Chapter 7. Service Discovery

Kubernetes is a very dynamic system. The system is involved in placing Pods on nodes, making sure they are up and running, and rescheduling them as needed. There are ways to automatically change the number of pods based on load (such as horizontal pod autoscaling [see "Autoscaling a ReplicaSet"]). The API-driven nature of the system encourages others to create higher and higher levels of automation.

While the dynamic nature of Kubernetes makes it easy to run a lot of things, it creates problems when it comes to *finding* those things. Most of the traditional network infrastructure wasn't built for the level of dynamism that Kubernetes presents.

What Is Service Discovery?

The general name for this class of problems and solutions is *service discovery*. Service discovery tools help solve the problem of finding which processes are listening at which addresses for which services. A good service discovery system will enable users to resolve this information quickly and reliably. A good system is also low-latency; clients are updated soon after the information associated with a service changes. Finally, a good service discovery system can store a richer definition of what that service is. For example, perhaps there are multiple ports associated with the service.

The Domain Name System (DNS) is the traditional system of service discovery on the internet. DNS is designed for relatively stable name resolution with wide and efficient caching. It is a great system for the internet but falls short in the dynamic world of Kubernetes.

Unfortunately, many systems (for example, Java, by default) look up a name in DNS directly and never re-resolve. This can lead to clients caching stale mappings and talking to the wrong IP. Even with short TTLs and well-behaved clients, there is a natural delay between when a name resolution changes and the client notices. There are natural limits to the amount and type of information that can be returned in a typical DNS query, too. Things start to break past 20–30 A records for a single name. SRV records solve some problems but are often very hard to use. Finally, the way that clients handle multiple IPs in a DNS record is usually to take the first IP address and rely on the DNS server to randomize or round-robin the order of records. This is no substitute for more purpose-built load balancing.

The Service Object

Real service discovery in Kubernetes starts with a `Service` object.

A `Service object` is a way to create a named label selector. As we will see, the `Service` object does some other nice things for us too.

Just as the `kubectl run` command is an easy way to create a Kubernetes deployment, we can use `kubectl expose` to create a service. Let's create some deployments and services so we can see how they work:

```
$ kubectl run alpaca-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
  --replicas=3 \
  --port=8080 \
  --labels="ver=1,app=alpaca,env=prod"
$ kubectl expose deployment alpaca-prod
$ kubectl run bandicoot-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=2 \
  --port=8080 \
  --labels="ver=2,app=bandicoot,env=prod"
$ kubectl expose deployment bandicoot-prod
$ kubectl get services -o wide

NAME             CLUSTER-IP     ... PORT(S)  ... SELECTOR
alpaca-prod      10.115.245.13  ... 8080/TCP ... app=alpaca,env=prod,ver=1
bandicoot-prod   10.115.242.3   ... 8080/TCP ... app=bandicoot,env=prod,ver=2
kubernetes       10.115.240.1   ... 443/TCP  ... <none>
```

After running these commands, we have three services. The ones we just created are `alpaca-prod` and `bandicoot-prod`. The `kubernetes` service is automatically created for you so that you can find and talk to the Kubernetes API from within the app.

If we look at the `SELECTOR` column, we see that the `alpaca-prod` service simply gives a name to a selector and specifies which ports to talk to for that service. The `kubectl expose` command will conveniently pull both the label selector and the relevant ports (8080, in this case) from the deployment definition.

Furthermore, that service is assigned a new type of virtual IP called a *cluster IP*. This is a special IP address the system will load-balance across all of the pods that are identified by the selector.

To interact with services, we are going to port-forward to one of the `alpaca`

pods. Start and leave this command running in a terminal window. You can see the port forward working by accessing the `alpaca` pod at *http://localhost:48858*:

```
$ ALPACA_POD=$(kubectl get pods -l app=alpaca \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $ALPACA_POD 48858:8080
```

Service DNS

Because the cluster IP is virtual it is stable and it is appropriate to give it a DNS address. All of the issues around clients caching DNS results no longer apply. Within a namespace, it is as easy as just using the service name to connect to one of the pods identified by a service.

Kubernetes provides a DNS service exposed to Pods running in the cluster. This Kubernetes DNS service was installed as a system component when the cluster was first created. The DNS service is, itself, managed by Kubernetes and is a great example of Kubernetes building on Kubernetes. The Kubernetes DNS service provides DNS names for cluster IPs.

You can try this out by expanding the "DNS Resolver" section on the `kuard` server status page. Query the A record for `alpaca-prod`. The output should look something like this:

```
;; opcode: QUERY, status: NOERROR, id: 12071
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;alpaca-prod.default.svc.cluster.local. IN       A

;; ANSWER SECTION:
alpaca-prod.default.svc.cluster.local.  30      IN      A       10.115.245.13
```

The full DNS name here is `alpaca-prod.default.svc.cluster.local.`. Let's break this down:

`alpaca-prod`
: The name of the service in question.

`default`
: The namespace that this service is in.

`svc`:: Recognizing that this is a service. This allows Kubernetes to expose other types of things as DNS in the future. *`cluster.local.`*
: The base domain name for the cluster. This is the default and what you will see for most clusters. Administrators may change this to allow unique DNS names across multiple clusters.

When referring to a service in your own namespace you can just use the service

name (`alpaca-prod`). You can also refer to a service in another namespace with `alpaca-prod.default`. And, of course, you can use the fully qualified service name (`alpaca-prod.default.svc.cluster.local.`). Try each of these out in the “DNS Resolver” section of `kuard`.

Readiness Checks

Oftentimes when an application first starts up it isn't ready to handle requests. There is usually some amount of initialization that can take anywhere from under a second to several minutes. One nice thing the `Service` object does is track which of your pods are ready via a readiness check. Let's modify our deployment to add a readiness check:

```
$ kubectl edit deployment/alpaca-prod
```

This command will fetch the current version of the `alpaca-prod` deployment and bring it up in an editor. After you save and quit your editor, it'll then write the object back to Kubernetes. This is a quick way to edit an object without saving it to a YAML file.

Add the following section:

```
spec:
  ...
  template:
    ...
    spec:
      containers:
        ...
        name: alpaca-prod
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          periodSeconds: 2
          initialDelaySeconds: 0
          failureThreshold: 3
          successThreshold: 1
```

This sets up the pods this deployment will create so that they will be checked for readiness via an HTTP `GET` to `/ready` on port 8080. This check is done every 2 seconds starting as soon as the pod comes up. If three successive checks fail, then the pod will be considered not ready. However, if only one check succeeds, then the pod will again be considered ready.

Only ready pods are sent traffic.

Updating the deployment definition like this will delete and recreate the `alpaca` pods. As such, we need to restart our `port-forward` command from earlier:

```
$ ALPACA_POD=$(kubectl get pods -l app=alpaca \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $ALPACA_POD 48858:8080
```

Open your browser to *http://localhost:48858* and you should see the debug page for that instance of `kuard`. Expand the "Readiness Check" section. You should see this page update every time there is a new readiness check from the system, which should happen every 2 seconds.

In another terminal window, start a `watch` command on the endpoints for the `alpaca-prod` service. Endpoints are a lower-level way of finding what a service is sending traffic to and are covered later in this chapter. The `--watch` option here causes the `kubectl` command to hang around and output any updates. This is an easy way to see how a Kubernetes object changes over time:

```
$ kubectl get endpoints alpaca-prod --watch
```

Now go back to your browser and hit the "Fail" link for the readiness check. You should see that the server is not returning 500s. After three of these this server is removed from the list of endpoints for the service. Hit the "Succeed" link and notice that after a single readiness check the endpoint is added back.

This readiness check is a way for an overloaded or sick server to signal to the system that it doesn't want to receive traffic anymore. This is a great way to implement graceful shutdown. The server can signal that it no longer wants traffic, wait until existing connections are closed, and then cleanly exit.

Press Control-C to exit out of both the `port-forward` and `watch` commands in your terminals.

Looking Beyond the Cluster

So far, everything we've covered in this chapter has been about exposing services inside of a cluster. Oftentimes the IPs for pods are only reachable from within the cluster. At some point, we have to allow new traffic in!

The most portable way to do this is to use a feature called `NodePorts`, which enhance a service even further. In addition to a cluster IP, the system picks a port (or the user can specify one), and every node in the cluster then forwards traffic to that port to the service.

With this feature, if you can reach any node in the cluster you can contact a service. You use the `NodePort` without knowing where any of the Pods for that service are running. This can be integrated with hardware or software load balancers to expose the service further.

Try this out by modifying the `alpaca-prod` service:

```
$ kubectl edit service alpaca-prod
```

Change the `spec.type` field to `NodePort`. You can also do this when creating the service via `kubectl expose` by specifying `--type=NodePort`. The system will assign a new `NodePort`:

```
$ kubectl describe service alpaca-prod

Name:                   alpaca-prod
Namespace:              default
Labels:                 app=alpaca
                        env=prod
                        ver=1
Annotations:            <none>
Selector:               app=alpaca,env=prod,ver=1
Type:                   NodePort
IP:                     10.115.245.13
Port:                   <unset> 8080/TCP
NodePort:               <unset> 32711/TCP
Endpoints:              10.112.1.66:8080,10.112.2.104:8080,10.112.2.105:8080
Session Affinity:       None
No events.
```

Here we see that the system assigned port 32711 to this service. Now we can hit any of our cluster nodes on that port to access the service. If you are sitting on

the same network, you can access it directly. If your cluster is in the cloud someplace, you can use SSH tunneling with something like this:

```
$ ssh <node> -L 8080:localhost:32711
```

Now if you open your browser to *http://localhost:8080* you will be connected to that service. Each request that you send to the service will be randomly directed to one of the Pods that implement the service. Reload the page a few times and you will see that you are randomly assigned to different pods.

When you are done, exit out of the SSH session.

Cloud Integration

Finally, if you have support from the cloud that you are running on (and your cluster is configured to take advantage of it) you can use the `LoadBalancer` type. This builds on `NodePorts` by additionally configuring the cloud to create a new load balancer and direct it at nodes in your cluster.

Edit the `alpaca-prod` service again (`kubectl edit service alpaca-prod`) and change `spec.type` to `LoadBalancer`.

If you do a `kubectl get services` right away you'll see that the `EXTERNAL-IP` column for `alpaca-prod` now says `<pending>`. Wait a bit and you should see a public address assigned by your cloud. You can look in the console for your cloud account and see the configuration work that Kubernetes did for you:

```
$ kubectl describe service alpaca-prod

Name:                   alpaca-prod
Namespace:              default
Labels:                 app=alpaca
                        env=prod
                        ver=1
Selector:               app=alpaca,env=prod,ver=1
Type:                   LoadBalancer
IP:                     10.115.245.13
LoadBalancer Ingress:   104.196.248.204
Port:                   <unset> 8080/TCP
NodePort:               <unset> 32711/TCP
Endpoints:              10.112.1.66:8080,10.112.2.104:8080,10.112.2.105:8080
Session Affinity:       None
Events:
  FirstSeen ... Reason               Message
  --------- ... ------               -------
  3m        ... Type                 NodePort -> LoadBalancer
  3m        ... CreatingLoadBalancer Creating load balancer
  2m        ... CreatedLoadBalancer  Created load balancer
```

Here we see that we have an address of 104.196.248.204 now assigned to the `alpaca-prod` service. Open up your browser and try!

This example is from a cluster launched and managed on the Google Cloud Platform via GKE. However, the way a `LoadBalancer` is configured is specific to a cloud. In addition, some clouds have DNS-based load balancers (e.g., AWS ELB). In this case you'll see a hostname here instead of an IP. Also, depending on the cloud provider, it may still take a little while for the load balancer to be

fully operational.

Advanced Details

Kubernetes is built to be an extensible system. As such, there are layers that allow for more advanced integrations. Understanding the details of how a sophisticated concept like services is implemented may help you troubleshoot or create more advanced integrations. This section goes a bit below the surface.

Endpoints

Some applications (and the system itself) want to be able to use services without using a cluster IP. This is done with another type of object called `Endpoints`. For every `Service` object, Kubernetes creates a buddy `Endpoints` object that contains the IP addresses for that service:

```
$ kubectl describe endpoints alpaca-prod

Name:           alpaca-prod
Namespace:      default
Labels:         app=alpaca
                env=prod
                ver=1
Subsets:
  Addresses:            10.112.1.54,10.112.2.84,10.112.2.85
  NotReadyAddresses:    <none>
  Ports:
    Name        Port    Protocol
    ----        ----    --------
    <unset>     8080    TCP

No events.
```

To use a service, an advanced application can talk to the Kubernetes API directly to look up endpoints and call them. The Kubernetes API even has the capability to "watch" objects and be notified as soon as they change. In this way a client can react immediately as soon as the IPs associated with a service change.

Let's demonstrate this. In a terminal window, start the following command and leave it running:

```
$ kubectl get endpoints alpaca-prod --watch
```

It will output the current state of the endpoint and then "hang":

```
NAME          ENDPOINTS                                            AGE
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080   1m
```

Now open up *another* terminal window and delete and recreate the deployment backing alpaca-prod:

```
$ kubectl delete deployment alpaca-prod
$ kubectl run alpaca-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
```

```
--replicas=3 \
--port=8080 \
--labels="ver=1,app=alpaca,env=prod"
```

If you look back at the output from the watched endpoint, you will see that as you deleted and re-created these pods, the output of the command reflected the most up-to-date set of IP addresses associated with the service. Your output will look something like this:

```
NAME          ENDPOINTS                                               AGE
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080   1m
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080   1m
alpaca-prod   <none>    1m
alpaca-prod   10.112.2.90:8080   1m
alpaca-prod   10.112.1.57:8080,10.112.2.90:8080   1m
alpaca-prod   10.112.0.28:8080,10.112.1.57:8080,10.112.2.90:8080   1m
```

The `Endpoints` object is great if you are writing new code that is built to run on Kubernetes from the start. But most projects aren’t in this position! Most existing systems are built to work with regular old IP addresses that don’t change that often.

Manual Service Discovery

Kubernetes services are built on top of label selectors over pods. That means that you can use the Kubernetes API to do rudimentary service discovery without using a `Service` object at all! Let's demonstrate.

With `kubectl` (and via the API) we can easily see what IPs are assigned to each pod in our example deployments:

```
$ kubectl get pods -o wide --show-labels

NAME                            ... IP          ... LABELS
alpaca-prod-12334-87f8h    ... 10.112.1.54 ... app=alpaca,env=prod,ver=1
alpaca-prod-12334-jssmh    ... 10.112.2.84 ... app=alpaca,env=prod,ver=1
alpaca-prod-12334-tjp56    ... 10.112.2.85 ... app=alpaca,env=prod,ver=1
bandicoot-prod-5678-sbxzl  ... 10.112.1.55 ... app=bandicoot,env=prod,ver=2
bandicoot-prod-5678-x0dh8  ... 10.112.2.86 ... app=bandicoot,env=prod,ver=2
```

This is great, but what if you have a ton of pods? You'll probably want to filter this based on the labels applied as part of the deployment. Let's do that for just the `alpaca` app:

```
$ kubectl get pods -o wide --selector=app=alpaca,env=prod

NAME                          ... IP          ...
alpaca-prod-3408831585-bpzdz ... 10.112.1.54 ...
alpaca-prod-3408831585-kncwt ... 10.112.2.84 ...
alpaca-prod-3408831585-l9fsq ... 10.112.2.85 ...
```

At this point we have the basics of service discovery! We can always use labels to identify the set of pods we are interested in, get all of the pods for those labels, and dig out the IP address. But keeping the correct set of labels to use in sync can be tricky. This is why the `Service` object was created.

kube-proxy and Cluster IPs

Cluster IPs are stable virtual IPs that load-balance traffic across all of the endpoints in a service. This magic is performed by a component running on every node in the cluster called the `kube-proxy` (Figure 7-1).

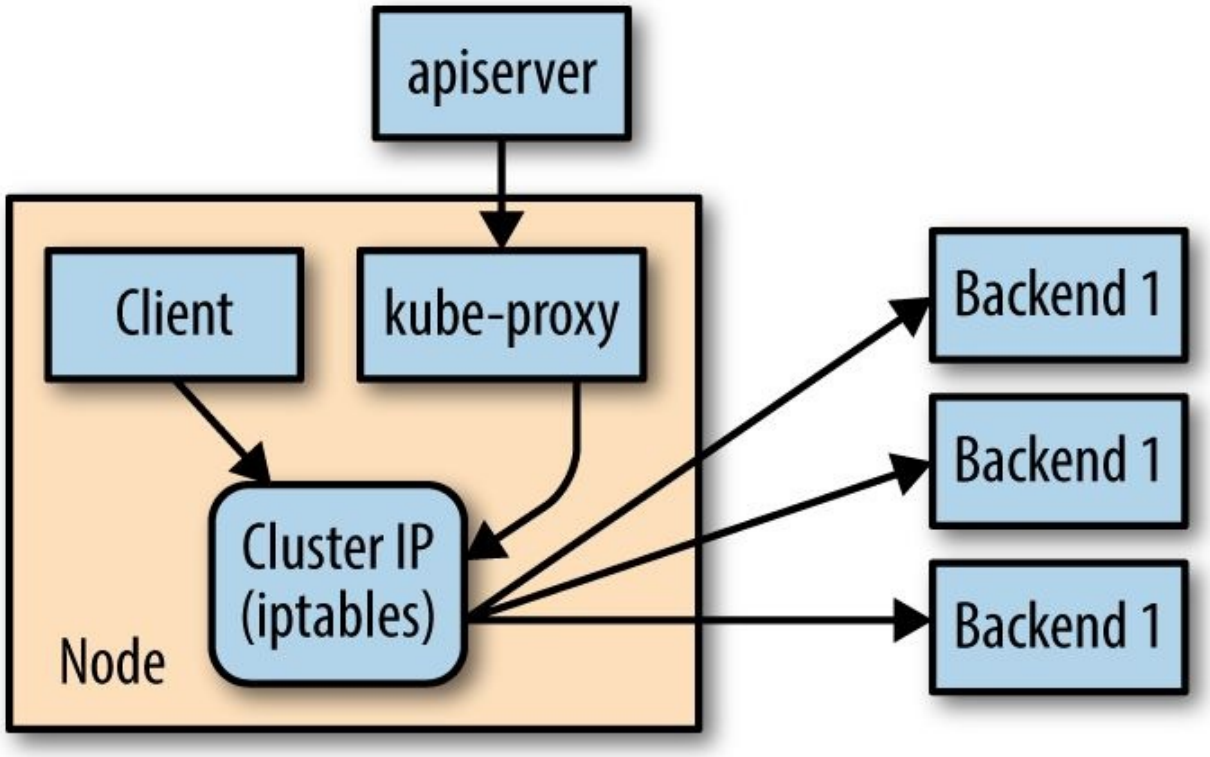

Figure 7-1. Configuring and using a cluster IP

In Figure 7-1, the `kube-proxy` watches for new services in the cluster via the API server. It then programs a set of `iptables` rules in the kernel of that host to rewrite the destination of packets so they are directed at one of the endpoints for that service. If the set of endpoints for a service changes (due to pods coming and going or due to a failed readiness check) the set of `iptables` rules is rewritten.

The cluster IP itself is usually assigned by the API server as the service is created. However, when creating the service, the user can specify a specific cluster IP. Once set, the cluster IP cannot be modified without deleting and

recreating the `Service` object.

> **NOTE**
>
> The Kubernetes service address range is configured using the `--service-cluster-ip-range` flag on the `kube-apiserver` binary. The service address range should not overlap with the IP subnets and ranges assigned to each Docker bridge or Kubernetes node.
>
> In addition, any explicit cluster IP requested must come from that range and not already be in use.

Cluster IP Environment Variables

While most users should be using the DNS services to find cluster IPs, there are some older mechanisms that may still be in use. One of these is injecting a set of environment variables into pods as they start up.

To see this in action, let's look at the console for the bandicoot instance of kuard. Enter the following commands in your terminal:

```
$ BANDICOOT_POD=$(kubectl get pods -l app=bandicoot \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $BANDICOOT_POD 48858:8080
```

Now open your browser to *http://localhost:48858* to see the status page for this server. Expand the "Environment" section and note the set of environment variables for the `alpaca` service. The status page should show a table similar to Table 7-1.

Table 7-1. Service environment variables

Name	Value
ALPACA_PROD_PORT	tcp://10.115.245.13:8080
ALPACA_PROD_PORT_8080_TCP	tcp://10.115.245.13:8080
ALPACA_PROD_PORT_8080_TCP_ADDR	10.115.245.13
ALPACA_PROD_PORT_8080_TCP_PORT	8080
ALPACA_PROD_PORT_8080_TCP_PROTO	tcp
ALPACA_PROD_SERVICE_HOST	10.115.245.13
ALPACA_PROD_SERVICE_PORT	8080

The two main environment variables to use are `ALPACA_PROD_SERVICE_HOST` and `ALPACA_PROD_SERVICE_PORT`. The other environment variables are created to be compatible with (now deprecated) Docker link variables.

A problem with the environment variable approach is that it requires resources to be created in a specific order. The services must be created before the pods that

reference them. This can introduce quite a bit of complexity when deploying a set of services that make up a larger application. In addition, using *just* environment variables seems strange to many users. For this reason, DNS is probably a better option.

Cleanup

Run the following commands to clean up all of the objects created in this chapter:

```
$ kubectl delete services,deployments -l app
```

Summary

Kubernetes is a dynamic system that challenges traditional methods of naming and connecting services over the network. The `Service` object provides a flexible and powerful way to expose services both within the cluster and beyond. With the techniques covered here you can connect services to each other and expose them outside the cluster.

While using the dynamic service discovery mechanisms in Kubernetes introduces some new concepts and may, at first, seem complex, understanding and adapting these techniques is key to unlocking the power of Kubernetes. Once your application can dynamically find services and react to the dynamic placement of those applications, you are free to stop worrying about where things are running and when they move. It is a critical piece of the puzzle to start to think about services in a logical way and let Kubernetes take care of the details of container placement.