KUBERNETES

Concepts

Abstract

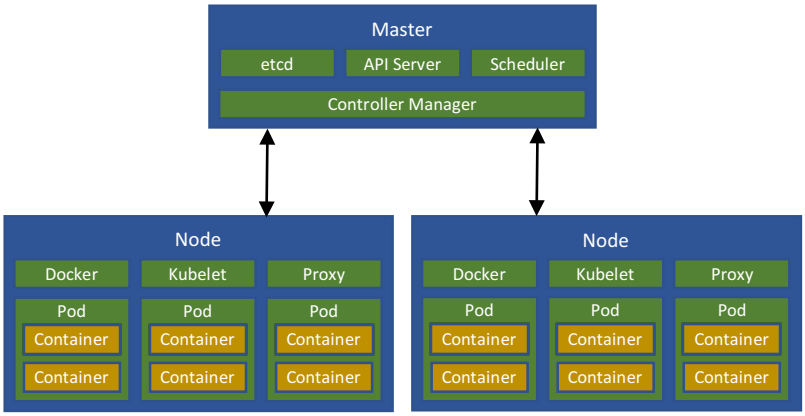
Notes on installation and configuration of Kubernetes
on a 5-node Raspberry 5 cluster.

Jeff Edwards
jeff@edwardsonthe.net

Table of Contents

Kubernetes Concepts

The basic Kubernetes architecture is illustrated in the following diagram:



As illustrated above, the Kubernetes cluster consists of a master node and a number of worker nodes.

Master

The Master node does not run any containers--it just handles and manages the cluster. The Master is the central control point that provides a unified view of the cluster. There is a single Master node that controls multiple worker nodes, which actually run our containers. The Master automatically handles the scheduling of the Pods across the worker nodes in the cluster - by taking into account the available resources on each node. A Master node consists of the following:

- etcd – Kubernetes stores all of its cluster state in etcd. etcd is a distributed, reliable key-value store for the most critical data of a distributed system.
- API Server – Provides the frontend to the cluster's shared state through which all other components interact. The API server is the central management entity and is the only Kubernetes component that connects to etcd. All the other components must go through the API server to work with the cluster state.
- Scheduler – Used by the Master to schedule application instances onto individual nodes in the cluster.
- Controller Manager – Once an application instance is up and running, the Controller Manager will continuously monitor the health of those instances. This is kind of a self-healing mechanism – if a node goes down or is deleted, the Controller Manager replaces it.

Nodes

A node provides an application-specific virtual host in a containerized environment that can run one or more Pods. They execute tasks that have been delegated by the Master node. A node consists of the following:

- Docker – Each node will contain a Docker container runtime, which is responsible for pulling the images and running containers.
- Kubelet – A process that responds to the commands coming from the Master node. Each worker node has a running Kubelet process that listens for tasking requests from the Master node.

- Proxy – A proxy runs on each node in the cluster and has the responsibility for routing network traffic to Pods executing on a node.

Kubernetes Objects

The basic Kubernetes objects include: Pods, Services, Volumes, and Namespaces.

Pods

The Pod consists of one or more Docker containers. This is the basic unit of the Kubernetes platform and an elementary piece of execution that Kubernetes works with. Containers running in the same Pod share the same common network namespace, disk, and security context. This allows containers within a given Pod to communicate with one another over localhost. Each container can also communicate with any other Pod or service within the cluster.

Service

A service is Kubernetes' abstraction to provide network connectivity between one or more Pods. Each service is given its own IP address and port which remains constant for the lifetime of the service. Services have an integrated load-balancer that will distribute network traffic to all Pods.

Volume

A Kubernetes volume has an explicit lifetime – the same as the pod that encloses it. Consequently, a volume outlives any containers that run within the Pod, and data is preserved across Container restarts. When a Pod ceases to exist, the volume will cease to exist, too. Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

Namespace

Namespaces provide a grouping mechanism inside of Kubernetes. Pods, volumes, ReplicaSets, and services can easily cooperate within a namespace, but the namespace provides an isolation from the other parts of the cluster. The use of namespaces provides a mechanism to manage different environments within the same cluster.

Kubernetes Controllers

Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality and convenience features. They include: ReplicaSets, Deployments, StatefulSets, DaemonSets, and Jobs.

ReplicaSets

ReplicaSets provides horizontal scaling of an application. A ReplicaSet ensures that a specified number of Pod clones, known as replicas, are running at any given time. It there are too many, they will be shut down. If there is a need for more, for example some of them died because of an error or crash, or maybe there's a higher load, some more Pods will be brought to life. Replication provides scaling, load balancing, and fault tolerance of our GPF microservices.

Deployment

A Deployment is responsible for creating and updating application instances. Once the Deployment has been created, the Kubernetes Master schedules the application instances onto individual nodes in the cluster. A Deployment is a higher level of abstraction; it manages ReplicaSets when doing Pod orchestration, creation, deletion, and updates. A Deployment provides declarative updates for Pods and ReplicaSets. The Deployment allows for easy updating of a Replica Set as well as the ability to roll back to a previous deployment.

StatefulSet

Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Job

A job creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the job tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. Deleting a Job will clean-up the pods it created.

kubectl

kubectl is a command line interface for running commands against Kubernetes clusters. kubectl provides a lot of functionalities, such as listing resources, showing detailed information about the resources, prints log, managing cluster, and executing commands on a container in a pod.

It is assumed that platform will be providing an abstraction, similar to the baseline controller, of kubectl for managing our cluster.

Kubernetes Best Practices

The following best practices are taken from the Kubernetes user guide:

- When defining configurations, specify the latest stable API version (currently v1).
- Configuration files should be stored in version control before being pushed to the cluster. This allows quick roll-back of a configuration if needed. It also aids with cluster re-creation and restoration if necessary.
- Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.
- Group related objects into a single file whenever it makes sense. One file is often easier to manage than several.
- Many of the kubectl commands can be called on a directory, so you can also call kubectl create on a directory of config files.
- Don’t specify default values unnecessarily – simple and minimal configs will reduce errors.
- Put an object description in an annotation to allow better introspection.
- Avoid creating naked pods – pods that are not bound to a replication controller.
- It’s typically best to create a service before corresponding replication controllers. This lets the scheduler spread the pods that comprise the service.

Creating a Kubernetes Cluster

This section describes the configuration of Kubernetes in a 5-node Raspberry Pi cluster.

Master

`node1` of our Raspberry Pi cluster will be used as the Kubernetes master server. To initialize our Kubernetes master node, we first need to determine the IP address that the master will listen on:

```
$ ip addr show eth0 | fgrep "inet " | sed -r -e 's/.*inet (.*)\/.*/\1/'
192.168.1.84
```

Execute the following command from the command line on `node1`:

```
$ sudo kubeadm init --apiserver-advertise-address=192.168.1.84 --token-ttl=0
.
.
.
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

    mkdir -p $HOME/.kube
    sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
    http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node
as root:

    kubeadm join --token 33d8ec.4f99f37f8db86e8b 192.168.1.84:6443 \
      --discovery-token-ca-cert-hash \
        sha256:995a2037832fc01ce68e60d3fed5491f239c775f781039da2a1dfe1916b283e8
```

As indicated above, execute the following command to configure our regular user to access the cluster:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Nodes

Execute the `kubeadm join` command from the command line on each of the worker nodes.

```
$ ssh -X node2
$ sudo kubeadm join --token 33d8ec.4f99f37f8db86e8b 192.168.1.84:6443 \
  --discovery-token-ca-cert-hash \
    sha256:995a2037832fc01ce68e60d3fed5491f239c775f781039da2a1dfe1916b283e8

$ ssh -X node3
$ sudo kubeadm join --token 33d8ec.4f99f37f8db86e8b 192.168.1.84:6443 \
  --discovery-token-ca-cert-hash \
    sha256:995a2037832fc01ce68e60d3fed5491f239c775f781039da2a1dfe1916b283e8

$ ssh -X node4
$ sudo kubeadm join --token 33d8ec.4f99f37f8db86e8b 192.168.1.84:6443 \
  --discovery-token-ca-cert-hash \
    sha256:995a2037832fc01ce68e60d3fed5491f239c775f781039da2a1dfe1916b283e8

$ ssh -X node5
$ sudo kubeadm join --token 33d8ec.4f99f37f8db86e8b 192.168.1.84:6443 \
  --discovery-token-ca-cert-hash \
    sha256:995a2037832fc01ce68e60d3fed5491f239c775f781039da2a1dfe1916b283e8
```

Apply a Container Network Interface (CNI) Plugin

In our example, Weave Net has been chosen as the CNI plugin to be used in our Kubernetes cluster. Weave Net is responsible for providing a layer 3 IPv4 network between multiple nodes in a cluster. Weave Net does not control how containers are networked to the host, only how the traffic is transported between hosts.

Install the Weave Net plugin on the master – `node1` – using the following command:

```
$kubectl apply -f \
  "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

Validate Cluster

Execute the `kubectl get nodes` command on the master – `node1` – to verify the 4 worker nodes have joined the cluster and have a `Ready STATUS`.

```
$ kubectl get nodes
NAME      STATUS     ROLES     AGE       VERSION
node1     Ready      master    25m       v1.8.4
node2     Ready      <none>    9m        v1.8.4
node3     Ready      <none>    8m        v1.8.4
node4     Ready      <none>    7m        v1.8.4
node5     Ready      <none>    7m        v1.8.4
```

Deploying Spring Boot Applications

A Kubernetes deployment can create, manage, and scale multiple instances of a Spring Boot application packaged in a Docker container image. Deployment of a containerized application into Kubernetes is done using the `kubectl run` command:

```
$ kubectl run <deployment-name> --image=<image> --port=<port>
```

`deployment-name` – The unique name to associate with the Deployment.

`image` – The Docker container image for the container to run. This will include the path to the registry and container image name. Optionally, it can contain a tag representing the container image version to pull. If no tag is provided, Docker will use the `:latest` tag as a default.

`port` – The port that this container exposes.

As an example, I will use a Docker container image of a simple Spring Boot application I have pushed to Docker Hub.

```
$ kubectl run mykube --image=jefferysedwards/mykube --port=8888
```

To view the Deployment, execute the following:

```
$ kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
mykube    1         1         1            1           1m
```

To view the application instances created by the deployment, run this command:

```
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
mykube-695b7458f-tf2rh    1/1       Running   0          6m
```

To view the logs of a Pod, run this command:

```
$ kubectl logs mykube-695b7458f-tf2rh
```

By default, a Pod is only accessible by its internal IP within the cluster. In order to make the mykube container accessible from outside the Kubernetes virtual network, you have to expose the Pod as a Service. To obtain access to a Pod outside of the cluster, use `kubectl expose` command combined with the `--type=LoadBalancer` flag. This flag is required for the creation of an externally accessible IP.

```
$ kubectl expose deployment mykube --type=LoadBalancer
```

The flag used in this command specifies that you'll be using the load-balancer provided by the underlying infrastructure. Note that you expose the Deployment, and not the Pod itself. This will cause the resulting service to load balance traffic across all Pods managed by the Deployment.

The Kubernetes master creates the load balancer and related Compute Engine forwarding rules, target pools, and firewall rules to make the service fully accessible from outside the cluster. To find the publicly-accessible IP address of the Service, simply request kubectl to list all the cluster services:

```
$ kubectl get services
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP    1d
mykube       LoadBalancer   10.103.10.70   <pending>     8888/TCP   8s
```

A powerful feature offered by Kubernetes is the ease in scaling an application. Use the following command to have the replication controller to manage a new number of replicas of an application instance:

```
$ kubectl scale deployment mykube --replicas=3
deployment "mykube" scaled

$ kubectl get deployment
NAME     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
mykube   3         3         3            3           3m
```

Note the declarative approach here – rather than starting or stopping new instances you declare how many instances should be running at all time. Kubernetes reconciliation loops simply make sure the reality matches what you requested and takes action if needed.