2024 Q1 AI Voice SRE - Kubernetes Services

SRE, Site Reliability Engineering, is what you get when you treat operations as if it's a software problem.

Fundamental of Kubernetes Services

The need for services is due to the fact that pods in Kubernetes are short lived, and they can be replaced at any time. Kubernetes guarantees the availability of a given pod and replica, but not the liveness of individual pods. This means that pods that need to communicate with another pod cannot rely on the IP address of the underlying single pod. Instead, they connect to the service, which relays them to a relevant, currently-running pod.

The service is assigned a virtual IP address, known as a clusterIP, which persists until it is explicitly destroyed. The service acts as a reliable endpoint for communication between components or applications.

For Kubernetes native applications, an alternative to using services is to make requests directly through the Kubernetes API Server. The API Server automatically exposes and maintains an endpoint for running pods.

The basic components of a Kubernetes service are a label selector that identifies pods to route traffic to, a clusterIP and port number, port definitions, and optional mapping of incoming ports to a targetPort.

Another, less popular option is to create a service without a pod selector. This lets you point a service to another namespace, another service in the cluster, or a static IP outside the cluster.

Example of creating a Kubernetes Service via YAML manifest

Example of a Kubernetes Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: nginx
  ports:
 -protocol: TCP
    port: 80
    targetPort: 8080
```

Deploying the service to the Kubernetes cluster

kubectl

```
$ kubectl apply -f /mypath/myservice.yaml
```

Types of Kubernetes Services

ClusterIP

ClusterIP is the default service type in Kubernetes. It receives a cluster-internal IP address, making its pods only accessible from within the cluster. If necessary, you can set a specific clusterIP in the service manifest, but it must be within the cluster IP range.

Example of a ClusterIp Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP
  clusterIP: 10.10.5.10
  ports:
  —name: http
    protocol: TCP
    port: 80
    targetPort: 8080
```

NodePort

A NodePort service builds on top of the ClusterIP service, exposing it to a port accessible from outside the cluster. If you do not specify a port number, Kubernetes automatically chooses a free port. The kube-proxy component on each node is responsible for listening on the node’s external ports and forwarding client traffic from the NodePort to the ClusterIP.

By default, all nodes in the cluster listen on the service’s NodePort, even if they are not running a pod that matches the service selector. If these nodes receive traffic intended for the service, it is handled by network address translation (NAT) and forwarded to the destination pod.

NodePort can be used to configure an external load balancer to forward network traffic from clients outside the cluster to a specific set of pods. For this to work, you must set a specific port number for the NodePort, and configure the external load balancer to forward traffic to that port on all cluster nodes. You also need to configure health checks in the external load balancer to determine whether a node is running healthy pods.

The nodePort field in the service manifest is optional, and lets you specify a custom port between 30000-32767.

Example of a NodePort Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  —name: http
    protocol: TCP
    port: 80
    targetPort: 8080
    nodePort: 32000
```

LoadBalancer

A LoadBalancer service is based on the NodePort service, and adds the ability to configure external load balancers in public and private clouds. It exposes services running within the cluster by forwarding network traffic to cluster nodes.

The LoadBalancer service type lets you dynamically implement external load balancers. This typically requires an integration running inside the Kubernetes cluster, which performs a watch on LoadBalancer services.

Example of a LoadBalancer Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  clusterIP: 10.0.160.135
  loadBalancerIP: 168.196.90.10
  selector:
    app: nginx
  ports:
  —name: http
    protocol: TCP
    port: 80
    targetPort: 8080
```

ExternalName

An ExternalName service maps the service to a DNS name instead of a selector. You define the name using the spec:externalName parameter. It returns a CNAME record matching the contents of the externalName field (for example, my.service.domain.com), without using a proxy.

This type of service can be used to create services in Kubernetes that represent external components such as databases running outside of Kubernetes. Another use case is allowing a pod in one namespace to communicate with a service in another namespace—the pod can access the ExternalName as a local service.

Example of a External Name Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ExternalName
  externalName: foo.my-service.domain.com
```

Discovering Kubernetes Services

There are two methods by which components in a Kubernetes cluster can discover a service:

- **DNS**—when DNS is enabled, a DNS server is added to a Kubernetes cluster. This server watches for relevant Kubernetes API requests and creates a DNS record for every new service that is created. This allows all pods in the cluster to perform name resolution of services.
- **Environment variables**—the kubelet adds environment variables for all pods running on a node for each active service. This allows pods to access other pods matching a service. However, this method only works if the service was created before the pods using that service (otherwise the required environment variables will not exist).

Troubleshooting Kubernetes Services

Common Problems with a Service

Assume you deployed pods in the cluster and set up a service that is supposed to route traffic to them. If a client attempts to access a pod and fails, this could indicate a number of problems with your service or the underlying pods:

- Service does not exist
- Service does not have the expected DNS name
- DNS is not working in the cluster in general
- Service is not defined correctly
- Service does not map correctly to your pods
- Pods are not working or unstable
- There is a kube-proxy error on your nodes

Run a busybox pod

Here is how to run a BusyBox in your cluster. The open source BusyBox project lets you run many common Linux utilities in one tiny executable:

running a busybox pod

```
$ kubectl run -it --rm --restart=Never busybox --image=gcr.io/google-containers/busybox sh
```

Bash into an existing pod

If you have a running pod that is supposed to be associated with your service, use the following command to bash into a container running on the pod and execute shell commands:

bash into an existing pod

```
$ kubectl exec <pod-name> -c <container-name> -- <commands to execute>
```

Debugging Procedure

The following debugging procedure will help you identify which part of the service communication chain is broken.

In each step, if you encounter an error, stop and fix the problem. If things are working properly, proceed to the next step.

1. Check if the service exists:

   ```
   $ kubectl get svc <service-name>
   ```

2. Try to access the service via DNS name in the same namespace:

   ```
   $ nslookup <service-name>
   ```

3. Try to access the service via DNS name in another namespace:

   ```
   $ nslookup <service-name>.<namespace>
   ```

   If this succeeds, it means you need to change the client application to access the service in another namespace, or run it in the same namespace as the service.
4. Check if DNS works in the cluster:

   ```
   $ nslookup kubernetes.default
   ```

5. Check if the service can be accessed by IP address

   debugging step 5

   ```
   for i in $(seq 1 3); do
       wget -qO- <ip-of-service>
   done
   ```

6. Check if the service is defined correctly - the following are common errors in a service manifest:
   - The service port applications are trying to access is not listed in spec.ports
   - The targetPort defined in the service is different from the port used by the pods
   - The port definition is defined as a string instead of a number
   - For a named port, the name specified in the service is not the same as the port name used by the pods
   - The protocol in the service definition is not the same as the protocol used by the pods

7. Check if labels defined in the service are matching pods

   ```
   $ kubectl get pods -l <label>
   ```

8. Check if the service has any endpoints

   ```
   $ kubectl get endpoints <service-name>
   ```

9. Check i the pods accessed on the endpoint IPs are working

debugging step 9

```
for ep in 10.0.0.1:9376 10.0.0.2:9376 10.0.0.3:9376; do
    wget -qO- $ep
done
```

10. Error with kube-proxy
    in the default implementation of a service, the kube-proxy mechanism that runs on every Kubernetes node is responsible for implementing the service abstraction. If you have gotten this far, the problem could be a malfunction in kube-proxy.

    `See the Kubernetes documentation for more details on debugging kube-proxy issues.`

Reference

https://komodor.com/learn/kubernetes-troubleshooting-the-complete-guide/