RX-M Cloud Native Consulting

Kubernetes

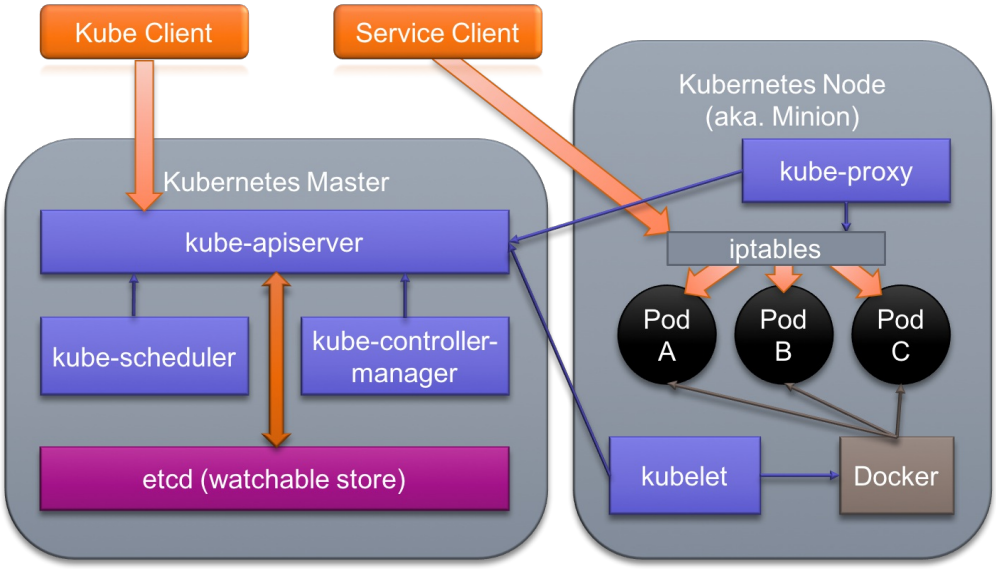
Lab 1 – Kubernetes Local Setup

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts. Kubernetes seeks to foster an ecosystem of components and tools that relieve the burden of running applications in public and private clouds and can run on a range of platforms, from your laptop, to VMs on a cloud provider, to racks of bare metal servers.

The effort required to set up a cluster varies from running a single command to installing and configuring individual programs on each node in your cluster. In this lab we will setup the simplest possible single node Kubernetes cluster using KubeAdm. This will help us to gain basic familiarity with Kubernetes and also give us a chance to examine a best practices installation. Our installation has several prerequisites:

- **Linux** – Our lab system vm is preinstalled with Ubuntu 16.04 though most Linux distributions supporting modern container managers will work. Kubernetes is easiest to install on RHEL/Centos 7 and Ubuntu 16.04.
- **Docker** – Kubernetes will work with a variety of container managers but Docker is the most tested and widely deployed (various minimum versions of Docker are required depending on the installation approach you take). The latest Docker version is almost always recommended, though Kubernetes is often not tested with the absolute latest version.
- **etcd** – Kubernetes requires a distributed key/value store to manage discovery and cluster metadata; though Kubernetes was originally designed to make this function pluggable, etcd is the only practical option.
- **Kubernetes** – Kubernetes is a microservice-based system and is composed of several services. The Kubelet handles container operations for a given node, the API server supports the main cluster API, etc.

This lab will walk you through a basic, from scratch, Kubernetes installation, giving you a complete ground up view of Kubernetes acquisition, build, and deployment in a simplified one node setting. The model below illustrates the Kubernetes master and worker node roles, we will run both on a single system.



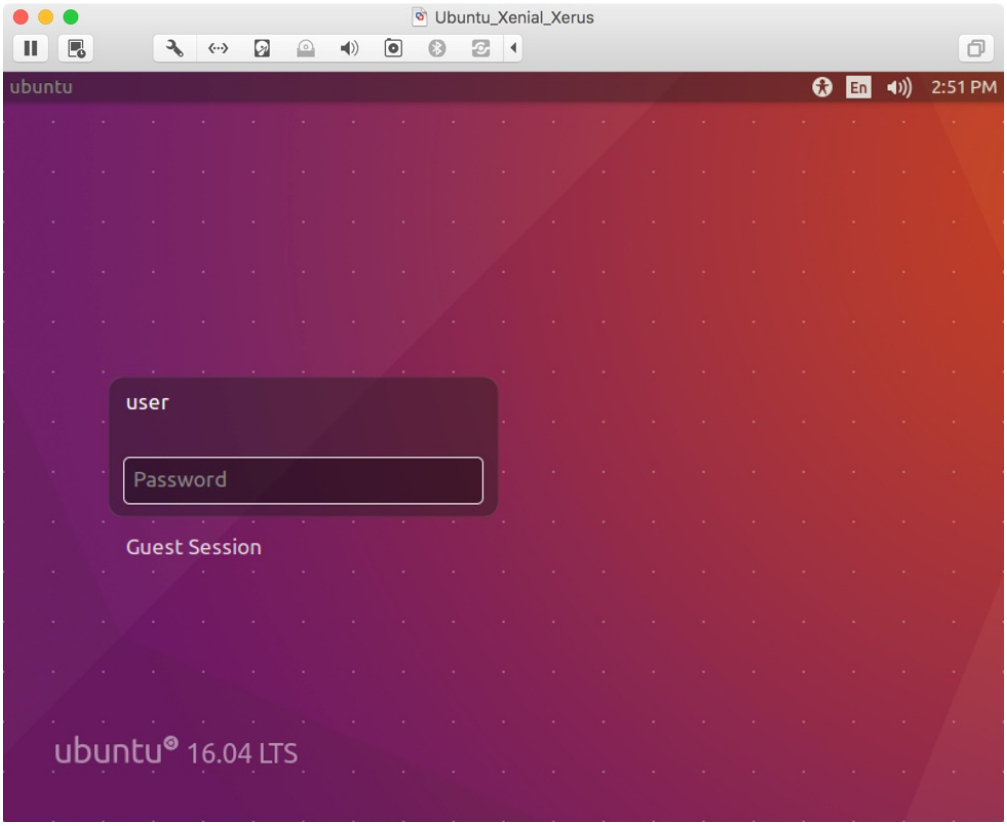
1. Run the Lab VM

The lab exercises for this course are designed for completion on a base Ubuntu 64 bit 16.04 system. The system should have 2+ CPUs, 2+ GBs of RAM and 20+ GB of disk (more memory will allow the machine to run faster so bump it to 4-8 GB ram if you can). Students who have access to such a system (e.g. a typical cloud instance) can perform the lab exercises on that system.

The RX-M preconfigured lab virtual machine is designed to run the labs perfectly. Instructions for downloading and running the free lab VM can be found here:

- https://github.com/RX-M/classfiles/blob/master/lab-setup.md

Login to the VM with the user name "**user**" and the password "**user**".



WARNING

There is no reason to update this lab system and doing so may require large downloads. If the vm prompts you to perform a system update, choose the "Remind me later" option to avoid tying up the class room Internet connection.

2. Install Docker Support

The Launch Pad on the left side of the desktop has short cuts to commonly used programs. Click the Terminal icon to launch a new Bash command line shell.

Docker supplies installation instructions for many platforms. The Ubuntu 16.04 installation guide for the enterprise edition (Docker EE) and the community edition (Docker CE, which we will be using) can be found online here:

- https://docs.docker.com/install/linux/docker-ce/ubuntu/

Docker requires a 64-bit system with a Linux kernel having version 3.10 or newer. Use the `uname` command to check the version of your kernel:

```
user@ubuntu:~$ uname -r

4.4.0-31-generic

user@ubuntu:~$
```

We will need to install some packages to allow apt to use the Docker repository over HTTPS.

> If you receive the error: "E: Unable to lock the administration directory (/var/lib/dpkg/), is another process using it?", your system is probably performing apt initialization in the background. If you wait a minute or two for all of the processes running apt to exit, you should then be able to perform the installation.

Docker provides a convenience script to install the latest version of Docker. It is not recommended to use in a production environment, however, it works great for using Docker in testing scenarios and labs.

```
user@ubuntu:~$ wget -qO- https://get.docker.com/ | sh

# Executing docker install script, commit: f45d7c11389849ff46a6b4d94e0dd1ffebca32c1
+ sudo -E sh -c apt-get update -qq >/dev/null
+ sudo -E sh -c DEBIAN_FRONTEND=noninteractive apt-get install -y -qq apt-transport-https ca-
certificates curl >/dev/null
+ sudo -E sh -c curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | apt-key add -qq -
>/dev/null
+ sudo -E sh -c echo "deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial stable" >
/etc/apt/sources.list.d/docker.list
+ sudo -E sh -c apt-get update -qq >/dev/null
+ [ -n  ]
+ sudo -E sh -c apt-get install -y -qq --no-install-recommends docker-ce >/dev/null
+ sudo -E sh -c docker version
Client: Docker Engine - Community
 Version:           19.03.5
 API version:       1.40
 Go version:        go1.12.12
 Git commit:        633a0ea838
 Built:             Wed Nov 13 07:50:12 2019
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.5
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.12
  Git commit:       633a0ea838
  Built:            Wed Nov 13 07:48:43 2019
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.2.10
  GitCommit:        b34a5c8af56e510852c35414db4c1f4fa6172339
 runc:
  Version:          1.0.0-rc8+dev
  GitCommit:        3e425f80a8c931f88e6d94a8c831b9d5aa481657
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
If you would like to use Docker as a non-root user, you should now consider
adding your user to the "docker" group with something like:

  sudo usermod -aG docker user

Remember that you will have to log out and back in for this to take effect!

WARNING: Adding a user to the "docker" group will grant the ability to run
         containers which can be used to obtain root privileges on the
         docker host.
         Refer to https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
         for more information.

user@ubuntu:~$
```

Take a moment to read the notes from the install script:

```
If you would like to use Docker as a non-root user, you should now consider
adding your user to the "docker" group with something like:

  sudo usermod -aG docker ubuntu

Remember that you will have to log out and back in for this to take effect!

WARNING: Adding a user to the "docker" group will grant the ability to run
         containers which can be used to obtain root privileges on the
         docker host.
         Refer to https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
         for more information.
```

Normal user accounts must use the `sudo` command to run command line tools like `docker`. For our in-class purposes, eliminating the need for `sudo` execution of the Docker command will simplify our practice sessions. To make it possible to connect to the local Docker daemon domain socket without `sudo` we need to add our user id to the *docker* group. To add the *ubuntu* user to the *docker* group execute the following command.

```
user@ubuntu:~$ sudo usermod -aG docker user

user@ubuntu:~$
```

Verify your addition to the Docker group:

```
user@ubuntu:~$ id user

uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),110(sambashare),999
(docker)

user@ubuntu:~$
```

As you can see from the `id user` command, the account “user” is now a member of the “docker” group. Now try running `id` without an account name to display the groups associated with the current shell process:

```
user@ubuntu:~$ id

uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambas
hare)

user@ubuntu:~$
```

Even though the *docker* group was added to your user's group list, your login shell maintains the old groups. After updating your user groups you will need to restart your login shell to ensure the changes take effect.

In the lab system the easiest approach is to logout at the command line:

```
user@ubuntu:~$ kill -9 -1
```

When the login screen returns log back in as *user* with the password *user*.

After logging back in, check to see that your user shell session is now a part of the *docker* group:

```
user@ubuntu:~$ id

uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambas
hare),999(docker)
```

```
user@ubuntu:~$
```

Great. Your current shell user, and any new shell sessions, can now use the `docker` command without elevation.

3. Verify the Installation

Check your Docker client version with the Docker client *--version* switch:

```
user@ubuntu:~$ docker --version

Docker version 19.03.5, build 633a0ea838

user@ubuntu:~$
```

Verify that the Docker server (aka. daemon, aka. engine) is running using the system service interface, on Ubuntu 16.04 systemd. The systemctl command allows us to check the status of services (you can exit the systemctl log output by typing `q`):

```
user@ubuntu:~$ systemctl status --all --full docker

● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: e
   Active: active (running) since Wed 2020-01-08 11:57:32 PST; 1min 51s ago
     Docs: https://docs.docker.com
 Main PID: 64913 (dockerd)
    Tasks: 9
   Memory: 49.3M
      CPU: 171ms
   CGroup: /system.slice/docker.service
           └─64913 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/contai

Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.292184525-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.292191188-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.292282012-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.376271456-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.415529150-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.476594665-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.476847492-08:00
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.476900523-08:00
Jan 08 11:57:32 ubuntu systemd[1]: Started Docker Application Container Engine.
Jan 08 11:57:32 ubuntu dockerd[64913]: time="2020-01-08T11:57:32.486650559-08:00
lines 1-21/21 (END)

q

user@ubuntu:~$
```

Examine the Docker Daemon command line:
`/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock`

By default, the Docker daemon listens on a Unix domain socket for commands `/var/run/docker.sock`. The dockerd `-H` switch adds listening interfaces (Host) to the docker daemon. The `fd://` argument tells docker to listen on the default file descriptor `/var/run/docker.sock`. Other interfaces, such as TCP/IP ports can also be added, dockerd supports multiple -H switches.

As we have seen, this Unix socket is only accessible locally by *root* and the *docker* group. In our lab system the Docker command line client will communicate with the Docker daemon using this domain socket. The Kubelet will communicate with the Docker daemon over this domain socket as well, using the user *root*. By keeping the Docker daemon restricted to listening on the `docker.sock` socket (and not over network interfaces) we limit the security risks associated with access to the Docker daemon.

Also not the second dockerd command line argument: `--containerd=/run/containerd/containerd.sock`

This specifies the low level container engine that Docker will use. Docker is a large system with build tools, SDN networking support and even Swarm built in. Containerd is the low level container engine that implements the OCI (Open Container Initiative) aspects of Docker. Kubernetes can use Containerd directly, making Docker unnecessary in production clusters. However, at present, the full Docker engine is still the most tested and trusted container solution for Kubernetes. Docker also has the advantage of providing a powerful set of command line features useful when debugging and diagnosing problems.

Now check the version of all parts of the Docker platform with the `docker version` subcommand.

```
user@ubuntu:~$ docker version

Client: Docker Engine - Community
 Version:           19.03.5
 API version:       1.40
 Go version:        go1.12.12
 Git commit:        633a0ea838
 Built:             Wed Nov 13 07:50:12 2019
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.5
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.12
  Git commit:       633a0ea838
  Built:            Wed Nov 13 07:48:43 2019
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.2.10
  GitCommit:        b34a5c8af56e510852c35414db4c1f4fa6172339
 runc:
  Version:          1.0.0-rc8+dev
  GitCommit:        3e425f80a8c931f88e6d94a8c831b9d5aa481657
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683

user@ubuntu:~$
```

The client version information is listed first followed by the server version information. You can also use the Docker client to retrieve basic platform information from the Docker daemon.

```
user@ubuntu:~$ docker system info

Client:
 Debug Mode: false

 Client:
  Debug Mode: false

 Server:
  Containers: 0
   Running: 0
   Paused: 0
   Stopped: 0
  Images: 0
  Server Version: 19.03.5
  Storage Driver: overlay2
   Backing Filesystem: extfs
   Supports d_type: true
   Native Overlay Diff: false
  Logging Driver: json-file
  Cgroup Driver: cgroupfs
  Plugins:
   Volume: local
   Network: bridge host ipvlan macvlan null overlay
   Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
  Swarm: inactive
  Runtimes: runc
  Default Runtime: runc
  Init Binary: docker-init
  containerd version: b34a5c8af56e510852c35414db4c1f4fa6172339
  runc version: 3e425f80a8c931f88e6d94a8c831b9d5aa481657
```

```
  init version: fec3683
  Security Options:
   apparmor
   seccomp
    Profile: default
  Kernel Version: 4.4.0-31-generic
  Operating System: Ubuntu 16.04.1 LTS
  OSType: linux
  Architecture: x86_64
  CPUs: 2
  Total Memory: 1.937GiB
  Name: ubuntu
  ID: VVIO:UWRM:GEQD:PGNM:J7TI:22JY:OAZO:KMLY:CJ2R:Z7WX:XCA4:Q7XI
  Docker Root Dir: /var/lib/docker
  Debug Mode: false
  Registry: https://index.docker.io/v1/
  Labels:
  Experimental: false
  Insecure Registries:
   127.0.0.0/8
  Live Restore Enabled: false

 WARNING: No swap limit support

user@ubuntu:~$
```

- What version of Docker is your "docker" command line client?
- How many containers are running on the server?
- What version of Docker is your Docker Engine?
- What is the Logging Driver in use by your Docker Engine?
- What is the Storage Driver in use by your Docker Engine?
- Does the word "Driver" make you think that these component can be substituted?
- Is the server in debug mode?
- What runtime is in use by your system?
- What is the Docker Engine root directory?

4. Run a Container

It's time to run our first container. Use the `docker run` command to run the *rxmllc/hello* image:

```
user@ubuntu:~$ docker container run rxmllc/hello

Unable to find image 'rxmllc/hello:latest' locally
latest: Pulling from rxmllc/hello
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
5c9b4d01f863: Pull complete
Digest: sha256:0067dc15bd7573070d5590a0324c3b4e56c5238032023e962e38675443e6a7cb
Status: Downloaded newer image for rxmllc/hello:latest
 _____________________________
/ RX-M - Cloud Native Consulting! \
\ rx-m.com                       /
 -------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

user@ubuntu:~$
```

You now have a working Docker installation!

- Did the Docker Engine run the container from an image it found locally?
- In the `docker run` output the image is listed with a suffix, what is the suffix?
- What do you think this suffix represents?

Docker has grown to become a powerful and feature rich container manager. The good news is that when you use Docker as your Kubernetes backend you lose none of the Docker functionality. You can always drop down to the command line and use the Docker command line tool to debug and diagnose container problems.

5. Install Kubernetes Package Support

With Linux running and Docker installed we can move on to setting up Kubernetes. Kubernetes packages are distributed in DEB and RPM formats. We will use the DEB based APT repository here: apt.kubernetes.io.

First we need to address a few Kubernetes installation prerequisites.

Swap (vs memory limits)

As of K8s 1.8, the kubelet fails if swap is enabled on a node. The 1.8 release notes suggest:

> To override the default and run with /proc/swaps on, set --fail-swap-on=false

However, for our purposes we can simply turn off swap:

```
user@ubuntu:~$ sudo cat /proc/swaps

Filename                                Type              Size    Used  Priority
/dev/sda5       partition       2094076 5000        -1

user@ubuntu:~$ sudo swapoff -a

user@ubuntu:~$ sudo cat /proc/swaps

Filename                                Type            Size    Used    Priority

user@ubuntu:~$
```

On boot Linux consults the /etc/fstab to determine which, if any, swap volumes to configure. We need to disable swap in the fstab to ensure swap is not reenabled after the system reboots. Comment out the swap volume entry in the file system table file, fstab:

```
user@ubuntu:~$ sudo nano /etc/fstab && cat /etc/fstab

# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda1 during installation
UUID=ae4d6013-3015-4619-a301-77a55030c060 /               ext4    errors=remount-ro 0       1
# swap was on /dev/sda5 during installation
# UUID=70f4d3ab-c8a1-48f9-bf47-2e35e4d4275f none            swap    sw              0       0

user@ubuntu:~$
```

If you do not comment out the swap volume (the last line in the example above) the swap will re-enable on reboot and the Kubelet will fail to start and therefore the rest of your cluster will not start either.

kubeadm

Some apt package repos use the aptitude protocol however the Kubernetes packages are served of https so we need to add the apt https transport:

```
user@ubuntu:~$ sudo apt-get update && sudo apt-get install -y apt-transport-https

Hit:1 https://download.docker.com/linux/ubuntu xenial InRelease
Hit:2 http://us.archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://us.archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
Get:4 http://security.ubuntu.com/ubuntu xenial-security InRelease [109 kB]
Get:5 http://us.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
Fetched 325 kB in 0s (346 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
apt-transport-https is already the newest version (1.2.32).
0 upgraded, 0 newly installed, 0 to remove and 259 not upgraded.

user@ubuntu:~$
```

Next add the Google cloud packages repo key so that we can install packages hosted by Google:

```
user@ubuntu:~$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
-

OK

user@ubuntu:~$
```

Now add a repository list file with an entry for Ubuntu Xenial apt.kubernetes.io packages. The following command copies the repo url into the "kubernetes.list" file:

```
user@ubuntu:~$ echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" \
| sudo tee -a /etc/apt/sources.list.d/kubernetes.list

deb http://apt.kubernetes.io/ kubernetes-xenial main

user@ubuntu:~$
```

Update the package indexes to add the Kubernetes packages from apt.kubernetes.io:

```
user@ubuntu:~$ sudo apt-get update

...
Get:6 https://packages.cloud.google.com/apt kubernetes-xenial InRelease [8,993 B]
Get:7 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 Packages [32.2 kB]
Fetched 41.2 kB in 0s (65.2 kB/s)
Reading package lists... Done

user@ubuntu:~$
```

Notice the new packages.cloud.google.com repository above. If you *do not* see it in your terminal output, you must fix the entry in `/etc/apt/sources.list.d/kubernetes.list` before moving on!

Now we can install standard Kubernetes packages.

6. Install Kubernetes with kubeadm

Kubernetes 1.4 added alpha support for the kubeadm tool; as of Kubernetes 1.13 Kubeadm is GA. The `kubeadmn` tool simplifies the process of installing a Kubernetes cluster. To use `kubeadm` we'll also need the `kubectl` cluster CLI tool and the `kubelet` node manager. We'll also install Kubernetes CNI (Container Network Interface) support for multi-host networking.

> Note: Kubeadm offers no cloud provider (AWS/GCP/etc.) integrations (load balancers, etc.). Kops, Kubespray and other tools are often used for K8s installation on cloud systems, however, many of these systems use Kubeadm under the covers!

Use the aptitude package manager to install the needed packages:

```
user@ubuntu:~$ sudo apt-get install -y kubelet=1.16.4-00 kubeadm=1.16.4-00 kubectl=1.16.4-00
kubernetes-cni

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  conntrack cri-tools ebtables socat
The following NEW packages will be installed:
  conntrack cri-tools ebtables kubeadm kubectl kubelet kubernetes-cni socat
0 upgraded, 8 newly installed, 0 to remove and 268 not upgraded.
Need to get 46.6 MB of archives.
After this operation, 269 MB of additional disk space will be used.

...

Setting up kubelet (1.16.4-00) ...
Setting up kubectl (1.16.4-00) ...
Setting up kubeadm (1.16.4-00) ...
Processing triggers for systemd (229-4ubuntu21.21) ...
Processing triggers for ureadahead (0.100.0-19) ...

user@ubuntu:~$
```

7. Install and start the Kubernetes Master Components

Before we use kubeadm take a look at the kubeadm help menu:

```
user@ubuntu:~$ kubeadm help

     ┌──────────────────────────────────────────────────────────┐
     │ KUBEADM                                                  │
     │ Easily bootstrap a secure Kubernetes cluster             │
     │                                                          │
     │ Please give us feedback at:                              │
     │ https://github.com/kubernetes/kubeadm/issues             │
     └──────────────────────────────────────────────────────────┘

Example usage:

    Create a two-machine cluster with one control-plane node
    (which controls the cluster), and one worker node
    (where your workloads, like Pods and Deployments run).

     ┌──────────────────────────────────────────────────────────┐
     │ On the first machine:                                    │
     ├──────────────────────────────────────────────────────────┤
     │ control-plane# kubeadm init                              │
     └──────────────────────────────────────────────────────────┘

     ┌──────────────────────────────────────────────────────────┐
     │ On the second machine:                                   │
     ├──────────────────────────────────────────────────────────┤
     │ worker# kubeadm join <arguments-returned-from-init>      │
     └──────────────────────────────────────────────────────────┘

    You can then repeat the second step on as many other machines as you like.

Usage:
  kubeadm [command]

Available Commands:
  alpha       Kubeadm experimental sub-commands
  completion  Output shell completion code for the specified shell (bash or zsh)
  config      Manage configuration for a kubeadm cluster persisted in a ConfigMap in the cluster
```

```
  help        Help about any command
  init        Run this command in order to set up the Kubernetes control plane
  join        Run this on any machine you wish to join an existing cluster
  reset       Run this to revert any changes made to this host by 'kubeadm init' or 'kubeadm
join'
  token       Manage bootstrap tokens
  upgrade     Upgrade your cluster smoothly to a newer version with this command
  version     Print the version of kubeadm

Flags:
  -h, --help                     help for kubeadm
      --log-file string          If non-empty, use this log file
      --log-file-max-size uint   Defines the maximum size a log file can grow to. Unit is
megabytes. If the value is 0, the maximum file size is unlimited. (default 1800)
      --rootfs string            [EXPERIMENTAL] The path to the 'real' host root filesystem.
      --skip-headers             If true, avoid header prefixes in the log messages
      --skip-log-headers         If true, avoid headers when opening log files
  -v, --v Level                  number for the log level verbosity

Use "kubeadm [command] --help" for more information about a command.

user@ubuntu:~$
```

Check the kubeadm version:

```
user@ubuntu:~$ kubeadm version

kubeadm version: &version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.4",
GitCommit:"224be7bdce5a9dd0c2fd0d46b83865648e2fe0ba", GitTreeState:"clean", BuildDate:"2019-12-
11T12:44:45Z", GoVersion:"go1.12.12", Compiler:"gc", Platform:"linux/amd64"}

user@ubuntu:~$
```

With all of the necessary prerequisites installed we can now use `kubeadm` to initialize a cluster.

NOTE in the output below, this line:
`[apiclient] All control plane components are healthy after 35.506703 seconds` indicates the approximate time it took to get the cluster up and running; this includes time spent downloading Docker images for the control plane components, generating keys, manifests, etc. This example was captured with an uncontended wired connection--yours may take several minutes on slow or shared wifi, *be patient!*.

```
user@ubuntu:~$ sudo kubeadm init --kubernetes-version 1.16.4

[init] Using Kubernetes version: v1.16.4
[preflight] Running pre-flight checks
        [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The
recommended driver is "systemd". Please follow the guide at
https://kubernetes.io/docs/setup/cri/
        [WARNING SystemVerification]: this Docker version is not on the list of validated
versions: 19.03.5. Latest validated version: 18.09
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-
flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Activating the kubelet service
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [ubuntu kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 192.168.228.157]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
```

```
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [ubuntu localhost] and IPs
[192.168.228.157 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [ubuntu localhost] and IPs
[192.168.228.157 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests". This can take up to 4m0s
[apiclient] All control plane components are healthy after 35.506703 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-
system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-1.16" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node ubuntu as control-plane by adding the label "node-
role.kubernetes.io/master=''"
[mark-control-plane] Marking the node ubuntu as control-plane by adding the taints [node-
role.kubernetes.io/master:NoSchedule]
[bootstrap-token] Using token: veieiy.9emnui21ba2pnqld
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for
nodes to get long term certificate credentials
[bootstrap-token] configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[bootstrap-token] configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following 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:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.228.157:6443 --token veieiy.9emnui21ba2pnqld \
    --discovery-token-ca-cert-hash
sha256:2de9f9b14892821990f91babe3ce5e60de7873279ec061945216c0688333358a

user@ubuntu:~$
```

Read through the kubeadm output. You **do not** need to execute the commands suggested yet, we will be discussing and performing them during the rest of this lab.

Note the preflight check warning:
`[WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. ...`

Control groups, or cgroups, are used to constrain resources that are allocated to processes. Using *cgroupfs* alongside *systemd* means

that there will then be two different cgroup managers. A single cgroup manager will simplify the view of what resources are being allocated and will by default have a more consistent view of the available and in-use resources, but using *cgroupfs* will not impact the operations performed in this lab and subsequent labs.

- What file is the kubelet configuration saved in?
- What IP addresses will the CA certificate generated for the API Server authenticate?
- What is the path of the "manifest" folder?
- What is the name of the config-map created to house the kubeadm configuration?
- What "labels" and "taints" were added to the local node ("ubuntu")?
- What essential addons were applied?

The `kubeadm` tool generates an auth token which we can use to add additional nodes to the cluster, and also creates the keys and certificates necessary for TLS between all of the cluster components. The initial master configures itself as a CA and self signs its certificate. All of the PKI/TLS related files can be found in `/etc/kubernetes/pki`.

```
user@ubuntu:~$ ls -l /etc/kubernetes/pki/

total 60
-rw-r--r-- 1 root root 1216 Jan  8 12:14 apiserver.crt
-rw-r--r-- 1 root root 1090 Jan  8 12:14 apiserver-etcd-client.crt
-rw------- 1 root root 1675 Jan  8 12:14 apiserver-etcd-client.key
-rw------- 1 root root 1679 Jan  8 12:14 apiserver.key
-rw-r--r-- 1 root root 1099 Jan  8 12:14 apiserver-kubelet-client.crt
-rw------- 1 root root 1679 Jan  8 12:14 apiserver-kubelet-client.key
-rw-r--r-- 1 root root 1025 Jan  8 12:14 ca.crt
-rw------- 1 root root 1679 Jan  8 12:14 ca.key
drwxr-xr-x 2 root root 4096 Jan  8 12:14 etcd
-rw-r--r-- 1 root root 1038 Jan  8 12:14 front-proxy-ca.crt
-rw------- 1 root root 1679 Jan  8 12:14 front-proxy-ca.key
-rw-r--r-- 1 root root 1058 Jan  8 12:14 front-proxy-client.crt
-rw------- 1 root root 1679 Jan  8 12:14 front-proxy-client.key
-rw------- 1 root root 1679 Jan  8 12:14 sa.key
-rw------- 1 root root  451 Jan  8 12:14 sa.pub

user@ubuntu:~$
```

The `.crt` files are certificates with public keys embedded and the `.key` files are private keys. The `apiserver` files are used by the `kube-apiserver` the ca files are associated with the certificate authority that `kubeadm` created. The front-proxy certs and keys are used to support TLS when using independent API server extension services. The `sa` files are the Service Account keys used to gain root control of the cluster. Clearly all of the files here with a key suffix should be carefully protected.

8. Exploring the Cluster

The `kubeadm` tool launches the `kubelet` on the local system to bootstrap the cluster services. Using the `kubelet`, the kubeadm tool can run the remainder of the Kubernetes services in containers. This is, as they say, eating one's own dog food. Kubernetes is a system promoting the use of microservice architecture and container packaging. Once the `kubelet` is running, the balance of the Kubernetes microservices can be launched via container images.

Display information on the `kubelet` process:

```
user@ubuntu:~$ ps -fwwp $(pidof kubelet) | sed -e 's/--/\n--/g'

UID          PID    PPID  C STIME TTY          TIME CMD
root       68925       1  1 12:15 ?        00:00:02 /usr/bin/kubelet
--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
--kubeconfig=/etc/kubernetes/kubelet.conf
--config=/var/lib/kubelet/config.yaml
--cgroup-driver=cgroupfs
--network-plugin=cni
--pod-infra-container-image=k8s.gcr.io/pause:3.1

user@ubuntu:~$
```

The switches used to launch the `kubelet` include:

- `--bootstrap-kubeconfig` - kubeconfig file that will be used to set the client certificate for kubelet
- `--kubeconfig` - a config file containing the kube-apiserver address and keys to authenticate with
- `--config` - sets the location of the kubelet's config file, detailing various runtime parameters for the kubelet
- `--cgroup-driver` - sets the container runtime interface. Defaults to cgroupfs (the same as docker)
- `--network-plugin` - sets the network plugin interface to be used
- `--pod-infra-container-image` - the image used to anchor pod namspaces

The package manager configured the `kubelet` as a systemd service when we installed it with apt-get. It is "enabled", so it will restart automatically when the system reboots. Examine the `kubelet` service configuration:

```
user@ubuntu:~$ systemctl --full --no-pager status kubelet

● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2020-01-08 12:15:03 PST; 4min 4s ago
     Docs: https://kubernetes.io/docs/home/
 Main PID: 68925 (kubelet)
    Tasks: 16
   Memory: 43.6M
      CPU: 3.177s
   CGroup: /system.slice/kubelet.service
           └─68925 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --
cgroup-driver=cgroupfs --network-plugin=cni --pod-infra-container-image=k8s.gcr.io/pause:3.1

Jan 08 12:18:44 ubuntu kubelet[68925]: W0108 12:18:44.097946   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:18:45 ubuntu kubelet[68925]: E0108 12:18:45.011007   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
Jan 08 12:18:49 ubuntu kubelet[68925]: W0108 12:18:49.098631   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:18:50 ubuntu kubelet[68925]: E0108 12:18:50.020777   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
Jan 08 12:18:54 ubuntu kubelet[68925]: W0108 12:18:54.098956   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:18:55 ubuntu kubelet[68925]: E0108 12:18:55.029878   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
Jan 08 12:18:59 ubuntu kubelet[68925]: W0108 12:18:59.099782   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:19:00 ubuntu kubelet[68925]: E0108 12:19:00.040591   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
Jan 08 12:19:04 ubuntu kubelet[68925]: W0108 12:19:04.100385   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:19:05 ubuntu kubelet[68925]: E0108 12:19:05.049800   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized

user@ubuntu:~$
```

As you can see from the "Loaded" line the service is enabled, indicating it will start on system boot.

Take a moment to review the systemd service start up files. First the service file:

```
user@ubuntu:~$ sudo cat /lib/systemd/system/kubelet.service

[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=https://kubernetes.io/docs/home/

[Service]
ExecStart=/usr/bin/kubelet
```

```
Restart=always
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target

user@ubuntu:~$
```

This tells systemd to start the service (/usr/bin/kubelet) and restart it after 10 seconds if it crashes.

Now look over the configuration files in the service.d directory:

```
user@ubuntu:~$ sudo ls /etc/systemd/system/kubelet.service.d

10-kubeadm.conf

user@ubuntu:~$
```

Files in this directory are processed in lexical order. The numeric prefix ("10") makes it easy to order the files. Display the one config file:

```
user@ubuntu:~$ sudo cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf

# Note: This dropin only works with kubeadm and kubelet v1.11+
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# This is a file that "kubeadm init" and "kubeadm join" generates at runtime, populating the
KUBELET_KUBEADM_ARGS variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last resort.
Preferably, the user should use
# the .NodeRegistration.KubeletExtraArgs object in the configuration files instead.
KUBELET_EXTRA_ARGS should be sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS

user@ubuntu:~$
```

Let's take a look at the kubelet's configuration, which in the above output is found as:
`Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml`

```
user@ubuntu:~$ sudo cat /var/lib/kubelet/config.yaml

address: 0.0.0.0
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 2m0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt

    ...

user@ubuntu:~$
```

Within the configuration yaml file, lets search for any settings that relate to a storage path.

```
user@ubuntu:~$ sudo cat /var/lib/kubelet/config.yaml | grep Path

staticPodPath: /etc/kubernetes/manifests

user@ubuntu:~$
```

This directory is created during the *kubeadm init* process. Once created, kubelet will monitor that directory for any pods that the kubelet will need to run at startup. Let's list its contents:

```
user@ubuntu:~$ ls -l /etc/kubernetes/manifests/

total 16
-rw------- 1 root root 1765 Jan  8 12:14 etcd.yaml
-rw------- 1 root root 3276 Jan  8 12:14 kube-apiserver.yaml
-rw------- 1 root root 2824 Jan  8 12:14 kube-controller-manager.yaml
-rw------- 1 root root 1119 Jan  8 12:14 kube-scheduler.yaml

user@ubuntu:~$
```

Each of these files specifies a pod description for each key component of our cluster's master node:

- The `etcd` component is the key/value store housing our cluster's state.
- The `kube-apiserver` is the service implementing the Kubernetes API endpoints.
- The `kube-scheduler` selects nodes for new pods to run on.
- The `kube-controller-manager` ensures that the correct number of pods are running.

These YAML files tell the `kubelet` to run the associated cluster components in their own pods with the necessary settings and container images. Display the images used on your system:

```
user@ubuntu:~$ sudo grep image /etc/kubernetes/manifests/*.yaml

/etc/kubernetes/manifests/etcd.yaml:    image: k8s.gcr.io/etcd:3.3.15-0
/etc/kubernetes/manifests/etcd.yaml:    imagePullPolicy: IfNotPresent
/etc/kubernetes/manifests/kube-apiserver.yaml:    image: k8s.gcr.io/kube-apiserver:v1.16.4
/etc/kubernetes/manifests/kube-apiserver.yaml:    imagePullPolicy: IfNotPresent
/etc/kubernetes/manifests/kube-controller-manager.yaml:    image: k8s.gcr.io/kube-controller-
manager:v1.16.4
/etc/kubernetes/manifests/kube-controller-manager.yaml:    imagePullPolicy: IfNotPresent
/etc/kubernetes/manifests/kube-scheduler.yaml:    image: k8s.gcr.io/kube-scheduler:v1.16.4
/etc/kubernetes/manifests/kube-scheduler.yaml:    imagePullPolicy: IfNotPresent

user@ubuntu:~$
```

In the example above, etcd v3.3.10 and Kubernetes 1.15 are in use. All of the images are dynamically pulled by Docker from the k8s.gcr.io registry server using the "google_containers" public namespace.

List the containers running under Docker:

```
user@ubuntu:~$ docker container ls --format "{{.Command}}" --no-trunc | awk -F"--" '{print $1}'

"/usr/local/bin/kube-proxy
"/pause"
"etcd
"kube-apiserver
"kube-controller-manager
"kube-scheduler
"/pause"
"/pause"
"/pause"
"/pause"

user@ubuntu:~$
```

We will discuss the `pause` containers later.

Several Kubernetes services are running:

- kube-proxy - Modifies the system iptables to support the service routing mesh (runs on all nodes)
- etcd - The key/value store used to hold Kubernetes cluster state
- kube-scheduler - The Kubernetes pod scheduler
- kube-controller-manager - The Kubernetes replica manager
- kube-apiserver - The Kubernetes api server

The `kube-proxy` service addon is included by `kubeadm`.

Configure kubectl

The command line tool used to interact with our Kubernetes cluster is `kubectl`. While you can use `curl` and other programs to communicate with Kubernetes at the API level, the `kubectl` command makes interacting with the cluster from the command line easy, packaging up your requests and making the API calls for you.

Run the `kubectl config view` subcommand to display the current client configuration.

```
user@ubuntu:~$ kubectl config view

apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []

user@ubuntu:~$
```

As you can see the only value we have configured is the apiVersion which is set to v1, the current Kubernetes API version. The `kubectl` command tries to reach the API server on port 8080 via the localhost loopback without TLS by default.

Kubeadm establishes a config file during deployment of the control plane and places it in /etc/kubernetes as admin.conf. We will take a closer look at this config file in lab 3 but for now follow the steps kubeadm describes in its output, placing it in a new .kube directory under your home directory.

```
user@ubuntu:~$ mkdir -p $HOME/.kube

user@ubuntu:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

user@ubuntu:~$ sudo chown user $HOME/.kube/config

user@ubuntu:~$
```

Verify the kubeconfig we just copied is understood:

```
user@ubuntu:~$ kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://192.168.228.157:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
```

```
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

```
user@ubuntu:~$
```

The default context should be `kubernetes-admin@kubernetes`.

```
user@ubuntu:~$ kubectl config current-context

kubernetes-admin@kubernetes

user@ubuntu:~$
```

If is not already active, activate the kubernetes-admin@kubernetes context:

```
user@ubuntu:~$ kubectl config use-context kubernetes-admin@kubernetes

Switched to context "kubernetes-admin@kubernetes".

user@ubuntu:~$
```

Verify that the new context can access the cluster:

```
user@ubuntu:~$ kubectl get nodes

NAME     STATUS     ROLES    AGE     VERSION
ubuntu   NotReady   master   7m13s   v1.16.4

user@ubuntu:~$
```

You can now use kubectl to gather information about the resources deployed with your Kubernetes cluster, but it looks like not the cluster is ready for operation.

Taints

During the default initialization of the cluster, kubeadm applies labels and taints to the master node so that no workloads will run there. Because we want to run a one node cluster for testing, this will not do.

In Kubernetes terms, the master node is tainted. A taint consists of a *key*, a *value*, and an *effect*. The effect must be *NoSchedule*, *PreferNoSchedule* or *NoExecute*. You can view the taints on your node with the `kubectl` command. Use the `kubectl describe` subcommand to see details for the master node having the host name "ubuntu":

```
user@ubuntu:~$ kubectl describe $(kubectl get node -o name) | grep -i taints

Taints:                 node-role.kubernetes.io/master:NoSchedule

user@ubuntu:~$
```

We will examine the full describe output later but as you can see the master has the "node-role.kubernetes.io/master" taint with the effect "NoSchedule".

This means the `kube-scheduler` can not place pods on this node. To remove this taint we can use the `kubectl taint` subcommand.

NOTE The command below removes ("-") the master taint from all (--all) nodes in the cluster. **Do not forget the trailing `-` following the taint key "master"!** The `-` is what tells Kubernetes to remove the taint!

> We know what you're thinking and we agree, "taint" is an awful name for this feature and a trailing dash with no space is an equally wacky way to remove something.

```
user@ubuntu:~$ kubectl taint nodes --all node-role.kubernetes.io/master-

node/ubuntu untainted

user@ubuntu:~$
```

Check again to see if the taint was removed:

```
user@ubuntu:~$ kubectl describe $(kubectl get node -o name) | grep -i taints

Taints:             node.kubernetes.io/not-ready:NoSchedule

user@ubuntu:~$
```

We definitely removed the master taint, which presents the master Kubernetes node from running pods. Another one just took its place though, so what happened?

Our first clue is that the taint mentions status `not ready`. Let's grep "ready" status from the node.

```
user@ubuntu:~$ kubectl describe $(kubectl get node -o name) | grep -i ready

Taints:             node.kubernetes.io/not-ready:NoSchedule
  Ready            False   Wed, 08 Jan 2020 12:27:31 -0800   Wed, 08 Jan 2020 12:15:27 -0800
KubeletNotReady              runtime network not ready: NetworkReady=false
reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config
uninitialized

user@ubuntu:~$
```

That's the same warning we observed when inspecting the Kubelet service status: Our Kubernetes cluster does not have any networking in place. Let's fix that!

9. Enable Networking and Related Features

In the previous step, we found out that our master node is hung in the `not ready` status, with Docker reporting that the network plugin is not ready.

Try listing the pods running on the cluster.

```
user@ubuntu:~$ kubectl get pods

No resources found in default namespace.

user@ubuntu:~$
```

Nothing is returned because we are configured to view the "default" cluster namespace. System pods run in the Kubernetes "kube-system" namespace. You can show all namespaces by using the *--all-namspaces* switch.

```
user@ubuntu:~$ kubectl get pods --all-namespaces

NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-5644d7b6d9-b4rnz          0/1     Pending   0          12m
kube-system   coredns-5644d7b6d9-lxdqv          0/1     Pending   0          12m
kube-system   etcd-ubuntu                       1/1     Running   0          11m
kube-system   kube-apiserver-ubuntu             1/1     Running   0          11m
kube-system   kube-controller-manager-ubuntu    1/1     Running   0          11m
kube-system   kube-proxy-npxks                  1/1     Running   0          12m
```

```
kube-system   kube-scheduler-ubuntu            1/1     Running   0          11m

user@ubuntu:~$
```

We've confirmed that no container(s) with DNS in the name are running, though we do see system pods responsible for dns are visible. Notice the STATUS is Pending and 0 of 1 containers are Ready for the coredns pods.

Why is it failing to start? Lets review the POD related events for readiness.

```
user@ubuntu:~$ kubectl get events --namespace=kube-system --sort-by='{.lastTimestamp}'

LAST SEEN   TYPE      REASON              OBJECT                                MESSAGE
14m         Normal    Created             pod/kube-controller-manager-ubuntu   Created container
kube-controller-manager
14m         Normal    Started             pod/kube-scheduler-ubuntu            Started container
kube-scheduler
14m         Normal    Created             pod/kube-scheduler-ubuntu            Created container
kube-scheduler
14m         Normal    Pulled              pod/kube-scheduler-ubuntu            Container image
"k8s.gcr.io/kube-scheduler:v1.16.4" already present on machine
14m         Normal    Started             pod/kube-controller-manager-ubuntu   Started container
kube-controller-manager
14m         Normal    Pulled              pod/etcd-ubuntu                      Container image
"k8s.gcr.io/etcd:3.3.15-0" already present on machine
14m         Normal    Created             pod/etcd-ubuntu                      Created container
etcd
14m         Normal    Started             pod/etcd-ubuntu                      Started container
etcd
14m         Normal    Pulled              pod/kube-apiserver-ubuntu            Container image
"k8s.gcr.io/kube-apiserver:v1.16.4" already present on machine
14m         Normal    Created             pod/kube-apiserver-ubuntu            Created container
kube-apiserver
14m         Normal    Started             pod/kube-apiserver-ubuntu            Started container
kube-apiserver
14m         Normal    Pulled              pod/kube-controller-manager-ubuntu   Container image
"k8s.gcr.io/kube-controller-manager:v1.16.4" already present on machine
13m         Normal    LeaderElection      endpoints/kube-scheduler             ubuntu_963908d2-
c037-41ac-913e-c3a84cfcb2d0 became leader
13m         Normal    LeaderElection      endpoints/kube-controller-manager    ubuntu_5e23caee-
30e4-471a-bf76-2b472eafda0b became leader
13m         Normal    ScalingReplicaSet   deployment/coredns                   Scaled up replica
set coredns-5644d7b6d9 to 2
13m         Normal    Scheduled           pod/kube-proxy-npxks                 Successfully
assigned kube-system/kube-proxy-npxks to ubuntu
13m         Normal    SuccessfulCreate    daemonset/kube-proxy                 Created pod:
kube-proxy-npxks
13m         Normal    SuccessfulCreate    replicaset/coredns-5644d7b6d9        Created pod:
coredns-5644d7b6d9-b4rnz
13m         Normal    SuccessfulCreate    replicaset/coredns-5644d7b6d9        Created pod:
coredns-5644d7b6d9-lxdqv
13m         Normal    Pulled              pod/kube-proxy-npxks                 Container image
"k8s.gcr.io/kube-proxy:v1.16.4" already present on machine
13m         Normal    Created             pod/kube-proxy-npxks                 Created container
kube-proxy
13m         Normal    Started             pod/kube-proxy-npxks                 Started container
kube-proxy
33s         Warning   FailedScheduling    pod/coredns-5644d7b6d9-lxdqv         0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.
33s         Warning   FailedScheduling    pod/coredns-5644d7b6d9-b4rnz         0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.

user@ubuntu:~$
```

That gives us a hint; we have a node running but why isn't it ready? It turns out that we told Kubernetes we would use CNI for networking but we have not yet supplied a CNI plugin. We can easily add the Weave CNI VXLAN based container networking drivers using a POD spec from the Internet.

The weave-kube path below points to a Kubernetes spec for a DaemonSet, which is a resource that runs on every node in a cluster.

You can review that spec via curl:

```
user@ubuntu:~$ curl -L \
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

apiVersion: v1
kind: List
items:
  - apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: weave-net

...

user@ubuntu:~$
```

You can test the spec without running it using the `--dry-run=true` switch:

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

serviceaccount/weave-net created (dry run)
clusterrole.rbac.authorization.k8s.io/weave-net created (dry run)
clusterrolebinding.rbac.authorization.k8s.io/weave-net created (dry run)
role.rbac.authorization.k8s.io/weave-net created (dry run)
rolebinding.rbac.authorization.k8s.io/weave-net created (dry run)
daemonset.apps/weave-net created (dry run)

user@ubuntu:~$
```

The config file creates several resources:

- The ServiceAccount, ClusterRole, ClusterRoleBinding, Role and Rolebinding configure the role-based access control, or RBAC, permissions for Weave
- The DaemonSet ensures that the weaveworks SDN images are running in a pod on all hosts

Run it for real this time:

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

serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created

user@ubuntu:~$
```

Rerun your `kubectl get pods` subcommand to ensure that all containers in all pods are running (it may take a minute for everything to start):

```
user@ubuntu:~$ kubectl get pods --all-namespaces

NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-5644d7b6d9-b4rnz          0/1     Pending   0          16m
kube-system   coredns-5644d7b6d9-lxdqv          0/1     Pending   0          16m
kube-system   etcd-ubuntu                       1/1     Running   0          15m
kube-system   kube-apiserver-ubuntu             1/1     Running   0          15m
kube-system   kube-controller-manager-ubuntu    1/1     Running   0          15m
kube-system   kube-proxy-npxks                  1/1     Running   0          16m
```

```
kube-system   kube-scheduler-ubuntu            1/1     Running   0          15m
kube-system   weave-net-rvhvk                  1/2     Running   0          17s

user@ubuntu:~$
```

If we check related DNS pod events once more, we see progress!

```
user@ubuntu:~$ kubectl get events --namespace=kube-system --sort-by='{.lastTimestamp}' | grep
dns

16m         Normal    ScalingReplicaSet   deployment/coredns                  Scaled up replica
set coredns-5644d7b6d9 to 2
16m         Normal    SuccessfulCreate    replicaset/coredns-5644d7b6d9       Created pod:
coredns-5644d7b6d9-lxdqv
16m         Normal    SuccessfulCreate    replicaset/coredns-5644d7b6d9       Created pod:
coredns-5644d7b6d9-b4rnz
23s         Warning   FailedScheduling    pod/coredns-5644d7b6d9-b4rnz        0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.
23s         Warning   FailedScheduling    pod/coredns-5644d7b6d9-lxdqv        0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.
20s         Normal    Pulled              pod/coredns-5644d7b6d9-b4rnz        Container image
"k8s.gcr.io/coredns:1.6.2" already present on machine
20s         Normal    Created             pod/coredns-5644d7b6d9-b4rnz        Created container
coredns
20s         Normal    Scheduled           pod/coredns-5644d7b6d9-b4rnz        Successfully
assigned kube-system/coredns-5644d7b6d9-b4rnz to ubuntu
20s         Normal    Scheduled           pod/coredns-5644d7b6d9-lxdqv        Successfully
assigned kube-system/coredns-5644d7b6d9-lxdqv to ubuntu
20s         Normal    Pulled              pod/coredns-5644d7b6d9-lxdqv        Container image
"k8s.gcr.io/coredns:1.6.2" already present on machine
20s         Normal    Created             pod/coredns-5644d7b6d9-lxdqv        Created container
coredns
19s         Normal    Started             pod/coredns-5644d7b6d9-lxdqv        Started container
coredns
19s         Normal    Started             pod/coredns-5644d7b6d9-b4rnz        Started container
coredns

user@ubuntu:~$
```

Lets look at the logs of the DNS related containers. We'll retrieve the names of our dns pods, then grab the logs from one of them.

```
user@ubuntu:~$ DNSPOD=$(kubectl get pods -o name --namespace=kube-system |grep dns |head -1) &&
echo $DNSPOD

pod/coredns-5644d7b6d9-b4rnz

user@ubuntu:~$ kubectl logs --namespace=kube-system $DNSPOD

.:53
2020-01-08T20:32:06.338Z [INFO] plugin/reload: Running configuration MD5 =
f64cb9b977c7dfca58c4fab108535a76
2020-01-08T20:32:06.338Z [INFO] CoreDNS-1.6.2
2020-01-08T20:32:06.338Z [INFO] linux/amd64, go1.12.8, 795a3eb
CoreDNS-1.6.2
linux/amd64, go1.12.8, 795a3eb

user@ubuntu:~$
```

Finally, let's check if there are any more taints on our master node and see if it is finally ready:

```
user@ubuntu:~$ kubectl describe $(kubectl get node -o name) | grep -i taints

Taints:             <none>

user@ubuntu:~$ kubectl get nodes
```

```
NAME     STATUS   ROLES    AGE     VERSION
ubuntu   Ready    master   18m     v1.16.4

user@ubuntu:~$
```

The taint has been cleared from our master node, and Kubernetes will now allow pods to run on our single node cluster.

Congratulations, you have completed the lab!