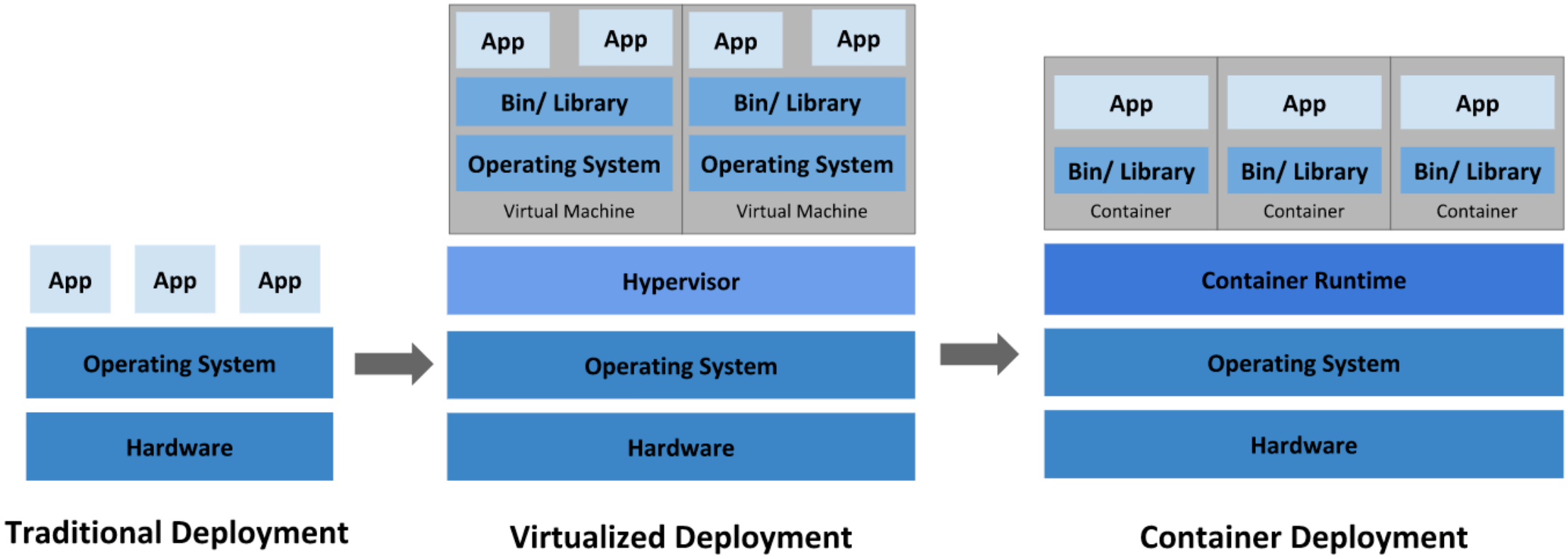
Problem / Overview

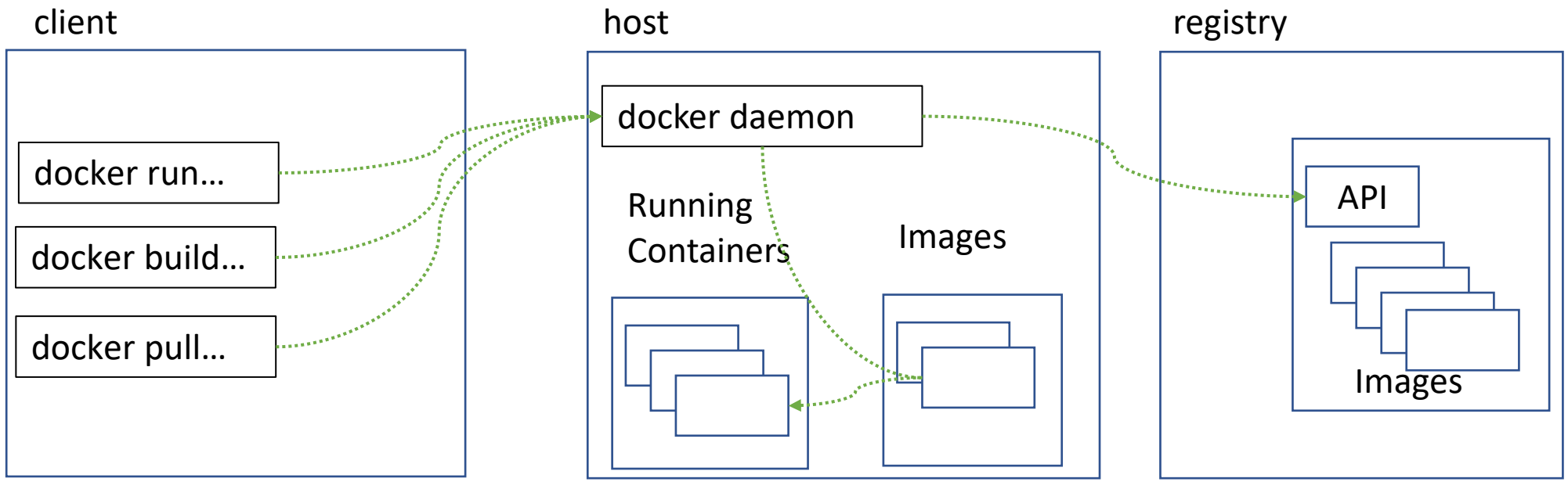
Course: Networking Principles in Practice – Linux Networking
Module: Kubernetes Networking with Linux

University of Colorado **Boulder**

Recall: Containers

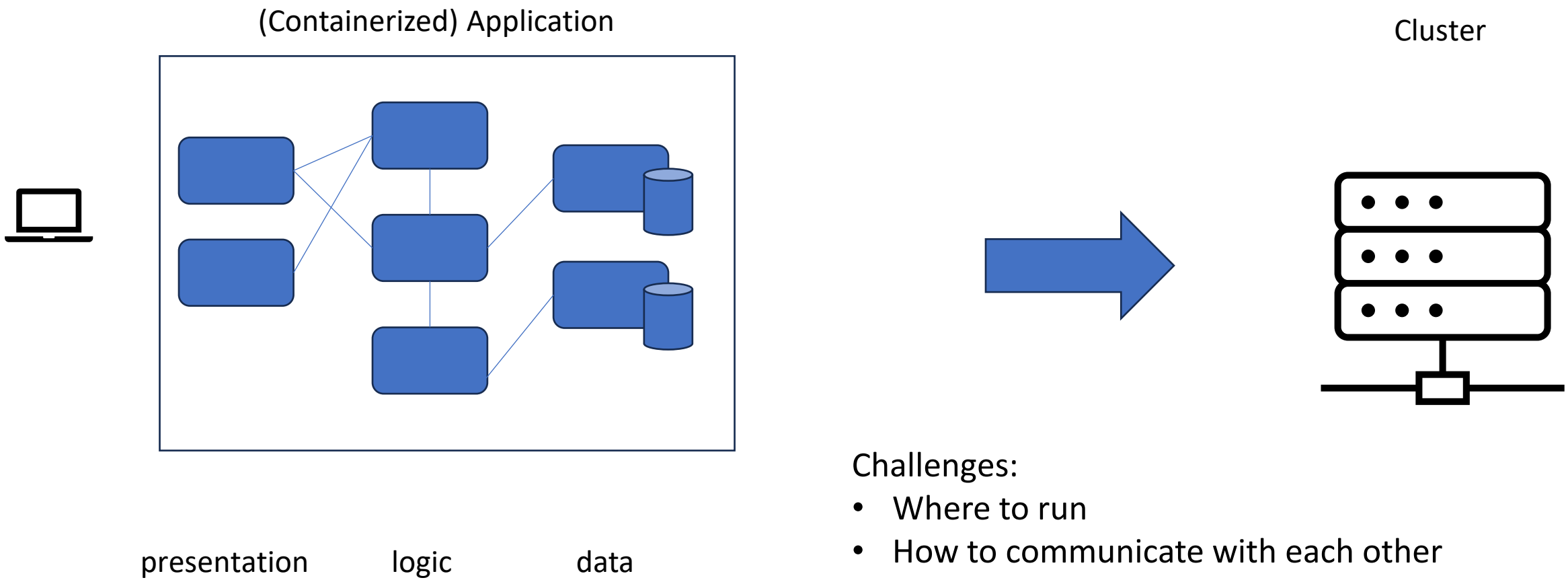


Recall: Docker




```
sudo docker run -ti --rm ubuntu:22.04 /bin/bash
```

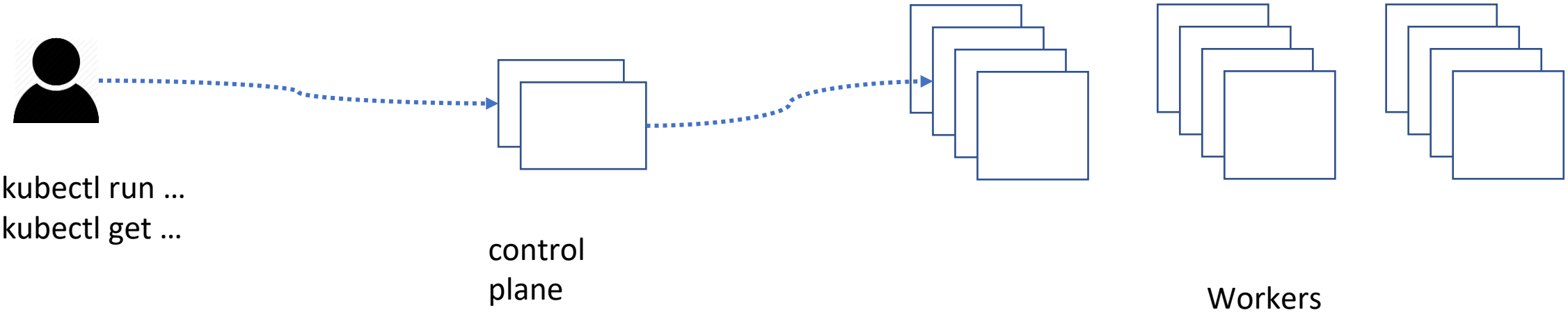
What about Running a Distributed Application



Challenges:

- Where to run
- How to communicate with each other
- Scaling the service
- Handling failure (restarting)

Kubernetes



“Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management.” (Wikipedia)

Challenges:

- Where to run => k8s includes a scheduler to pick nodes
- How to communicate with each other => ensures communication in face of changes
- Scaling the service => has constructs to support scaling
- Handling failure (restarting) => monitors and can restart containers

Outline

- Kubernetes Use
- Kubernetes Architecture
- Kubernetes Networking
- Creating a Kubernetes Network Plugin

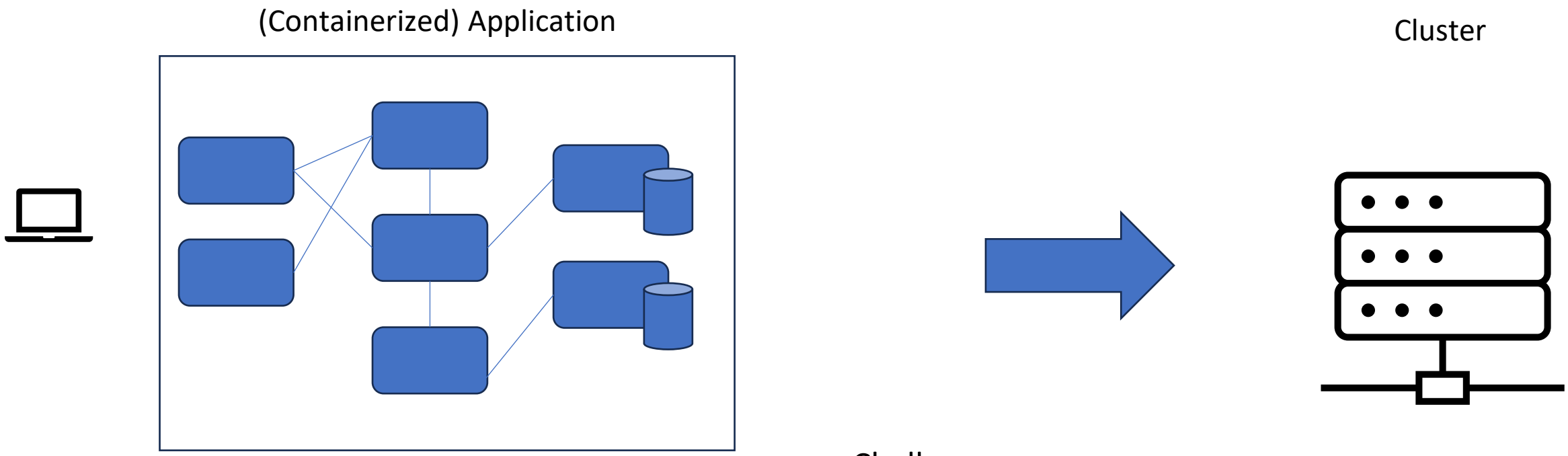
University of Colorado **Boulder**

Kubernetes Use

Course: Networking Principles in Practice – Linux Networking
Module: Kubernetes

University of Colorado **Boulder**

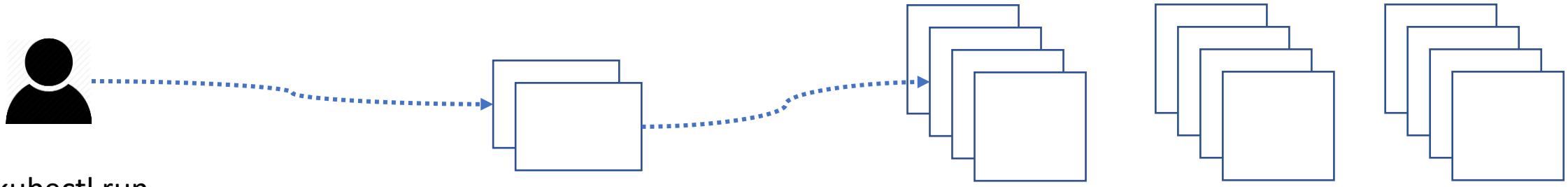
What about Running a Distributed Application



Challenges:

- Where to run
- How to communicate with each other
- Scaling the service
- Handling failure (restarting)

Kubernetes



kubectl run …
kubectl get …

control
plane

Workers

“Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management.” (Wikipedia)

Often abbreviated k8s

Tell k8s the desired state, it will best figure out (1) how to do that, (2) ensure it stays in that state.

Creating a Kubernetes Cluster

- Option 1: Launch VMs and install K8s with kubeadm
  - A Vagrant file and some scripts to do this are provided
  - (https://github.com/techiescamp/vagrant-kubeadm-kubernetes/tree/main)
- Option 2: Single VM and use KinD (Kubernetes in Docker) (https://kind.sigs.k8s.io/)
  - Each node (control plane or worker) is a Docker container
  - A Vagrant file is provided which installs kind
- We'll do everything with kind

Creating a Cluster with Kind

- Show configuration file

kind create cluster --config cluster-configs/1master2worker.yaml

docker ps (note the three containers. You can docker exec into them)

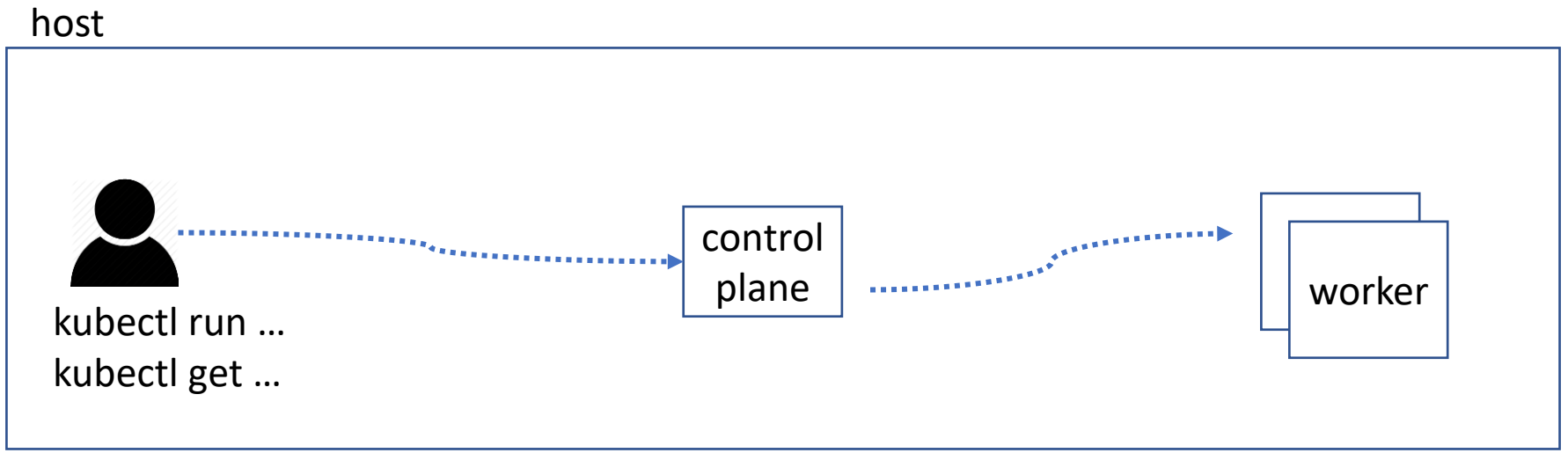
kind delete cluster

Can run multiple clusters. We'll only use one.

https://kind.sigs.k8s.io/docs/user/configuration/

kubectl

- Management utility for controlling cluster
- .kube/config – tells the utility to send API calls to the control plane container
- Try: kubectl get nodes



Kubectl – commands we'll use

- apply
- delete

- get
- describe
- logs

Pod

- A Pod is the unit of work in Kubernetes
- Often a single container, but can be multiple containers
- Receives a unique IP address in the Kubernetes cluster

kubectl apply -f <pod-config.yaml>

kubectl delete --force -f <pod-config.yaml>

Example YAML

Base fields for kubernetes resources:

apiVersion:

kind:

metadata:

spec:

Aside on Yaml

- Configuration as a set of key – value pairs (values can be further key/value pairs)

<key>:value

<key>:

    <value>

<key:>

  - <value[0]>

  - <value[1]>

Example: simple-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mynginx
```

Example: simple-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mynginx
spec:
  containers:
  - name: mynginx
    image: nginx
```

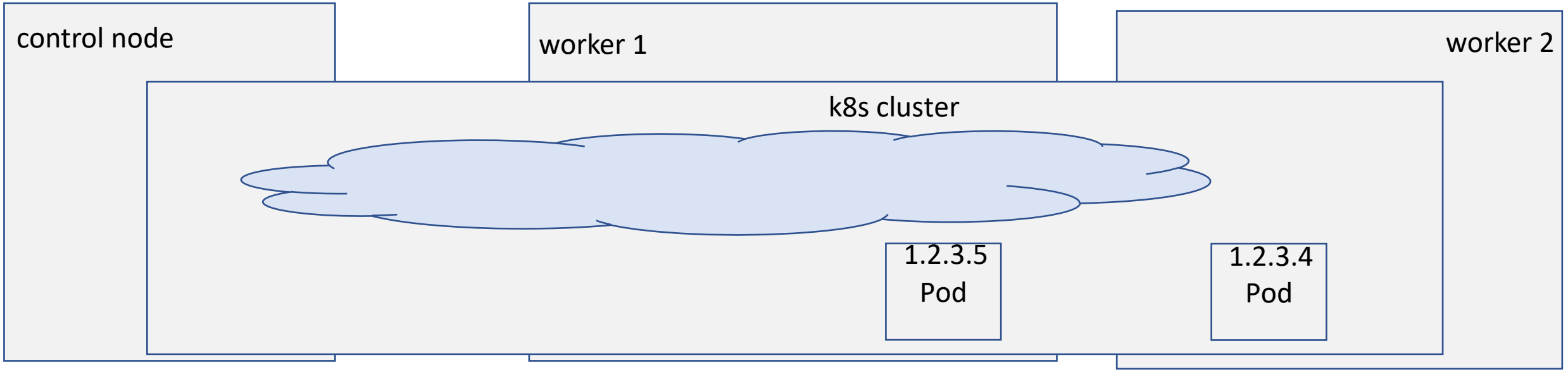
Let’s Try it

kubectl apply -f simple-nginx.yaml

See it running:
  kubectl get pods -o wide
Get its logs:
  kubectl logs mynginx
See more details about it (useful for troubleshooting):
  kubectl describe pod mynginx
Delete it:
  kubectl delete –force pod mynginx

Networking (just enough to test)

- When you run a pod - it gets an IP address
  kubectl get pods -o wide
- That IP address is local to the cluster network



kubectl port-forward

- Port forward from where kubectl is being run to the Kubernetes network (useful for simple testing)

kubectl port-forward pod/mynginx 8000:80

curl 127.0.0.1:8000

Controllers

We just were creating Pod's directly.

```
kubectl apply -f somepod.yaml
```

But, better to not do that, and instead create another resource which then creates and manages pods.

Deployment

- You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
- Note: actually controls another controller (ReplicaSet).
- Uses:
  - Scale # pods up / down
  - Controlled upgrade / downgrade
  - Controlled update of pod spec

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Example YAML

File: simple-nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mynginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Replicas specify how many Pods to ensure are running

Selector says which Pods to apply that condition to

Specification of Pod to create

Aside: Labels / Selectors

- Arbitrary key-value pair you can attach to a resource (e.g., a Pod) and can be used for label selection
- In yaml:

```
metadata:
    labels:
        app: nginx
```

- Or via kubectl

```
kubectl label pods mynginx app=nginx
```

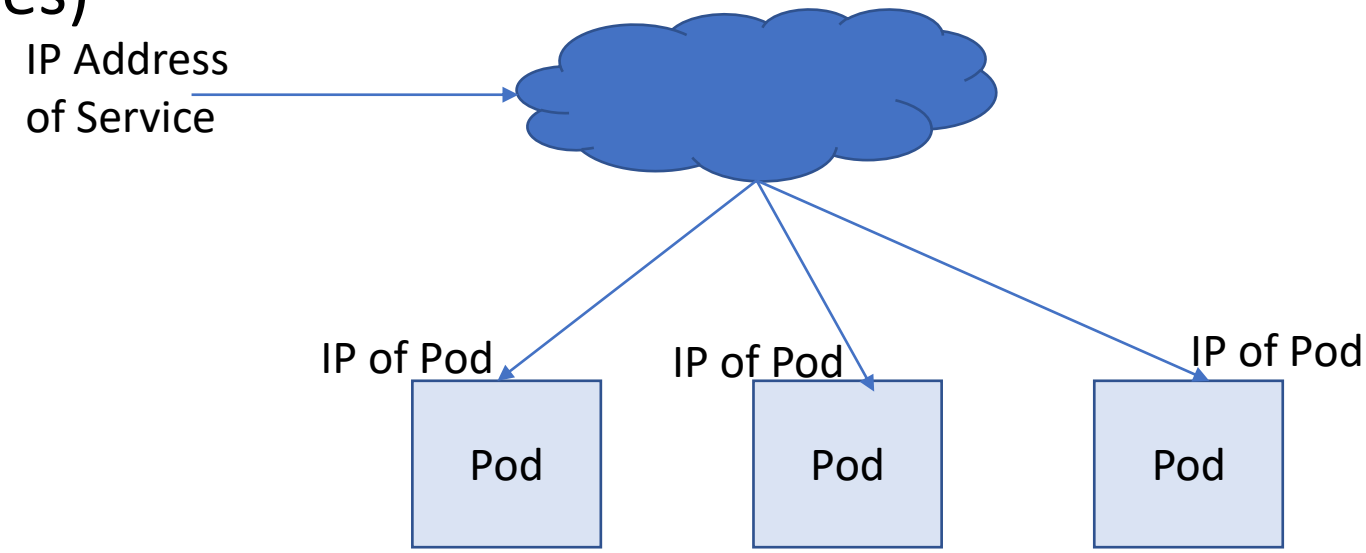
Try It

- kubectl apply -f simple-nginx-deployment.yaml
- kubectl get pods
- kubectl get deployments
- (kubectl get all)

- Change replicas to 4 and apply again
- Change replicas to 2 and apply again

Service

- “A Service is an abstraction which defines a logical set of Pods and a policy by which to access them”
- Load Balance between then
- Uses a selector to see which nodes should be in set (across scaling and failures)



Example YAML

- Selector match label of Pod
  (e.g., from template in Deployment)
- Try it:

kubectl apply –f simple-nginx-service.yaml

kubectl describe service mynginx-service

- Note the End Points – will be IP of the Pods

simple-nginx-service.yaml

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

kubectl port-forward

- Port forward from where kubectl is being run to the Kubernetes network (useful for simple testing)

kubectl port-forward service/mynginx-service 8000:80

curl 127.0.0.1:8000
(will go to one of the 4, round robin between them)

Much more to explore

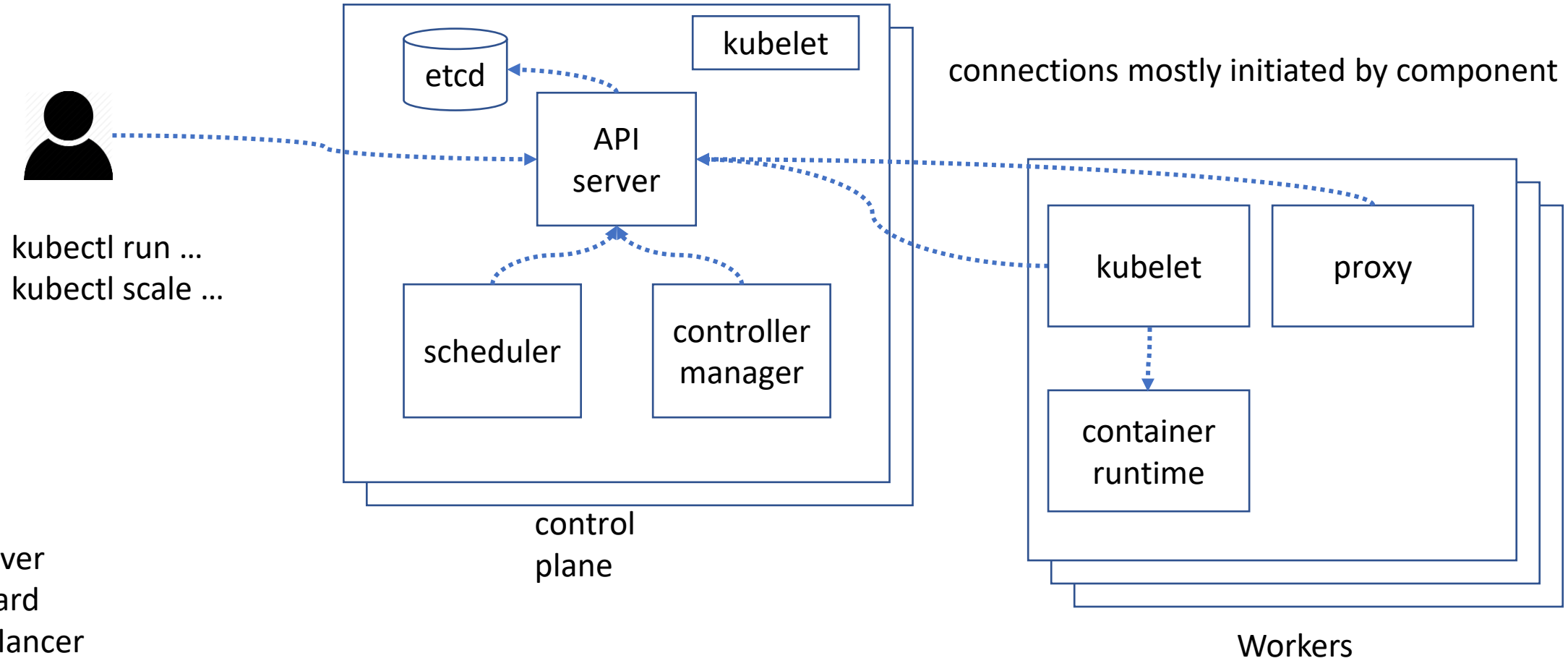
University of Colorado **Boulder**

Kubernetes Architecture

Course: Networking Principles in Practice – Linux Networking
Module: Kubernetes

University of Colorado **Boulder**

High-level Architecture (partial)



Others:

- DNS Server
- Dashboard
- Load balancer
  (Ingress Controller, External Load balancer)
- CNI plugin

How components are run

- Kubelet is the only component that always runs as a regular system component
- Kubelet then runs all other components as pods
  (kubelet runs on the control nodes to launch control plane)

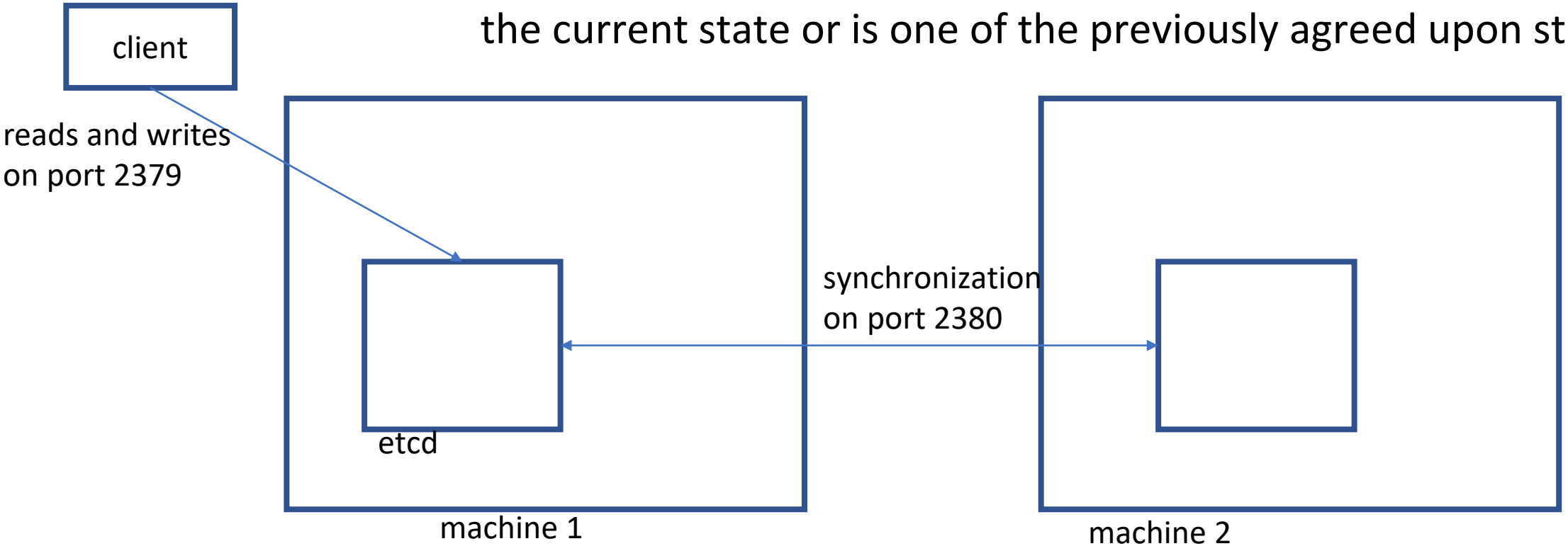
Explore a bit

- docker exec -it <each node> /bin/bash
- Then:
  - ps aux
  - crictl ps

- kubectl get all -A

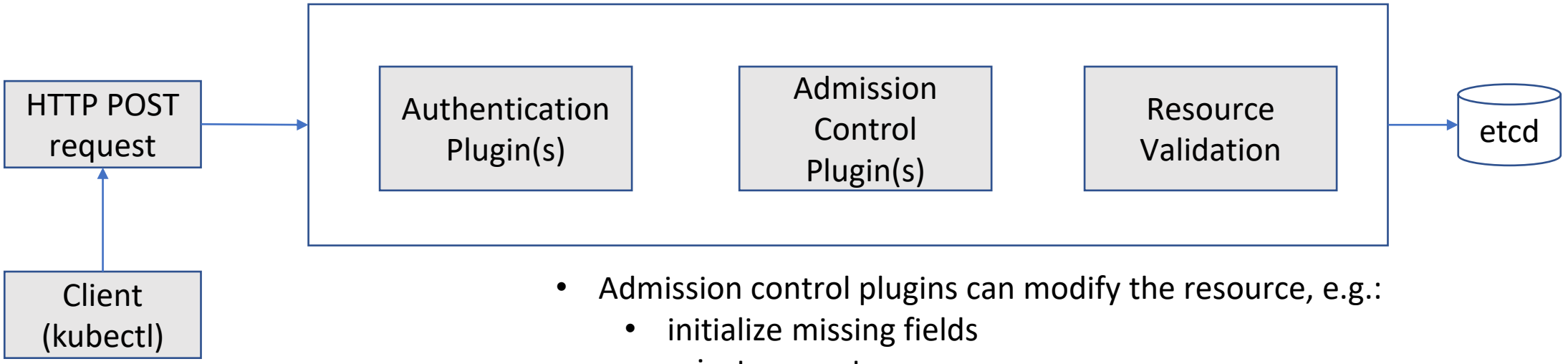
etcd - a distributed key value store

- Stores state for the k8s cluster (info about pods, deployments, etc.)
- Uses RAFT consensus algorithm
  - Each node's state is either what the majority of the nodes agree is the current state or is one of the previously agreed upon states



API Server

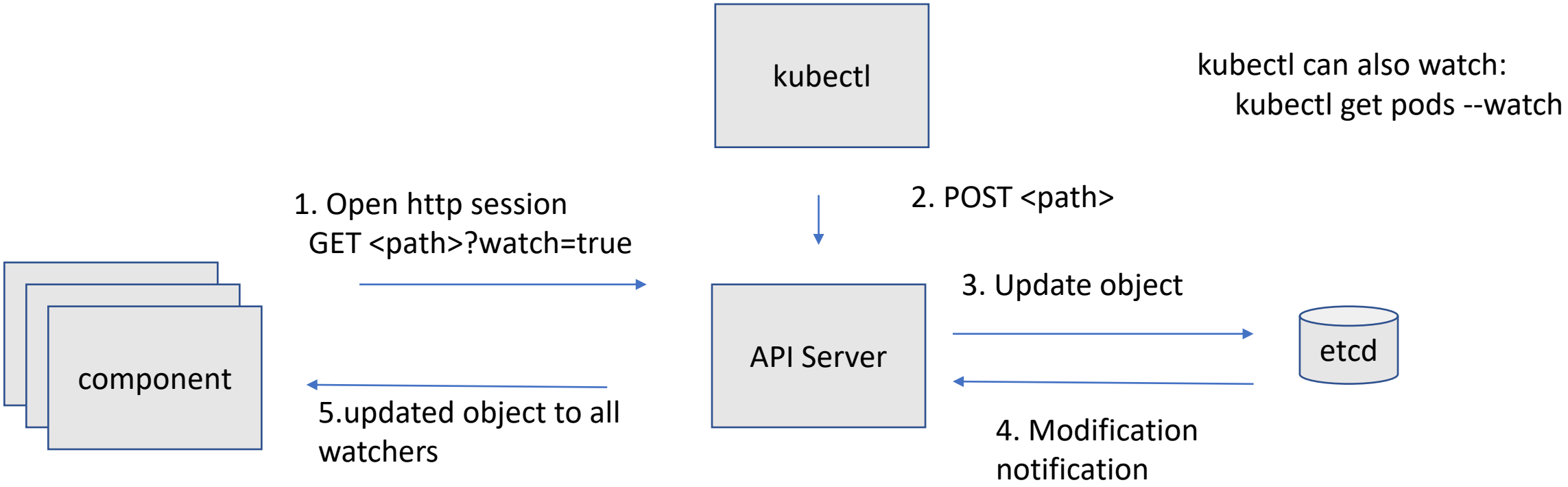
- Provides a CRUD (create, read, update, delete) interface for querying and modifying the cluster state over a REST API
  - Stores the state in etcd



- Admission control plugins can modify the resource, e.g.:
  - initialize missing fields
  - reject requests
- e.g., ServiceAccount - apply the default service account to pods
- e.g., NamespaceLifecycle - prevent creation of pods in namespaces being deleted

https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

API Server to Component Comm



Scheduler

https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

- Waits for newly created pods (API server watch) then assign a node to each new pod
  - Note: Doesn’t instruct the selected node to run the pod
- Basic operation:
  - Filter - finds the set of Nodes where it's feasible to schedule the Pod
  - Score - ranks the remaining nodes to choose the most suitable Pod placement
  - Broken down into 11 “extension points” https://kubernetes.io/docs/reference/scheduling/config/#extension-points
- Defaults: https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins

Controller Manager

Runs multiple controllers as processes performing various reconciliation tasks

- Replication Manager (for replicationcontroller resources)
- ReplicaSet
- DaemonSet
- Deployment
- Node
- Service
- …

Controller Manager - controllers

- Watch the API Server for changes to resources they care about and perform operations for each change
  - e.g., ReplicaSet watches ReplicaSet resources, and Pod resources, and creates/deletes Pod resources
  - e.g., Endpoints watches Services resources and Pod resources, and creates Endpoints resources

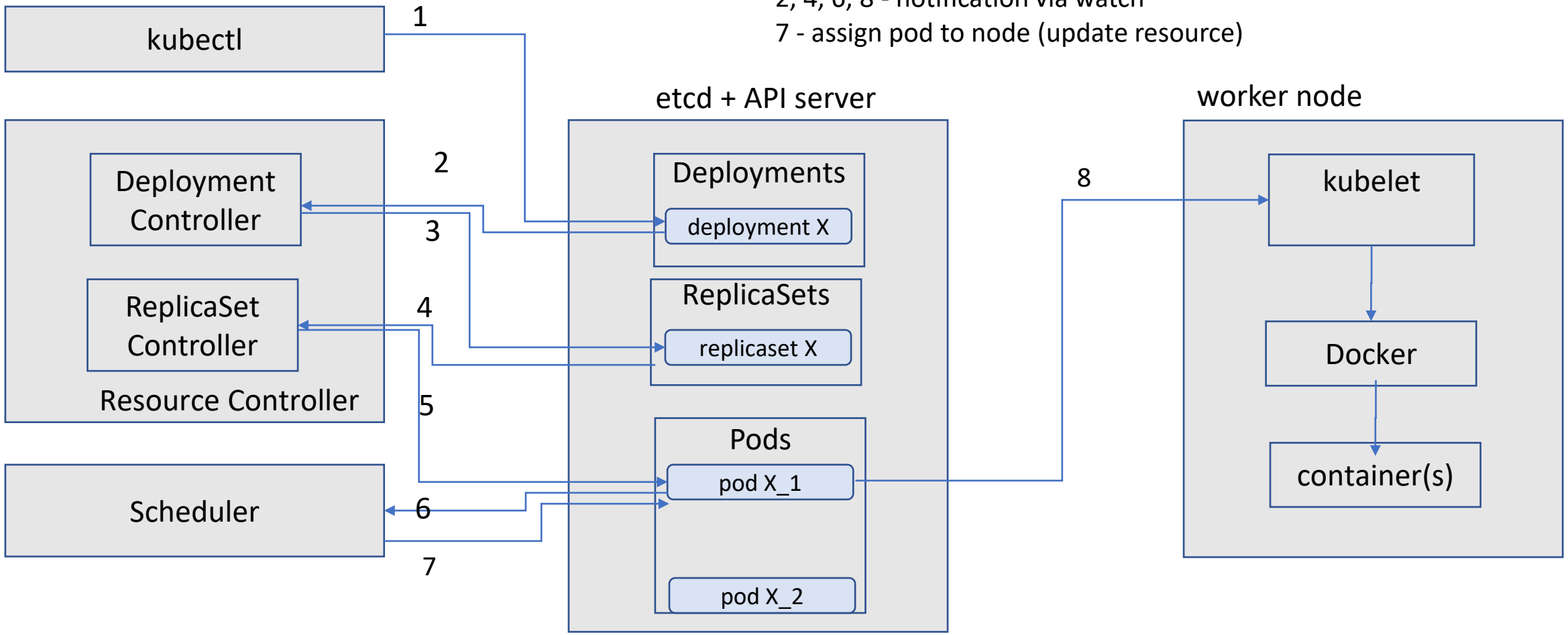
- Each runs a reconciliation loop (desired state specified / current state as seen from the API server)
  - https://github.com/kubernetes/kubernetes/tree/master/pkg/controller

Kubelet

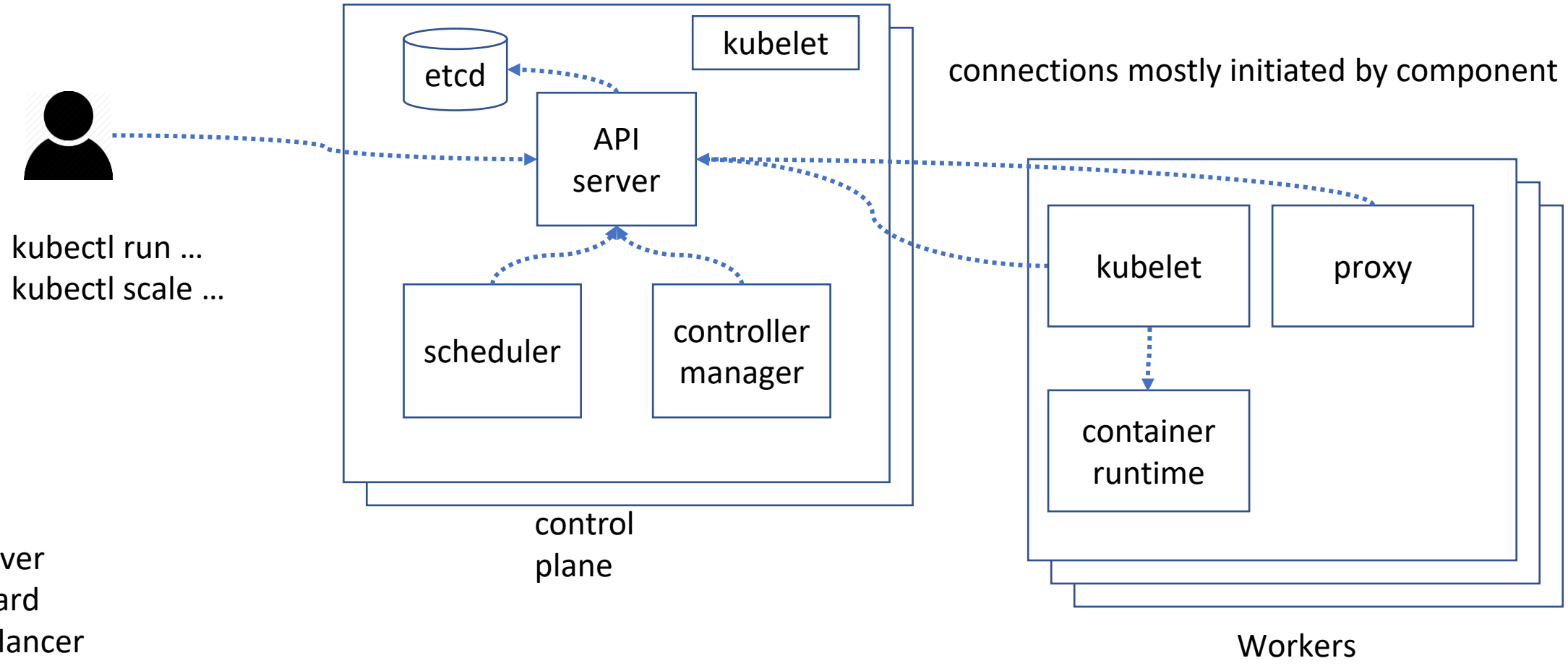
- Register the node it’s running on by creating a Node resource in the API server
- Watches API server for Pods that have been scheduled to its node
- Starts the Pods containers by telling the container runtime to run a container from a given image
- Monitors running containers and reports any events
- Runs the liveness probes, restarts containers when probes fail
- Deletes containers when Pod is deleted (watch API server)
- Can run pods specified in local pod manifest (for running system components) - called Static Pods.
  - /etc/kubernetes/manifests/

Summary - Chain of Events (Fig 11.12)

1, 3, 5 - create <> resource
2, 4, 6, 8 - notification via watch
7 - assign pod to node (update resource)



Recall: High-level Architecture (partial)



Others:

- DNS Server
- Dashboard
- Load balancer
  (Ingress Controller, External Load balancer)
- CNI plugin

Next: Networking

University of Colorado **Boulder**

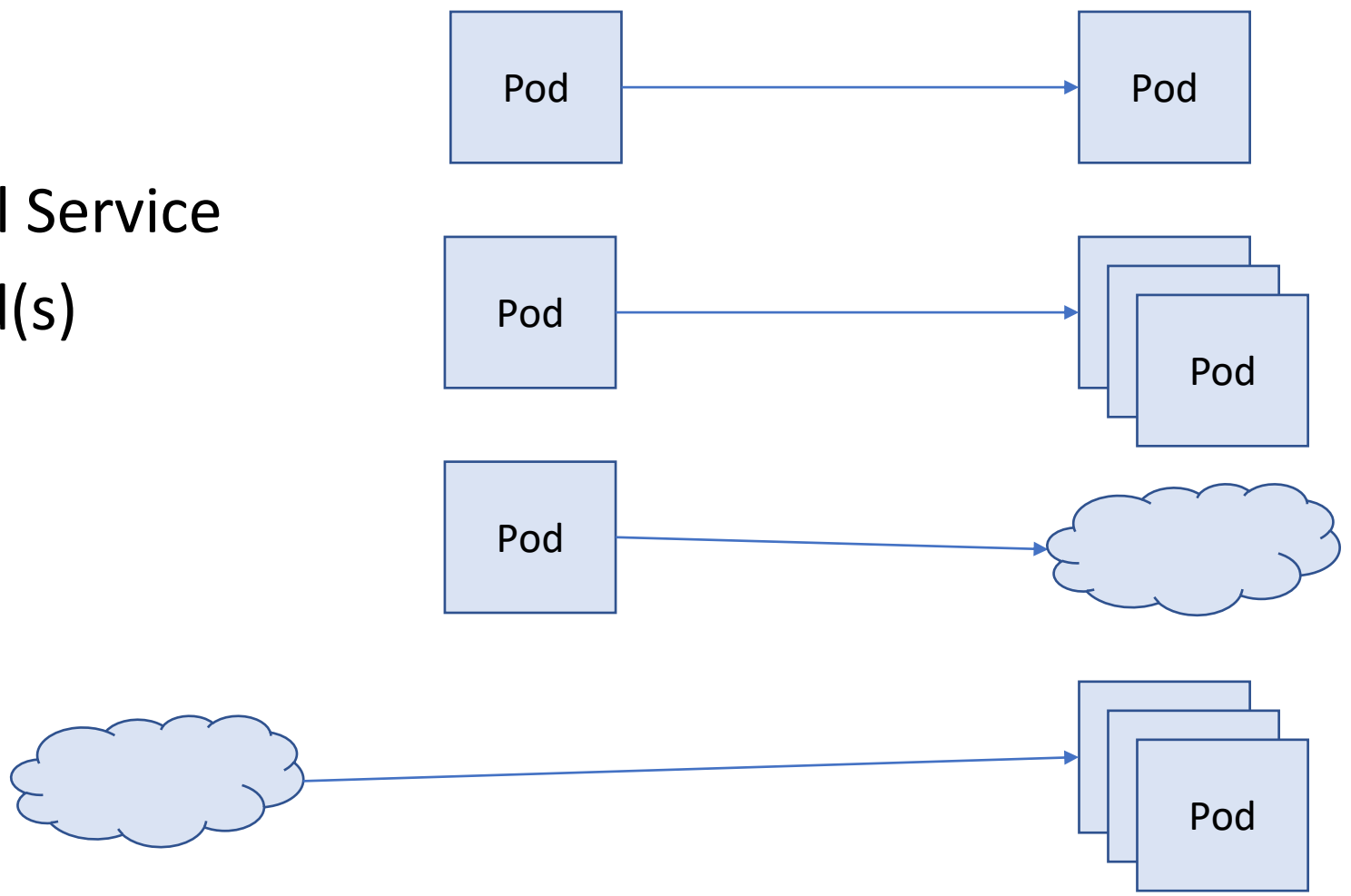
Kubernetes Networking

Course: Networking Principles in Practice – Linux Networking
Module: Kubernetes

University of Colorado **Boulder**

Needed Communication

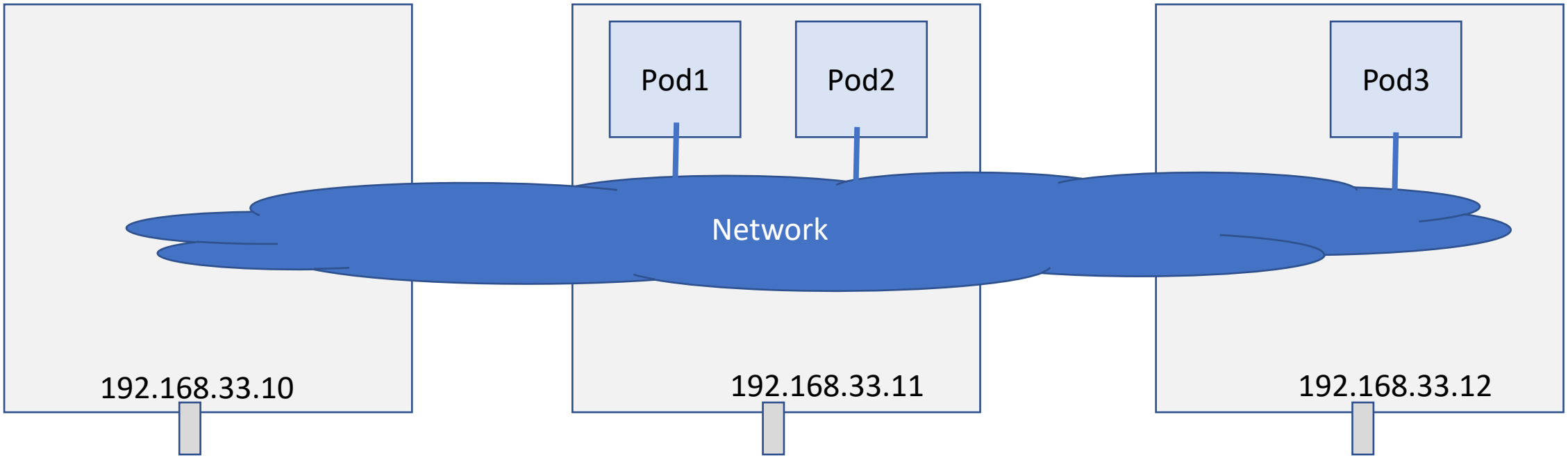
- Pod to Pod
- Pod to Pods
- Pod to External Service
- External to Pod(s)



Pod to Pod

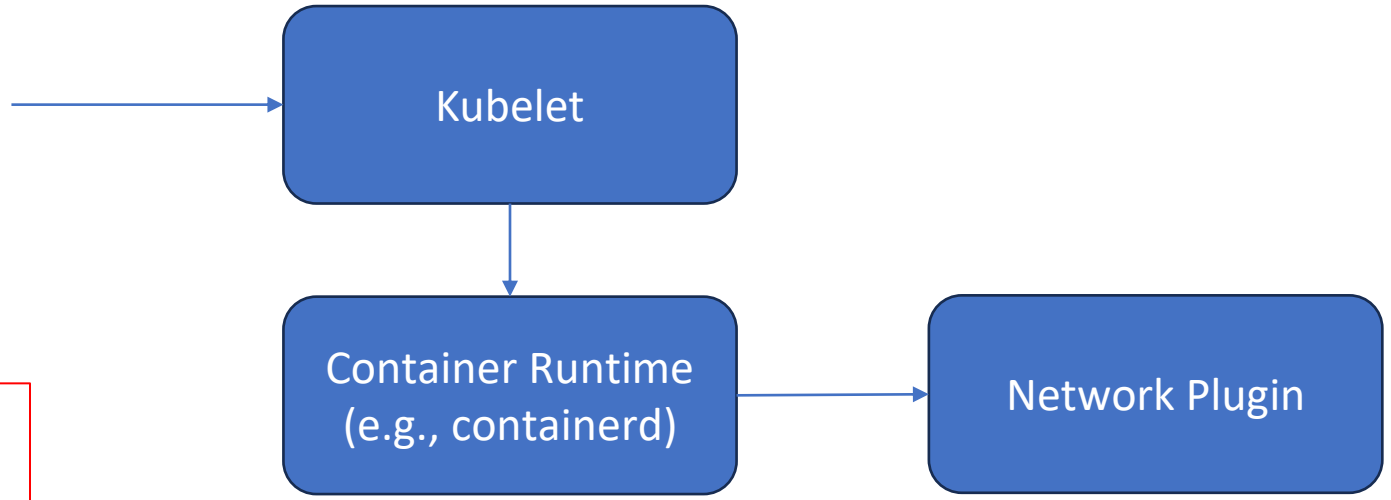
Pods

- Each Pod gets an IP address (in 10.244.0.0/16 space)



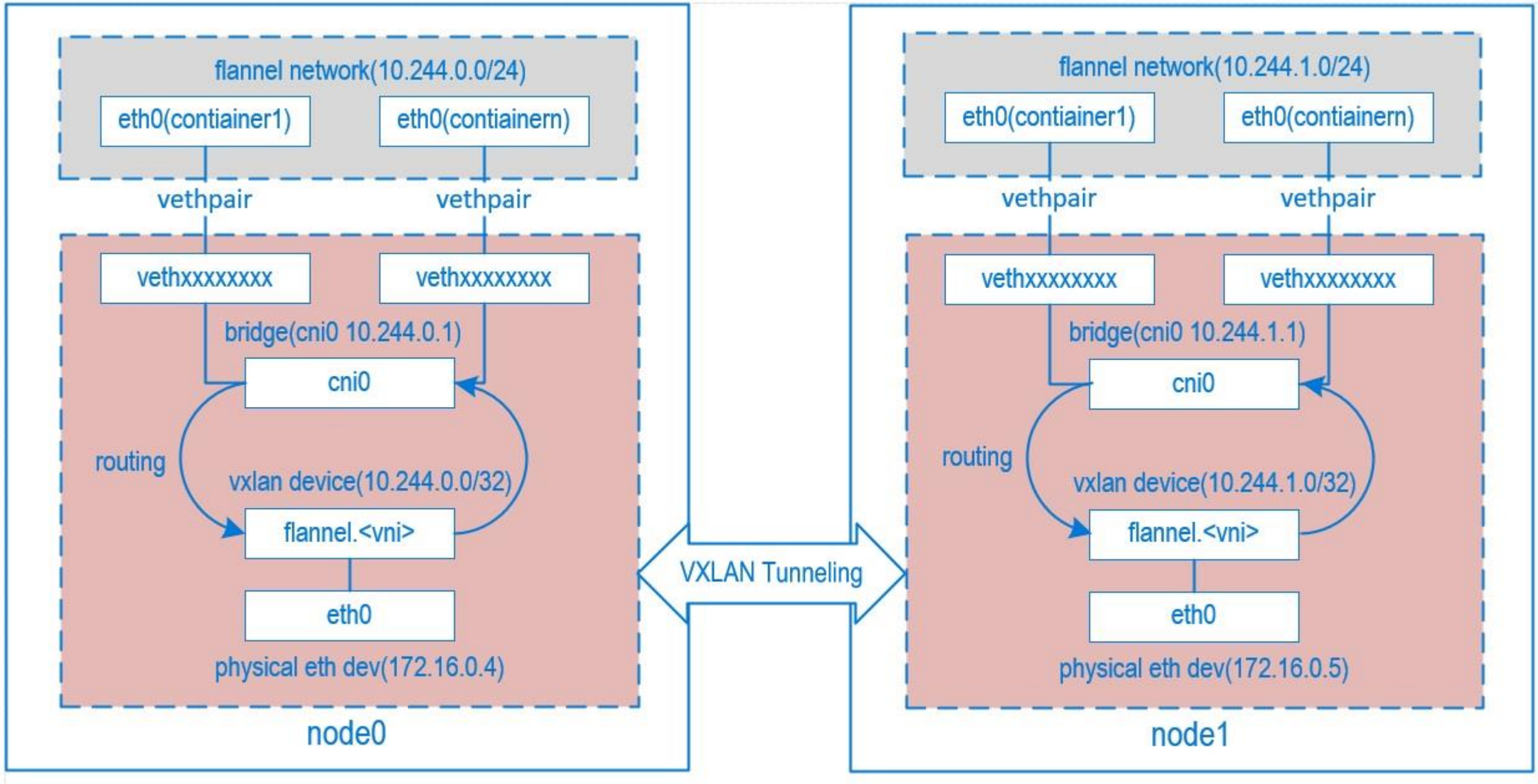
Container Network Interface (CNI)

- Container runtime creates the Pod, and calls a network plugin to set up the networking for that Pod
- The configuration and API of a network plugin follows the CNI spec
  - ADD
  - DEL
  - CHECK
  - …



Next lesson, we will create a network plugin

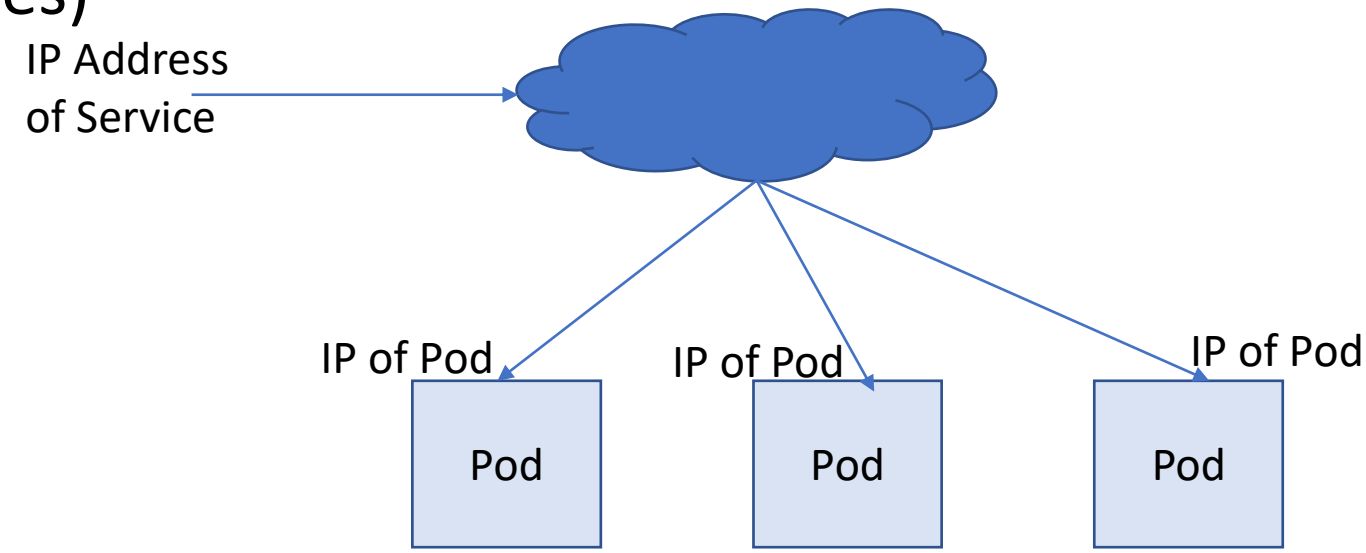
https://github.com/containernetworking/cni/blob/main/SPEC.md


flannel network(10.244.0.0/24)
eth0(contiainer1)
eth0(contiainern)
vethpair
vethpair
vethxxxxxxxx
vethxxxxxxxx
bridge(cni0 10.244.0.1)
cni0
routing
vxlan device(10.244.0.0/32)
flannel.<vni>
eth0
physical eth dev(172.16.0.4)
node0
VXLAN Tunneling
flannel network(10.244.1.0/24)
eth0(contiainer1)
eth0(contiainern)
vethpair
vethpair
vethxxxxxxxx
vethxxxxxxxx
bridge(cni0 10.244.1.1)
cni0
routing
vxlan device(10.244.1.0/32)
flannel.<vni>
eth0
physical eth dev(172.16.0.5)
node1

Pod to Pods

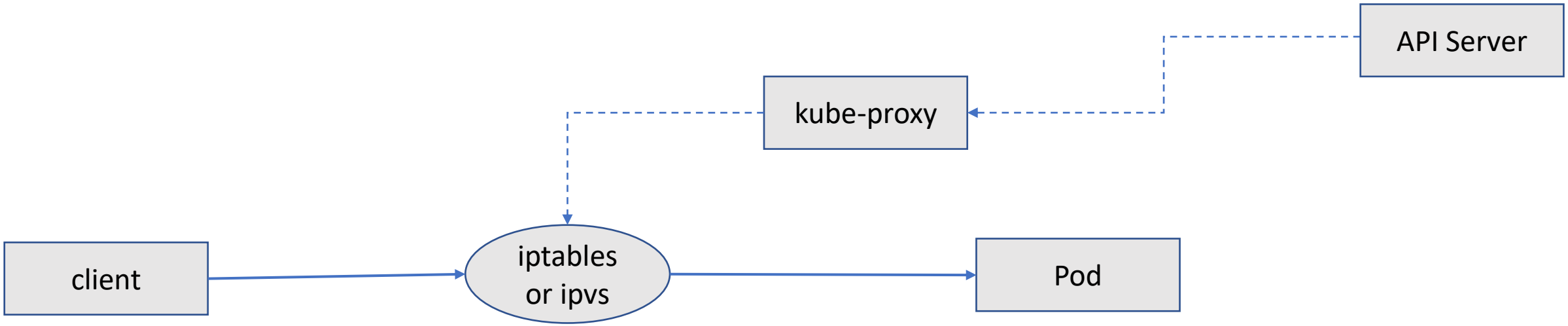
Service

- “A Service is an abstraction which defines a logical set of Pods and a policy by which to access them”
- Load Balance between then
- Uses a selector to see which nodes should be in set (across scaling and failures)



kube-proxy

- Makes sure clients can connect to the services you define, load balanced when needed
- Runs on every node
- Not in path of traffic - it just interfaces to iptables or ipvs



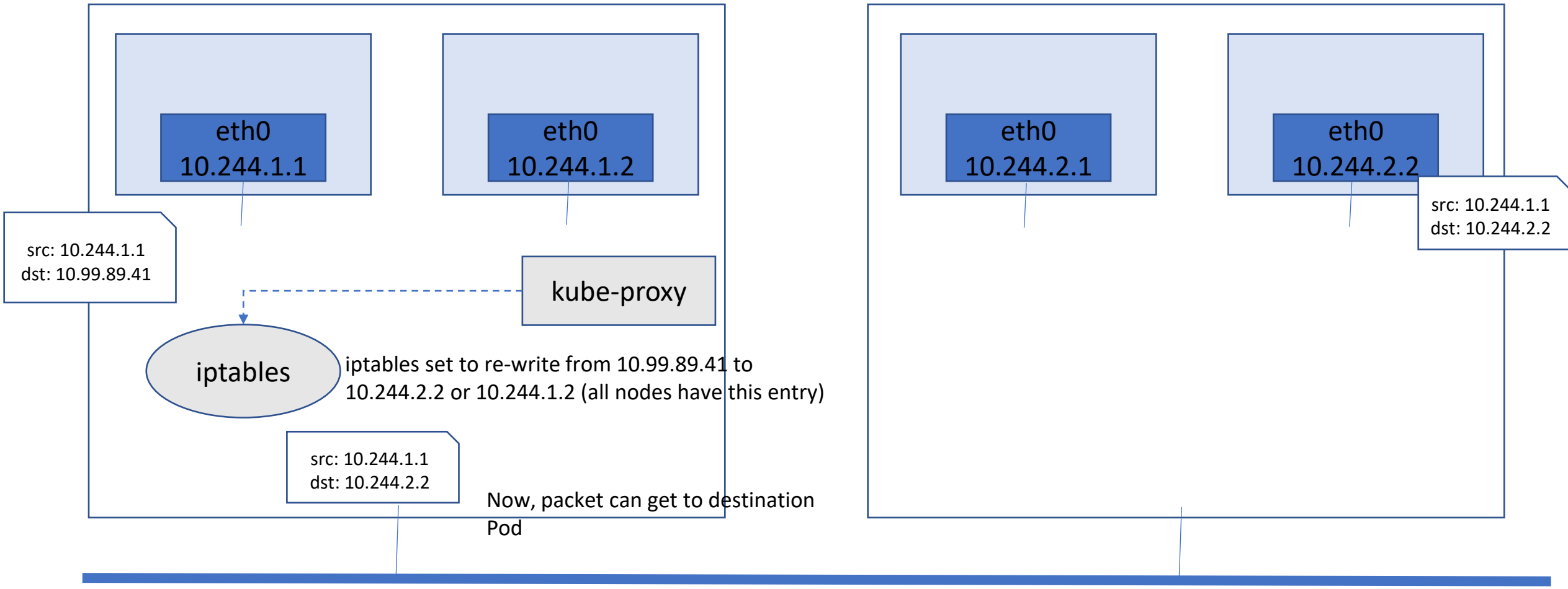
Load Balancing with iptables

```
iptables -A PREROUTING -t nat -p tcp -d 192.168.1.1 --dport 27017 \
    -m statistic --mode random --probability 0.33 \
    -j DNAT --to-destination 10.0.0.2:1234

iptables -A PREROUTING -t nat -p tcp -d 192.168.1.1 --dport 27017 \
    -m statistic --mode random --probability 0.5 \
    -j DNAT --to-destination 10.0.0.3:1234

iptables -A PREROUTING -t nat -p tcp -d 192.168.1.1 --dport 27017 \
    -j DNAT --to-destination 10.0.0.4:1234
```

https://scalingo.com/blog/iptables

Services Networking

- Service gets IP address (no interfaces created - just a virtual IP address)
  - example End Points: 10.244.1.2 (node 1) and 10.244.2.2 (node 2)



Pod to External

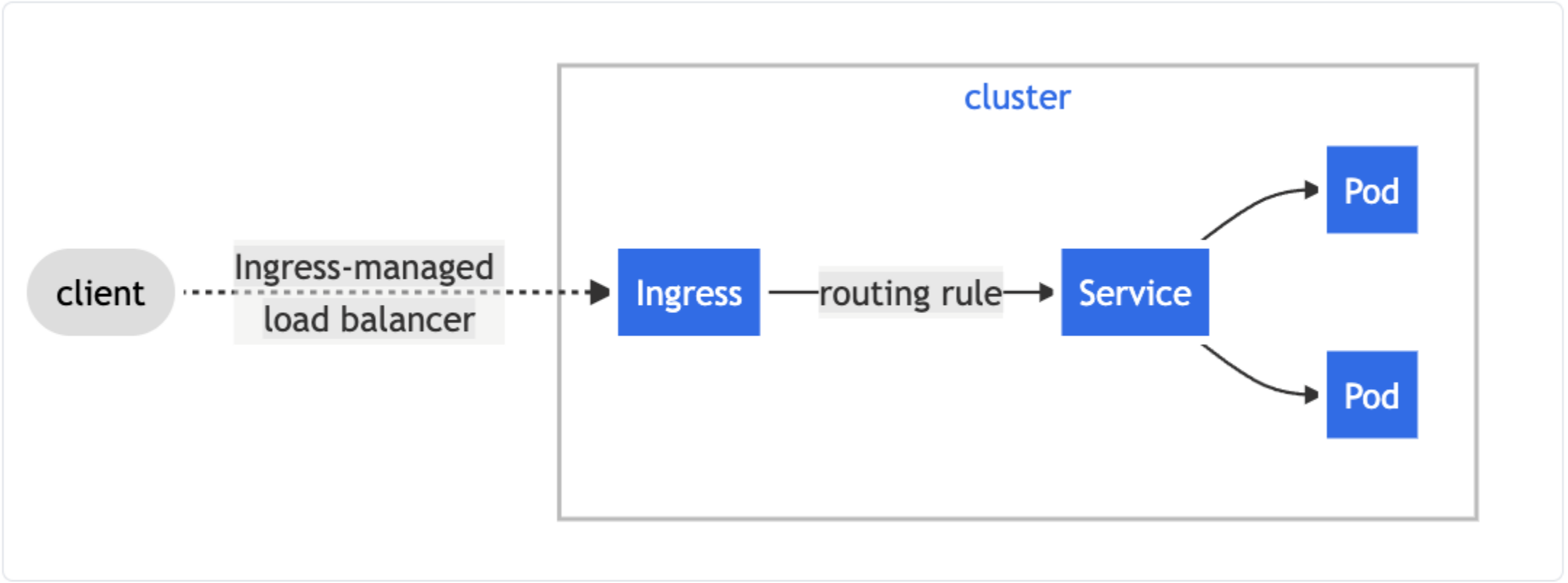
Services - Internal to External

- Allows same mechanism to be used in both cases
- Allows switching back and forth


```
apiVersion: v1
kind: Service
metadata:
  name: someservice
spec:
  type: ExternalName
  externalName: blah.com
  ports:
  - port: 80
```

External to Internal

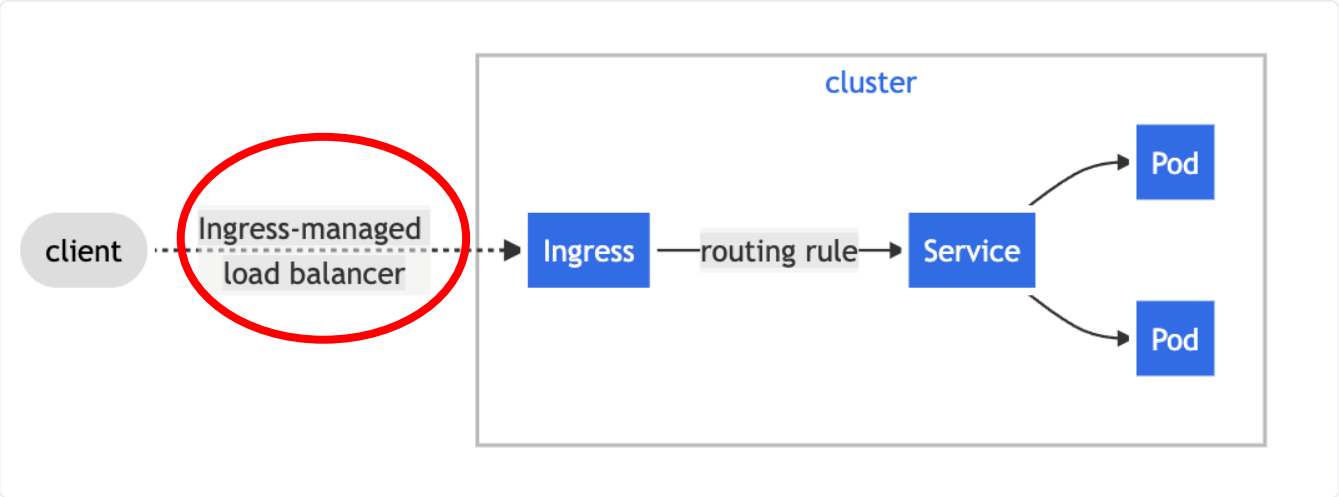
Ingress and Ingress Controller



Ingress Controller

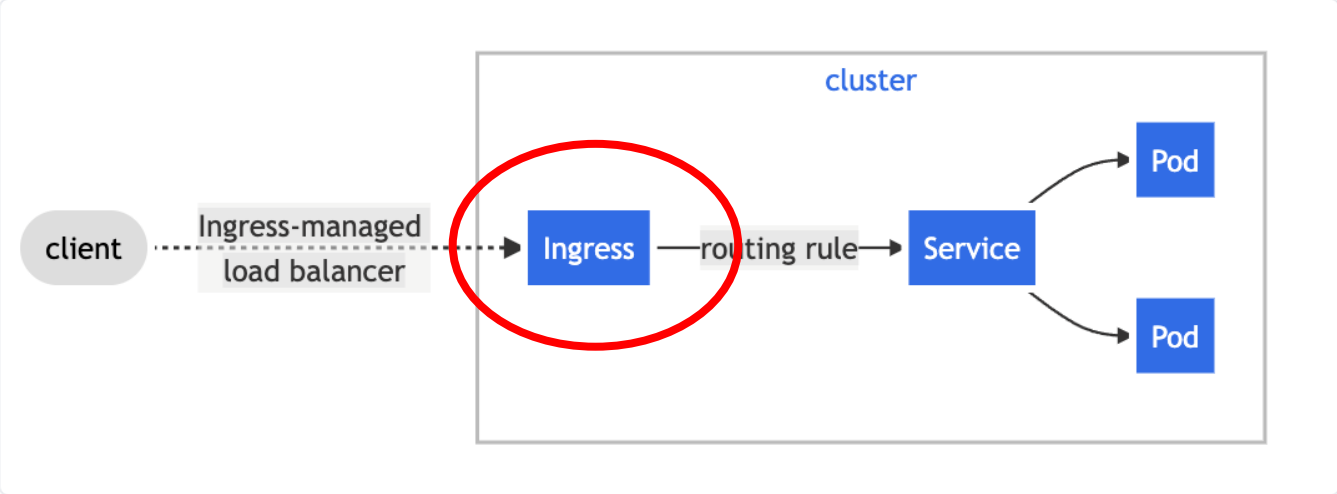
- A load balancer, such as nginx, traefik, haproxy, etc.
- Need to deploy an Ingress Controller first
  - Note: Cloud Providers have already done so

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.4/deploy/static/provider/cloud/deploy.yaml



Ingress

- Then, create an Ingress to tell the Ingress Controller how to reach a service




```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  namespace: default
  annotations:
    # use the shared ingress-nginx
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: localhost
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: blog-svc
            port:
              number: 9999
```

Next: Creating a Network Plugin

University of Colorado **Boulder**

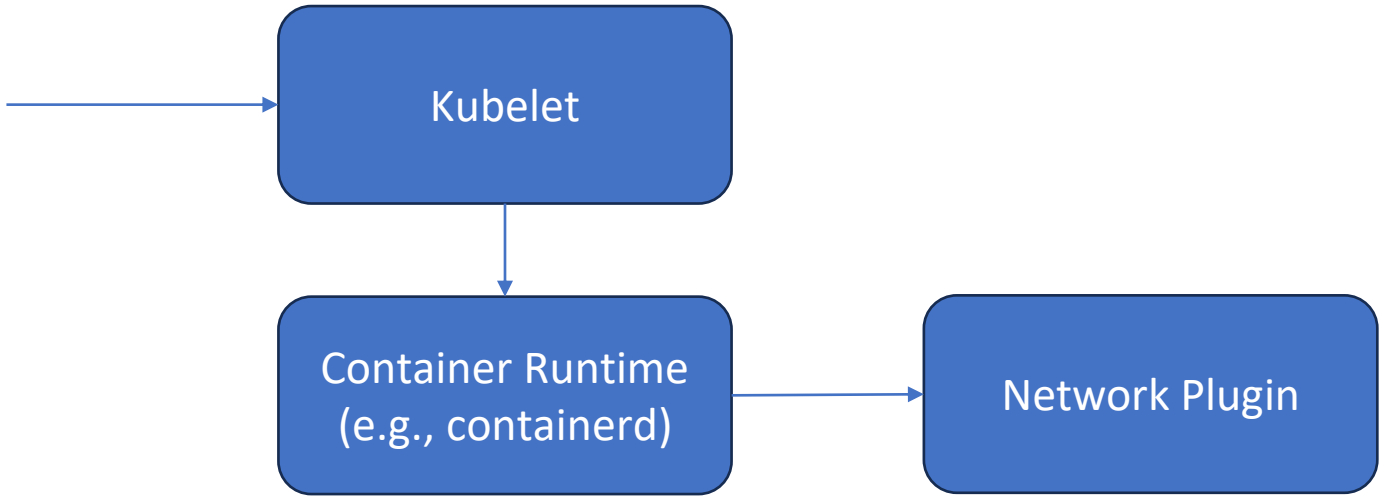
Creating a Network Plugin

Course: Networking Principles in Practice – Linux Networking
Module: Kubernetes

University of Colorado **Boulder**

Recall: Container Network Interface (CNI)

- Container runtime creates the Pod, and calls a network plugin to set up the networking for that Pod
- The configuration and API of a network plugin follows the CNI spec
  - ADD
  - DEL
  - CHECK
  - …



https://github.com/containernetworking/cni/blob/main/SPEC.md

How it works (1): config in /etc/cni/net.d

- It’ll check /etc/cni/net.d for configuration files
- The one with the lowest prefix number will be used (10-something.conf, 20-something.conf…)

Example plugin that kind installs:

- docker exec -it kind-worker /bin/bash
- cd /etc/cni/net.d
- ls
- cat 10-kindnet.conflist

How it works (2): executable in /opt/cni/bin

- Note the plugins type. (ptp)
- That's a binary in /opt/cni/bin/ that implements the CNI's commands
- Look at the spec for what the rest of the fields do.

```
{
	"cniVersion": "0.3.1",
	"name": "kindnet",
	"plugins": [
	{
		"type": "ptp",
		 …
```

How it works (3): installed on each node

- If you look on each node, both the config and executable will be there (the config may be different – e.g., different IP address

control-plane

```
…
    "type": "ptp",
    …
    [ { "subnet": "10.244.0.0/24" } ]
…
```

worker

```
…
    "type": "ptp",
    …
    [ { "subnet": "10.244.2.0/24" } ]
…
```

worker2

```
…
    "type": "ptp",
    …
    [ { "subnet": "10.244.1.0/24" } ]
….
```

Let’s Start with a Skeleton

- https://github.com/eric-keller/npp-linux-05-kubernetes

- Config: 09-nppnet-skel.conflist
- Executable: nppnet-skel (bash script)
  - Note: this will print out to /var/log/nppnet.log

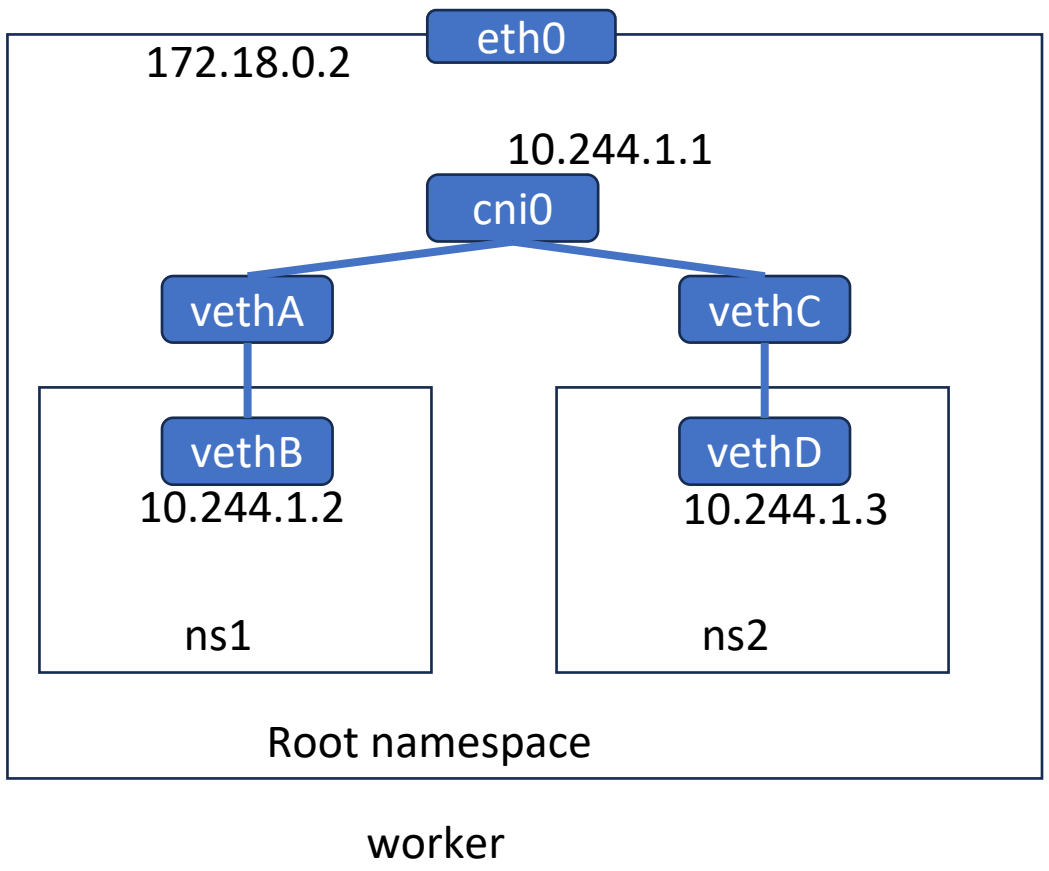
Test it out (1) - setup

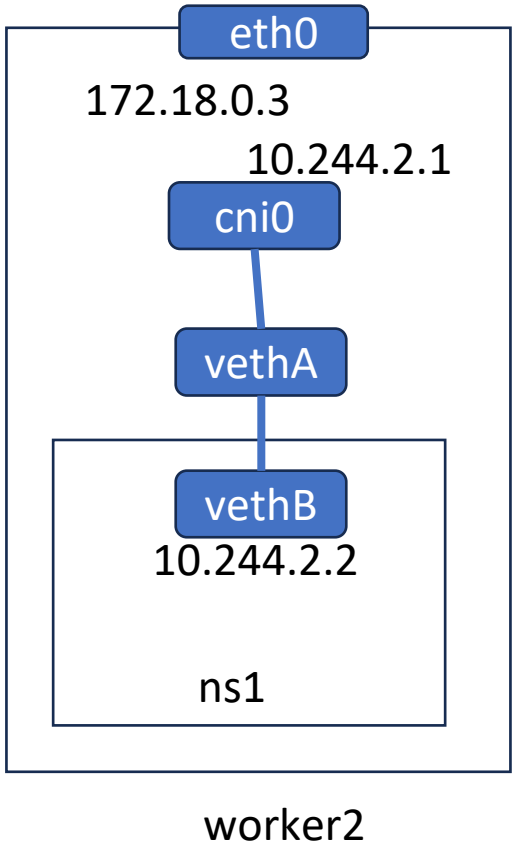
- ./make-dirs.sh
- kind create cluster --config cluster-configs/1worker2Mount.yaml (note the mounting of directories)
- Copy bin and conf into tmp/w1
  - cp cni/09-nppnet-skel.conflist tmp/w1
  - cp cni/nppnet-skel tmp/w1
- Exec into worker1
  - docker exec -it kind-worker /bin/bash
- Copy bin and conf into correct locations
  - cp /npp-temp/09-nppnet-skel.conflist /etc/cni/net.d/
  - cp /npp-temp/nppnet-skel /opt/cni/bin

Test it out (2) run a Pod and see logs/errors

- Apply a label to worker1
  - kubectl label node kind-worker node=node1
- Launch a pod (it has selector to make it get scheduled on worker1)
  - kubectl apply -f pod-configs/simple-nginx-node1.yaml
- In host
  - kubectl describe node kind-worker
    (probably no errors unless there's an error in the config)
  - kubectl describe pod mynginx
    (should show an error)
- Inside of kind-worker container
  - journalctl -u kubelet
  - cat /var/log/nppnet.log

Creating Full Network Plugin





In container (ns1):
10.244.0.0/16 via 10.244.1.1

In host (install time):
10.244.2.0/24 via 172.18.0.3
Note: kind does this by default

What Our Network Plugin Needs to Do

- Allocate IP address
- Create entry in /var/run/netns/ (for ip netns)
- Create veth pair
- Put veth into namespace
- Set up networking within the namespace (address, route)

Network Plugin Installation

- Repeat (in case you haven't yet)
  - Make tmp/w1 and tmp/w2 directories (for mounting)
  - Create Cluster
    kind create cluster --config ./cluster-configs/1master2workerMount.yaml
  - Set labels: e.g., kubectl label node kind-worker node=node1)

- Network Plugin Installation
  - Put config in /etc/cni/net.d (Edit unique subnet for each worker)
  - Put executable in /opt/cni/bin
  - Create cni0 bridge
  - Script provided to do all of that (nppnet-install.sh – must be edited)

Sample Activities

- Run pods and look at their info
  - kubectl apply -f pod-configs/forexec1-node1.yaml
  - kubectl get pods -o wide
  - kubectl describe pod forexec1
- Test connectivity with ping
  - kubectl exec -it forexec3 -- ping 10.244.2.2
- Get inside of one of the nodes (and look at the log):
  - docker exec -it kind-worker /bin/bash
  - cat /var/log/nppnet.log

Recap of Network Principles in Practice: Linux Networking

- Module 1 – Introduced Linux's networking capabilities
  - Starting with bridge and devices
- Module 2 – Explored Linux's capabilities to create a router
  - With command line options for setting up the forwarding table, and software for running routing protocols
- Module 3 – Extended to Layer 4 capabilities
  - Filtering (iptables), Load balancing (ipvs), and Traffic Shaping (tc)
- Module 4 – Covered Linux's support for virtual networking
  - With network namespaces, and docker networking
- Module 5 – Saw virtual networking applied for large scale systems
  - Going over Kubernetes, Kubernetes networking abstractions, and how to write a network plugin
- Suggested Next: eBPF

University of Colorado **Boulder**