name to the resource. Simple, right?

11.7 Summary

Hopefully, this has been an interesting chapter that has improved your knowledge of the inner workings of Kubernetes. This chapter has shown you

- What components make up a Kubernetes cluster and what each component is responsible for
- How the API server, Scheduler, various controllers running in the Controller Manager, and the Kubelet work together to bring a pod to life
- How the infrastructure container binds together all the containers of a pod
- How pods communicate with other pods running on the same node through the network bridge, and how those bridges on different nodes are connected, so pods running on different nodes can talk to each other
- How the kube-proxy performs load balancing across pods in the same service by configuring `iptables` rules on the node
- How multiple instances of each component of the Control Plane can be run to make the cluster highly available

Next, we'll look at how to secure the API server and, by extension, the cluster as a whole.