K8s setup on Debian 11

In this document, we will walk you through the process of setting up your very own Kubernetes[1] cluster on the VMs provided by the University.

Note: While it's very tempting to do this alone, this process is extremely impractical to do thrice; thus, we highly recommend you follow the steps in this document together with everyone in your group. This way, all of you get the experience of setting up a Kubernetes cluster.

Your TA should have emailed the credentials of the VMs (IP address, username and password) to you. To login to your VMs, use the following command:

```
1. ssh <username>@<IP>
```

And then enter your password.

You are now logged in on your VM; every command you run this way, you run on the VM instead of your local machine.

To logout again, use:

```
2. exit
```

Conventions

The rest of this document assumes that you are at least a little familiar with Unix shells on Debian or Ubuntu. If you aren't, one of your team members probably is, and can explain the terms or procedures you don't understand.

That said, this document does try to explain the commands and procedures used as best as possible. Feel free to skip over the explanations if you already know them; but they might help you to understand what exactly is going on.

Any text in `monospace` is something that is a command or that directly relates to a command (like command output).

Additionally, the text is also full of references with background material or sources. Most of them are as footnotes[2], but when referenced explicitly in the text, they can also be given directly as a link.

Installing Docker

[1] https://kubernetes.io/
[2] Like this

Kubernetes is merely an orchestrator for containers; the containers themselves are run in a third-party container engine. In this tutorial, we will install the Docker[3] engine (based on this document).

First, make sure that the dependencies for the installation process are installed:

```
3. sudo apt-get update && sudo apt-get install ca-certificates curl
   gnupg lsb-release
```

Note that apt will probably ask you if you are sure you want to install these packages and their dependencies. You probably are, so type 'y' and then hit enter. To avoid this, we will use the '-y' flag in subsequent commands that do this automatically.

Then, we add Docker's repository to Debian's apt package manager. To do so, use the following two commands:

```
4. sudo install -m 0755 -d /etc/apt/keyrings
5. curl -fsSL https://download.docker.com/linux/debian/gpg | sudo
   gpg --dearmor -o /etc/apt/keyrings/docker.gpg
6. sudo chmod a+r /etc/apt/keyrings/docker.gpg
7. echo \
     "deb [arch="$(dpkg --print-architecture)" signed-
   by=/etc/apt/keyrings/docker.gpg]
   https://download.docker.com/linux/debian \
     "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
     sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

The first command downloads the GPG[4] key for Docker's repository, and the second modifies apt's files to add the repository.

Now, we can update the package list (command 6) and then download the Docker engine and related tools (command 9):

```
8. sudo apt-get update
9. sudo apt-get -y install docker-ce docker-ce-cli containerd.io
```

After running command 6, you should now see the Docker repositories (docker.download.com) being refreshed as well.

If you are curious what you are installing exactly, this is what each of the three packages does[5]:

- **containerd.io:** The actual thing that manages and works with containers. This is also the main part that is used by Kubernetes.
- **docker-ce:** Interface to interact with containerd.io (instead of interacting to it with Kubernetes). Although this is called an interface, it itself runs as a daemon (service) on the node, so cannot be interacted with directly.

---

[3] https://www.docker.com/

[4] Gnu Privacy Guard is an open source protocol for key exchange. Not too relevant to the course per-se but if you're interested, you can find here a good introductory video: https://www.youtube.com/watch?v=DMGIlj7u7Eo

[5] https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

- **docker-ce-cli:** The actual command-like interface to docker-ce that you use when running `docker` commands.

The Docker engine is now installed. However, we would like to do two additional steps before we start to use it: first, we want to auto-start the Docker engine once the VM boots; and secondly, we want to use the docker-related commands without `sudo`-rights.

The first step can be done by running:

```
10.     sudo systemctl enable docker
```

Which "enables" the Docker engine (named `docker`) to start up once the system does.

The second tweak requires two steps:

```
11.     sudo usermod -aG docker "$USER"
12. exit
```

The first command adds your user (`$USER`) to the docker group, which is the group of all users who can use docker without sudo. Then, you need to logout of your VM and login again (`exit`, then run the SSH command) to refresh your permissions.

You can check that you have the correct permissions by running:

```
13.     groups
```

You should now see '`docker`' in the list.

You should repeat this process on every VM you want to add to the Kubernetes cluster (which is probably all of them).

Installing Kubernetes

The next step is installing the Kubernetes software which you will use to run your cluster. This does not make your VMs a cluster, however; in this section, we only install the software which we will use to make a cluster in the next section.

We will follow the "Installing kubeadm, kubelet and kubectl" section from this document.

First, install a few extra dependencies that we didn't install for Docker:

```
14.     sudo apt-get -y install apt-transport-https net-tools
```

Next, we add another repository for Kubernetes as well, like we did for Docker. This can be done by running:

```
15.     sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-
   keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
16.     echo "deb [signed-by=/usr/share/keyrings/kubernetes-
   archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial
   main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Command 15 downloads the GPG key for this repository, and command 16 adds it to apt's database (very similar to the commands for Docker).

With the repository added, install the required packages. For Kubernetes, these are[6]:

- **kubelet:** A daemon (service) running on each node that launches and removes images and containers.
- **kubectl:** The interface to the control plane that can be used to manage pods, services, etc.
- **kubeadm:** Tool that helps setting a cluster up by automatically generating the required configuration files.

We can install them with apt-get again:

```
17.     sudo apt-get update && sudo apt-get -y install kubelet
   kubectl kubeadm
```

Be sure to run `sudo apt-get update` as well, since apt won't download the package lists from the new repository otherwise (resulting in errors like "package kubelet is unknown").

This time, though, we want to notify apt that we don't want it to update the packages automatically. We do this by marking the packages as *hold:*

```
18.     sudo apt-mark hold kubelet kubectl kubeadm
```

We do this because updating a Kubernetes cluster might break all sort of dependencies between nodes, and so we don't want apt to do this at arbitrary moments. Additionally, Kubernetes provides its own tools[7] for upgrading the cluster.

Like for Docker, you have to install Kubernetes on every node that you want added to your cluster.

Setting up your cluster: Control node

We can now finally move to setting up your cluster.

As you know, a Kubernetes cluster exists of two types of nodes: control nodes (part of the control plane) and worker nodes.

This means that you have to divide your VMs into control and worker nodes.

Because you only have three VMs, we suggest you to setup one control node and two worker nodes.

To begin, first SSH into your control node so that all subsequent commands are executed there.

We use kubeadm to setup the control plane for us. To let it do so, run:

```
19.     IP=$(ip -4 -o a | grep -i "ens3" | cut -d ' ' -f 2,7 | cut
   -d '/' -f 1 | awk '{print $2}') sudo kubeadm init --pod-network-
```

[6] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#installing-kubeadm-kubelet-and-kubectl

[7] https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/

```
cidr=192.168.0.0/16 --control-plane-endpoint=$IP --apiserver-
cert-extra-sans=$IP
```

This command might look scary, so we'll break it down for you:

- `IP=$(ip -4 … awk '{print $2}')`: Fetches the IP address of the current node. To fully understand how this works, we recommend you run the part in between the brackets incrementally; first everything up to the first bar, then everything (including the first part) up to the second bar, etc. You should be able to see how each part reduces the output from `ip`-tool until only the IP is left.
- `sudo kubeadm init`: Tells the kubeadm to initialize a new cluster on your control node.
- `--pod-network-cidr`: We tell kubeadm that we would like to use a particular subnet (range of IP addresses) for the pod network. In a Kubernetes cluster, each pod will be assigned its own IP, and this subnet determines how those IPs look like.
- `192.168.0.0/16`: The subnet to use for the pods. The `/16` means that the first 16 bits (the first two numbers, since each number is 8 bits long) are set; thus, all pods will be assigned an IP that starts with '`192.168`'.
- `--control-plane-endpoint`: This tells kubeadm the IP of the control plane. Because we use only one node, it should be the IP of the node you are running this command on.
- `$IP`: This will expand to the IP address we fetched at the start of the command. We thus specify the current node as the control plane endpoint.
- `--apiserver-cert-extra-sans`: This tells kubeadm to add the following string of IP addresses and hostnames to the certificates that Kubernetes generates.
- `$IP`: Once again, this will expand to the IP address we fetched at the start of the command. This time, it specifies that external hosts may connect to this cluster via this IP as well.

The command will run for a while as it sends up the cluster (potentially a few minutes or so). Eventually, you'll see something like:

```
...

You can now join any number of control-plane nodes by copying
certificate authorities and service account keys on each node and then
running the following as root:

  kubeadm join <IP>:6443 --token <token> \

        --discovery-token-ca-cert-hash <hash> \

        --control-plane

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

kubeadm join <IP>:6443 --token <token> \

        --discovery-token-ca-cert-hash <hash>
```

Make sure that you copy the **last** join command (including the backslash and the `--discovery-token-ca-cert-hash` line) and store it somewhere temporarily. This command will be used to join

this cluster from another node as a worker node. The other command (the first one) can be used to extend the cluster with more control nodes; but since you have only one of those, this is not relevant for you. Thus, again, copy only the last one.

Before adding other nodes, we first have to do something on the control node:

20. `sudo echo KUBELET_KUBEADM_ARGS=\"--network-plugin=cni --pod-infra-container-image=k8s.gcr.io/pause:3.2 --node-ip=$(ip -4 -o a | grep -i "ens3" | cut -d ' ' -f 2,7 | cut -d '/' -f 1 | awk '{print $2}')\" | sudo tee /var/lib/kubelet/kubeadm-flags.env`
21. `sudo systemctl restart kubelet.service`

These commands change the arguments that are passed to the kubelet service on the control node. The first (command 20) writes the new commands to the `/var/lib/kubelet/kubeadm-flags.env` file, and the second restarts the kubelet service so it uses those arguments.

But what are the arguments that we are setting here? Once again, we break down the command, but we'll do so in two steps:

- First, we'll look at the part that is the arguments for kubelet: `KUBELET_KUBEADM_ARGS=\"--network-plugin=cni --pod-infra-container-image=k8s.gcr.io/pause:3.2 --node-ip=$(ip -4 -o a | grep -i "ens3" | cut -d ' ' -f 2,7 | cut -d '/' -f 1 | awk '{print $2}')\"`
  - `--network-plugin=cni`: Tells kubelet that it should use a CNI (Container Network Interface)[8] plugin instead of the simple, default network that Kubernetes uses. More on that once we've setup the worker nodes.
  - `--pod-infra-container-image=k8s.gcr.io/pause:3.2`: Specifies the so-called *pause container* to use in each pod[9]. Basically, Kubernetes can run multiple containers per pod by using a container that is always on but does nothing to serve as the 'base' or the pod, and then use that to group other containers. The image specified here is the image that Kubernetes uses for the pause container.
  - `--node-ip=$(ip –4 … awk '{print $2}')`: Tells kubelet the IP-address of the node that its running on. Note that the bit after the dollar and in between the brackets is the same used to get the IP address in the main '`sudo kubeadm init`'-command. Check the explanation of the command to learn how it works.
- The rest of the command is basically there to write the arguments to the file `/var/lib/kubelet/kubeadm-flags.env`, and is a bit of shell magic:
  - `sudo echo`: Echoes whatever follows back to *stdout*[10]. You can use this command (minus the `sudo`) yourself to have the computer repeat a silly message back to you (try '`echo "Hello there"`').

[8] https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/
[9] https://www.ianlewis.org/en/almighty-pause-container
[10] https://www.howtogeek.com/435903/what-are-stdin-stdout-and-stderr-on-linux/

- `|` (the bar after the arguments): *pipes*[11] the stdout of the command before it (the echo command) to *stdin*[12] of the command after it. This pattern of '`echo <text> | …`' is often used to automatically provide input to some program you would otherwise have to manually type.
- `sudo tee /var/lib/kubelet/kubeadm-flags.env`: Writes whatever is passed to it on stdin to the given file.

After you have executed these two commands, it's time to initialize the worker nodes.

Setting up your cluster: Worker nodes

The next few steps will have to be repeated on each of your worker nodes.

First, logout of your control node if you were still logged-in (run `exit`). Then, login on the worker node you'd like to add. You should execute the following commands on that same node.

Now you should retrieve the '`kubeadm join …`' command that you copied from the control node. It will probably look something like this:

```
22.     sudo kubeadm join <IP>:6443 --token <token> \
   --discovery-token-ca-cert-hash <hash>
```

(Pay attention to also copy the backslash and the `--discovery-token-ca-cert-hash` line).

Note that we have prepended this command with `sudo`, whereas the command given to you by Kubernetes is not; you should do this too, or else the command will fail to run (saying something about not having enough permissions).

Tip: If you still forgot to prepend sudo, simply run: '`sudo !!`'; the '`!!`' automatically expands to the previous command.

This command logs in (`join`) on the node with IP-address `<IP>` and on port `6443`. The other two arguments function as a kind of credentials; this prevents anybody with access to that port to just joining the cluster. These will thus also be different every time someone runs `kubeadm init`.

If you have no longer access to the original join command from `kubeadm init`, then we refer you to this page, that explains how you can get the token and the hash needed to re-create it.

Once the command has completed, it should say something along the lines of "This node has joined the cluster". With that done, you can add the same kubelet arguments as you did on the control node:

```
23.     sudo echo KUBELET_KUBEADM_ARGS=\"--network-plugin=cni --
   pod-infra-container-image=k8s.gcr.io/pause:3.2 --node-ip=$(ip -4
   -o a | grep -i "ens3" | cut -d ' ' -f 2,7 | cut -d '/' -f 1 | awk
   '{print $2}')\" | sudo tee /var/lib/kubelet/kubeadm-flags.env
```

[11] https://www.howtogeek.com/435903/what-are-stdin-stdout-and-stderr-on-linux/
[12] https://www.howtogeek.com/435903/what-are-stdin-stdout-and-stderr-on-linux/

24. `sudo systemctl restart kubelet.service`

(Note that, while the command is the same, the arguments technically aren't; the IP will differ depending on which node you run it).

You should repeat this process on every node you intend to add as a worker node.

Setting up your cluster: The conclusion

There are only a few steps left to setting up your cluster.

First, we finally execute the three commands that the `kubeadm init`-command itself recommends to us:

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

These three commands basically copy the default configuration for the kubectl command from the root user (located at `/etc/kubernetes/admin.conf`) to the directory for the current user (`/home/<username>/.kube/config`). Command 25 creates the target directory (`$HOME` expands to your user's directory), command 26 copies the configuration file and command 27 changes the file's owner to you (allowing you access to the file).

To elaborate a little on command 27:

- `sudo chown`: Runs the shell tool to **ch**ange **own**ership (as sudo, because the file currently belongs to the super user).
- `$(id -u):$(id -g)`: Defines the `<user>:<group>` to transfer the ownership to. The correct user ID for your user is obtained by `$(id -u)`, and the ID for your user group is obtained by `$(id -g)` (every user also has a group with the same name; check `groups` again).

The only thing that remains to be done is to select a *networking plugin*[13]*.* This is a third-party plugin that implements the network between the pods according to the CNI (Control Network Interface) specification defined by Kubernetes.

In this tutorial, we'll setup Calico[14], one of the most used and versatile network plugins for Kubernetes.

Installing a network plugin is usually rather straightforward. For most plugins, this is simply applying a configuration YAML[15] in much like the same way you would deploy a service yourself.

---

[13] https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/
[14] https://www.tigera.io/project-calico/
[15] https://www.redhat.com/en/topics/automation/what-is-yaml#:~:text=YAML%20is%20a%20data%20serialization,is%20for%20data%2C%20not%20documents.

For Calico, however, this is slightly more complicated, as it requires *two* YAML configuration files to be applied:

```
kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml
kubectl create -f https://docs.projectcalico.org/manifests/custom-resources.yaml
```

And that's it! Your Kubernetes cluster should now be set up.

If you like, you can see your nodes coming online by running:

```
watch kubectl get nodes
```

The watch-command simply calls the command you give it every two seconds and displays its results.

The same can be done for all the system pods:

```
watch kubectl get -A pods
```

Once those are all online, you should be able to run your own services as is required by the assignment.

Troubleshooting

Unfortunately, setting up a cluster won't always go right. This can be due to entering commands wrong or in the wrong order, or due to external things like network errors.

Please, please, *please* try to find a solution yourself first before asking any of the TAs. Not only does this reduce workload for the TAs, finding solutions yourself is also much more educational and a really important skill to practice.

Good resources on Kubernetes include the many GitHub issues[16] that have been filed over the years or the official Kubernetes documentation[17]. But probably even more efficient than visiting those locations is just to use a search-engine like Google to search across all sites.

That said, there are a few techniques that we list here to help you debug the problem and solve some issues:

Inspecting logs

Like Docker, Kubernetes keeps track of the output of the containers it runs as logs.

To inspect them, first determine the pod name by running:

```
kubectl get pods
```

This returns all of your pods. Alternatively, you can use the `-A` flag to specify you want to see pods from *all* namespaces, including the system namespaces:

[16] https://github.com/kubernetes/kubernetes/issues
[17] https://kubernetes.io/docs/home/

```
33.     kubectl get -A pods
```

Once you know the name of the pod to inspect, run:

```
34.     kubectl logs <pod_name>
```

Note, though, that this command only searches in the default namespaces (usually called `default`). However, as stated before, not all pods that run in your cluster belong to that namespace, and this command will fail to find them. To fix this, use the `-n` or `--namespace` option:

```
35.     kubectl -n <namespace> logs <pod_name>
```

This should then give you the output of the pod since it started running.

Also note that, while Kubernetes' "Ready" flag is very useful, pods can still report that they are ready and then encounter issues (especially your own services). It thus often makes sense to also look in seemingly healthy pods if you cannot find the problem in non-ready ones.

Restarting pods

Sometimes, Kubernetes pods can benefit from a reboot, like any service or computer.

However, Kubernetes does not have native support to do this, and so we have to use the Docker daemon directly to tell the pod to restart.

That means that we have to know on which node the pod is running (since every node has its own daemon). To find out, run:

```
36.     kubectl get pod <pod_name> -o wide
```

And check the `NODE`-column.

Once you know where the pod in question is running, SSH into the node and run the following command to find the specific container:

```
37.     docker ps
```

You will probably see a lot of containers, so you will have to search a bit until you find a container that has a name that contains the name of your pod. Note, though, that you should look for a container that has '`POD`' in it as well, since otherwise you will restart the pause container - which will have little effect.

Once you have found the appropriate container, copy the container's ID, and run:

```
38.     docker restart <ID>
```

And Docker will restart the container in question.

Resetting your cluster

You can relatively easily tear a cluster down and then set it up again. This is useful if you change your mind midway through the setup process or if kubeadm crashed after the preflight-phase.

To reset your cluster, login to each node that is part of your cluster, and run the following commands on each of them:

```
39.     sudo kubeadm reset
```

(You will be prompted if you are sure; type ‘y’, then hit enter)

```
40.     sudo rm -rf /etc/cni /var/lib/cni /opt/cni /var/log/calico
   /var/lib/calico /etc/kubernetes
41.     rm -rf ./.kube
```

The first command instructs kubeadm to remove most of the cluster, and the other two commands remove any configuration files that are left and that are not removed by kubeadm.

Once the files are cleared, you should restart your node (`sudo reboot`) to also clear any rules that Kubernetes may have added to your routing tables.

If you want to be extra thorough, you can also clear all containers and images from the local docker daemons:

```
42.     docker container prune
```

(Enter ‘y’, then hit enter if prompted if you are sure)

```
43.     docker image prune -a
```

(Again, it asks for confirmation, so enter ‘y’ and hit enter)

Once the reboot of all nodes is complete, you can setup your cluster again as specified from “Setting up your cluster: Control node” and onwards.

Downgrading your cluster

Sometimes, the newer Kubernetes versions have bugs. For example, there is one bug that can cause the newer version Kubernetes to fail to setup NodePorts in some situations.

One way to work around those bugs is to downgrade your Kubernetes cluster.

First, tear the cluster down as described in the “Resetting your cluster” section (remember to do so for all nodes!).

Then, on all nodes, remove the existing Kubernetes installation:

```
44.     sudo apt-get remove --purge kubelet kubeadm kubectl
```

Also remove the package’s dependencies to force apt to download the proper versions for those too:

```
45.     sudo apt-get autoremove
```

Then, you can re-install Kubernetes again, but now with the desired versions:

```
46.     sudo apt-get install kubelet=<version> kubeadm=<version>
   kubectl=<version>
```

Note that all versions have to be the same.

To see a list of available versions, you could run:

```
47.     sudo apt-cache madison kubeadm
```

(the same command works for kubelet or kubectl too)

For example, to install version 1.19.6-00:

```
48.     sudo apt-get install kubelet=1.19.6-00 kubeadm=1.19.6-00
   kubectl=1.19.6-00
```

Don't forget to re-mark the packages as hold, to prevent apt from upgrading it:

```
49.     sudo apt-mark hold kubelet kubeadm kubectl
```

Once that is installed, you can go back to "Setting up your cluster: Control node" and follow the tutorial from there onwards.

Freeing space

One final issue you may run into, especially when using the more resource-limited VMs, is that the disk space may quickly get used up by Docker. This is because Docker tends to cache a lot, especially old version of images you rebuilt or old containers.

One of the easiest way to clean up disk space is to run:

```
50.     docker system prune -af
```

(You can omit the `-f` from the command, in which case you will be prompted if you are sure)

This will clean all the unused containers, systems, volumes and other Docker resources. Essentially, it will only leave you with the currently running containers and everything needed to do so.

Another tip when you're running low on disk space is to use the `du`-command, which gives you the **d**isk **u**sage of a particular directory. By using just a little bit of shell-fu, you can get the biggest files and directories on your system:

```
51. sudo du -h / | sort -rh | head -n 10
```

This command will list the disk usage (`du`) of files in the root directory (`/`), using human-readable size identifiers (`-h`; otherwise, you will see the raw number of bytes instead of KB, MB, etc). The result of that is sorted from low-to-high (`sort -r`; the `-h` means it will sort based on the human-readable identifiers, not raw numbers) and then only take the top 10 of that (`head -n 10`). Finally, the `sudo` is only used to also search the contents of root-only directories, and is only necessary if you plan to search the whole filesystem instead of a local directory.

Once you found large files or folders this way, you can get more precise information by changing the '`/`' in the command to be that of a particular folder.