Cloud Native Networking

Understand the current cloud networking basics.

Have a hands-on experience with a learning tool focused on Networking aspects of Kubernetes

(and multi-cluster networking using Skupper)

Hands On Learning Tool: What it is not

1. ❗ a real Kubernetes setup
2. ❗ a single-node k8s runner like minikube, kind

Hands On Learning Tool: What it is

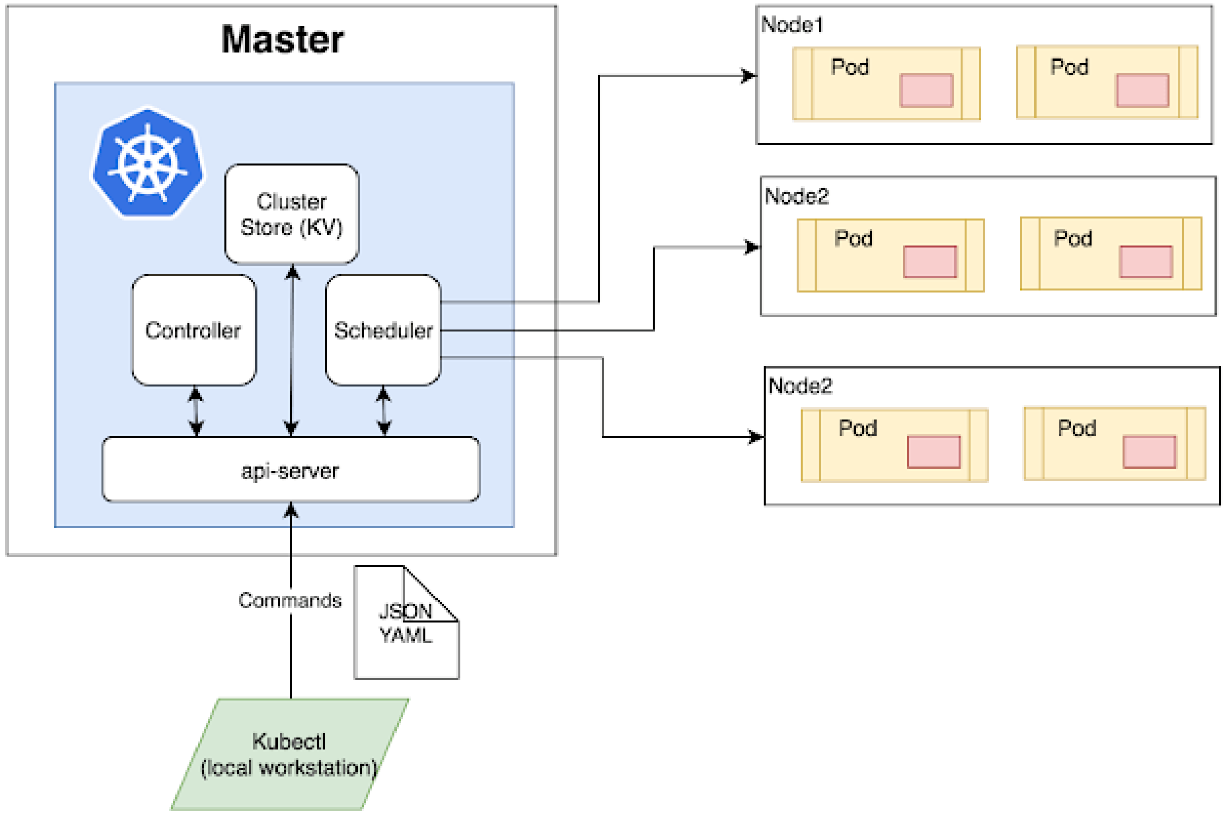
1. Mininet based simulation
2. Uses as many real components
3. Models reality as close as possible

Refresher: Containers

- Application executable packaged with its libraries and dependency
- No separate guest OS like virtual machines
- Light-weight and portable
- Isolation through namespace and cgroups
    - Namespace: isolated view of resources like file system, network resources
    - cgroups: Resource limit per process/container

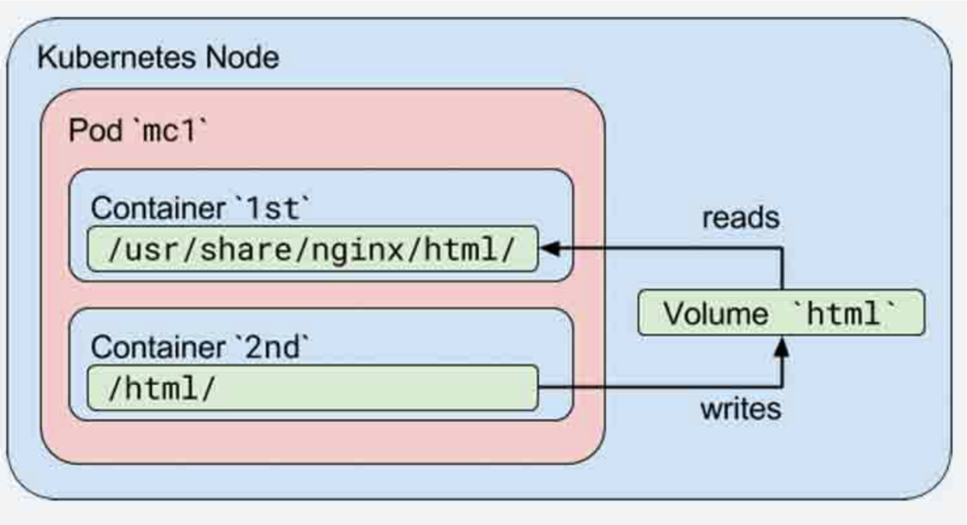
What is Kubernetes?

- Docker: Single machine container deployment
- Kubernetes (k8s): Container Orchestration
  - Across a cluster of machines
  - Manage automated deployment, scaling



Kubernetes Overview

- Pods:Application specific logical host.
  - group of containers with shared storage and network resources.




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Kubernetes Overview: Deployment

- Manage replicas and scaling of pods (for a desired state)
- Group of containers with shared storage and network resources.

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           18s
```

```
NAME                                READY     STATUS    RESTARTS   AGE       LABELS
nginx-deployment-75675f5897-7ci7o   1/1       Running   0          18s       app=nginx,
nginx-deployment-75675f5897-kzszj   1/1       Running   0          18s       app=nginx,
nginx-deployment-75675f5897-qqcnn   1/1       Running   0          18s       app=nginx,
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Kubernetes Overview

- Containers in a pod can communicate over localhost
- Pods identified by a cluster IP.
- Service:
  - Expose an application/pod
  - Handle multiple replica with single end point
  - Support dynamic up/down of pods

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 80
        name: http-web-svc

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app.kubernetes.io/name: proxy
  ports:
  - name: name-of-service-port
    protocol: TCP
    port: 80
    targetPort: http-web-svc
```

Mini demo: k8s hands on 🔨

Get yourself a temporary k8s instance here:

https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-interactive/

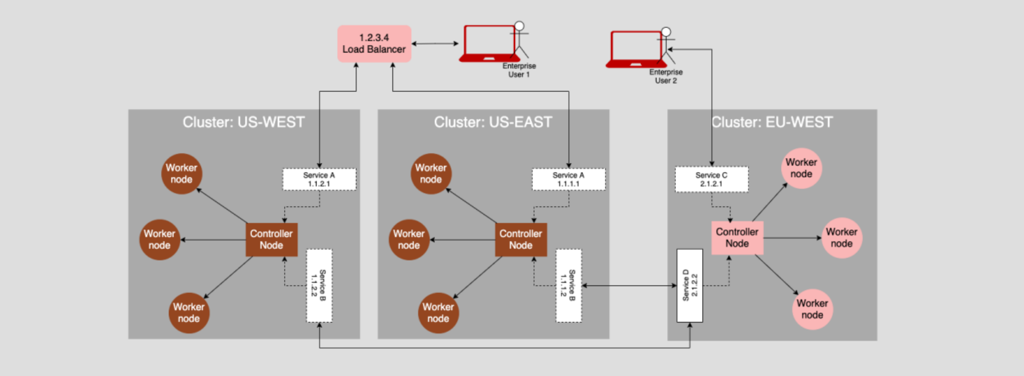
Follow the instructions here:

https://kubebyexample.com/concept/deployments

What is the deal with Kubernetes networking?

1. Connect containers running on different workers
2. Load balance multiple replicas as a single service
3. Expose services by name
4. App net features such as: rate limiting, health checks, blue-green testing

Now, do this across multiple clusters 😱



Aside: Reality of multi-cluster deployments

Most organization workloads are spanning multiple clusters now.

This is still an unsolved problem!

Rest of the deck is organized as follows

1. Introduce a Kubernetes networking concept
2. Discuss how it works in reality
3. Discuss how we run it in *knetsim*
4. Hands on!

Setup

Either build the image:

```
docker build -t knetsim -f ./Dockerfile .
```

or pull the image:

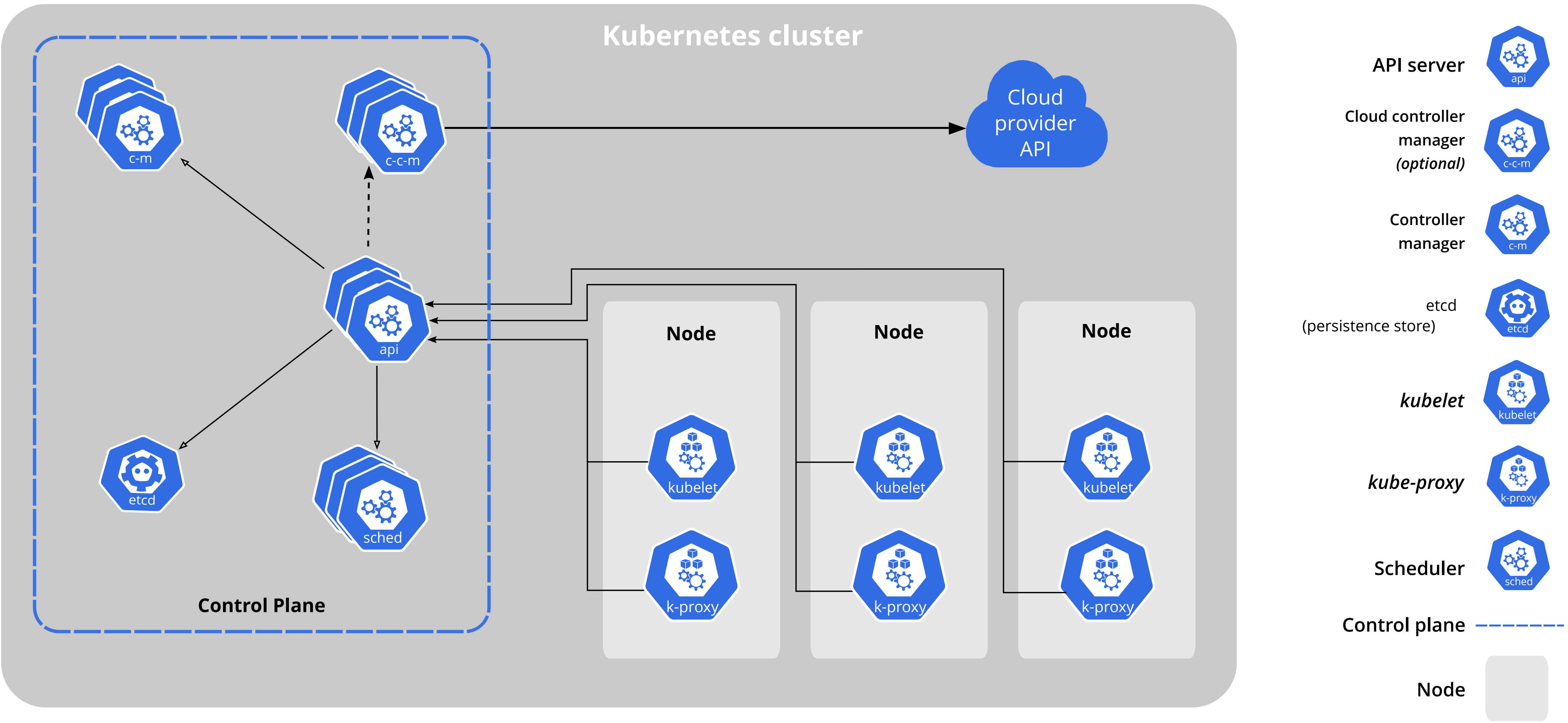
```
docker pull ghcr.io/IBM/k8s-netsim:master
docker tag k8s-netsim:master knetsim
```

To run the container:

```
docker run -it --privileged --rm --entrypoint bash knetsim
```

Structure

1. Workers and containers ⬅️
2. Container-Container communication
3. Service abstraction
4. Multi-cluster communication

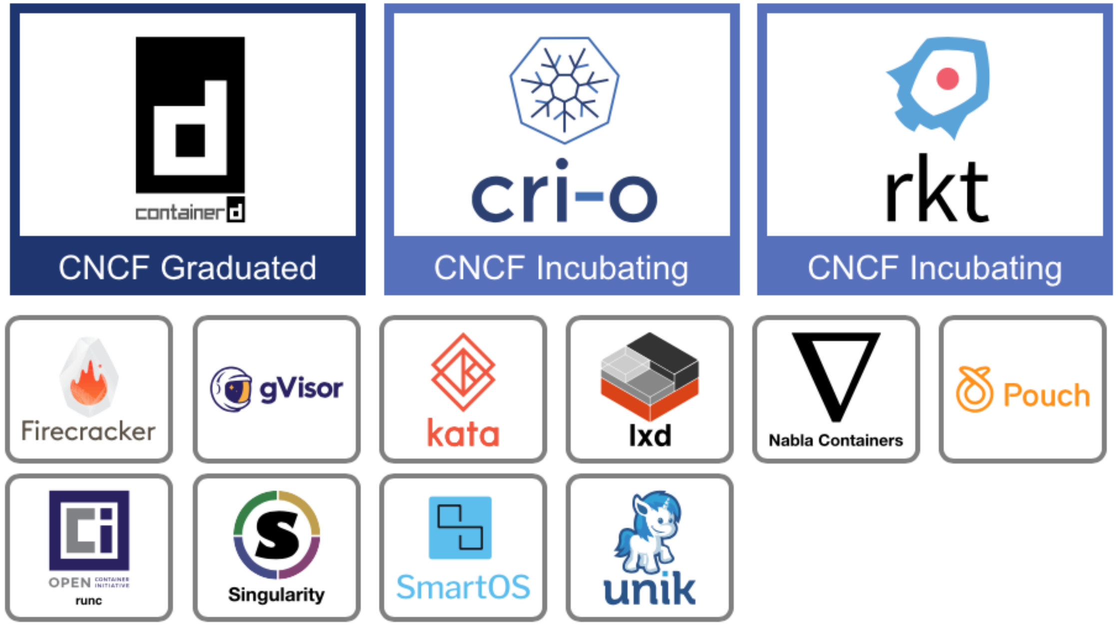

Kubernetes cluster
c-m
c-c-m
Cloud provider API
api
etcd
sched
Control Plane
Node
Node
Node
kubelet
kubelet
kubelet
k-proxy
k-proxy
k-proxy
API server
api
Cloud controller manager *(optional)*
c-c-m
Controller manager
c-m
etcd (persistence store)
etcd
kubelet
kubelet
kube-proxy
k-proxy
Scheduler
sched
Control plane
Node

C1: Workers and containers

- Kubernetes clusters consist of workers, each running pods
- Control Plane: Manages workers and pods scheduling, fault-tolerance
  - `kube-apiserver`: REST based front end for k8s frontend
  - `etcd`: highly-available key value store for backing k8s data
- Node Components:
  - `kubelet`: runs on every node to ensure containers are in running state
  - `kube-proxy`: Maintains network rules on a node.
    Leverages system packet filtering layer if available
  - `container runtime`: software for running containers. e.g. `containerd`, `docker`

C1: How does it work?

1. Each worker is setup with a `kubelet` component that manages containers
2. Containers are run using a runtime, like `containerd` or `docker`



C1: How do we do it?

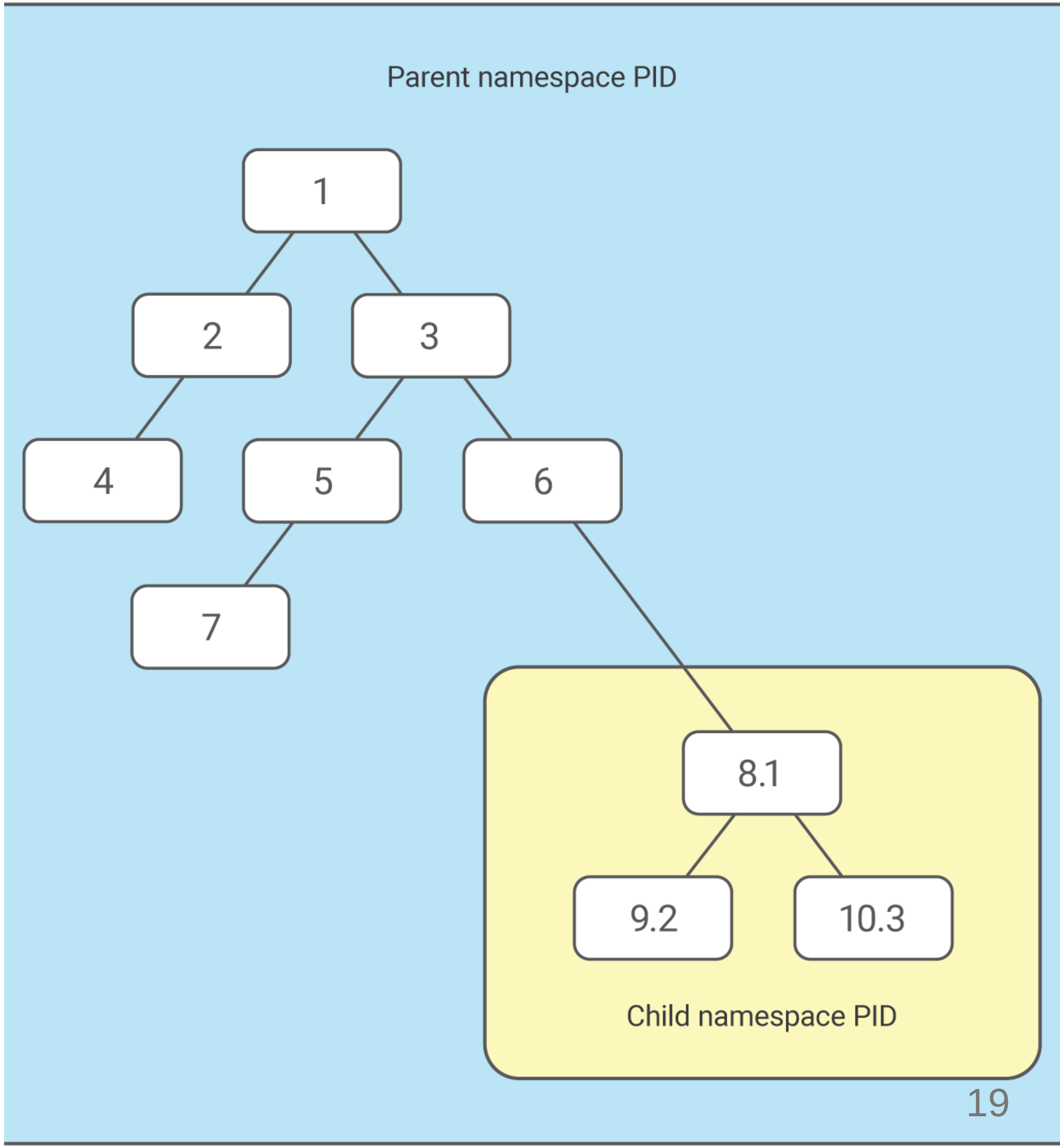
1. We use mininet `hosts` to represent each worker.
2. We run network namespaces to represent each container.

What are namespaces?

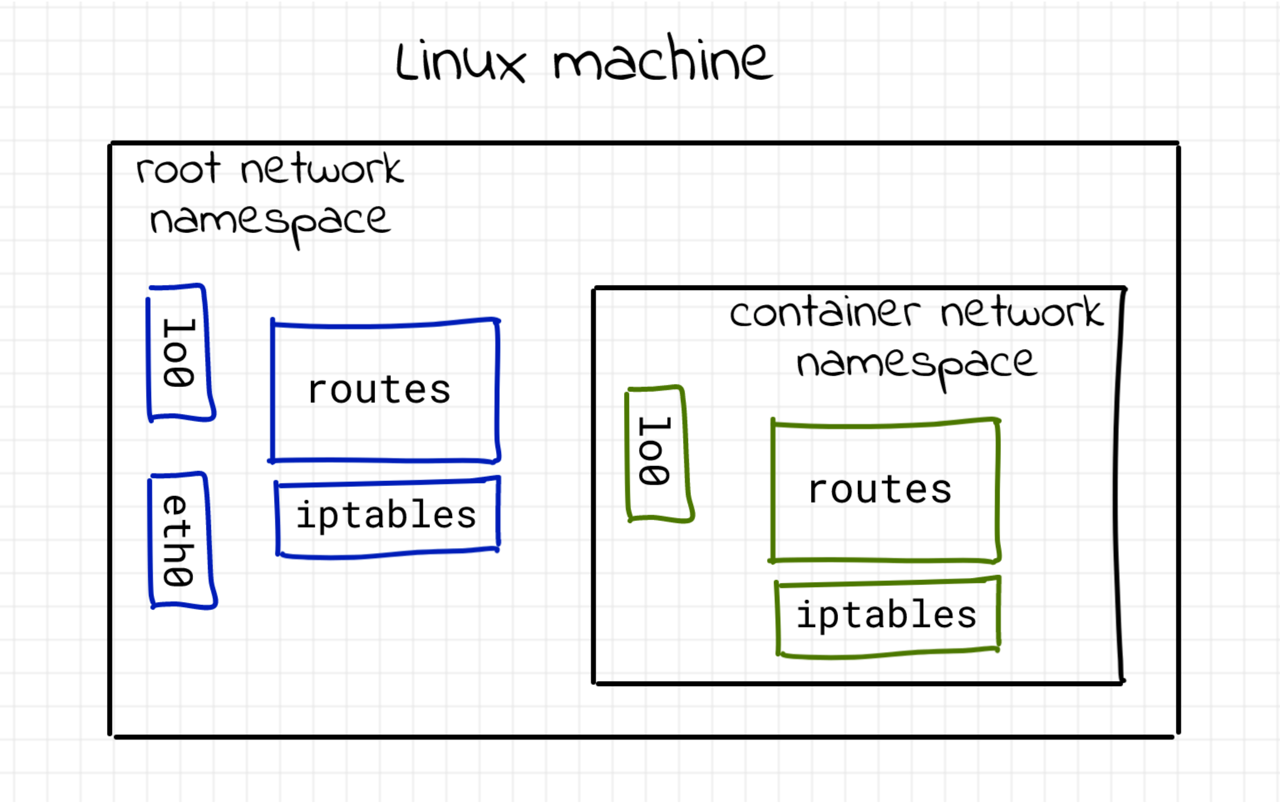
Kernel namespaces allow isolation of resources.

- user namespace: process can have `root` privilege within its user namespace
- process ID (PID) namespace: Have PIDs in namespace that are independent of other namespaces.
- network namespace: have an independent network stack (with routing rules, IP address)
- mount namespace: have moubt points without effecting host filesystem
- IPC namespace, UTS namespace

Example of pid namespacing



Example of networking namespacing



Mini demo: `ip netns` 🔨

Can use the `ip netns` command to create network namespaces.

Create a new container to play with:

```
docker run -it --privileged --rm --entrypoint bash knetsim
```

Check the root namespace:

```
ifconfig
```

List namespace:

```
ip netns list
```

(should come up empty)

Creating a new net ns 🔨

Create it:

```
ip netns add myns
```

It will currently be empty:

```
ip netns exec myns ifconfig
```

Create a new link:

```
ip netns exec myns ip link set lo up
```

Hands on 🔨

Ping localhost both in the root namespace and the newly created net namespace. Verify that the counters reported by `ifconfig` are independent.

Cleaning up 🔨

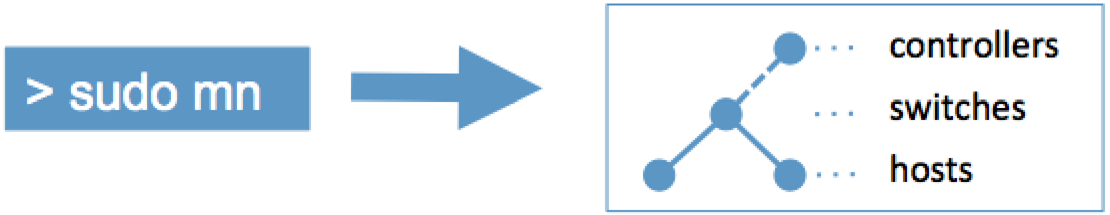
Delete the ns:

```
ip netns del myns
```

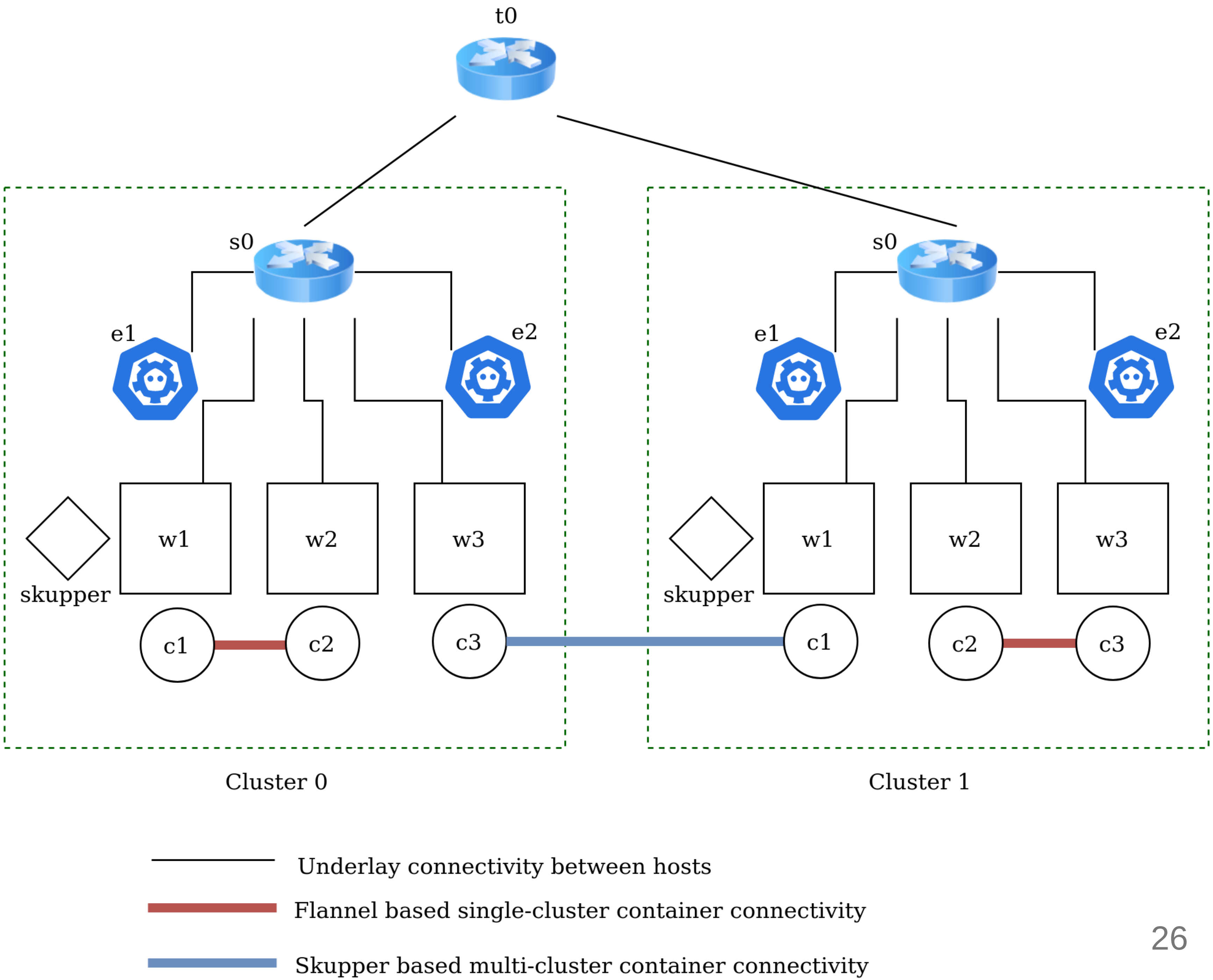
(Delete the container too)

What is mininet?

- Network Emulator
- Network of virtual hosts, switch, links on a machine



Our Topology



C1: Hands on

Run the simulator:

```
docker run -it --privileged --rm knetsim
```

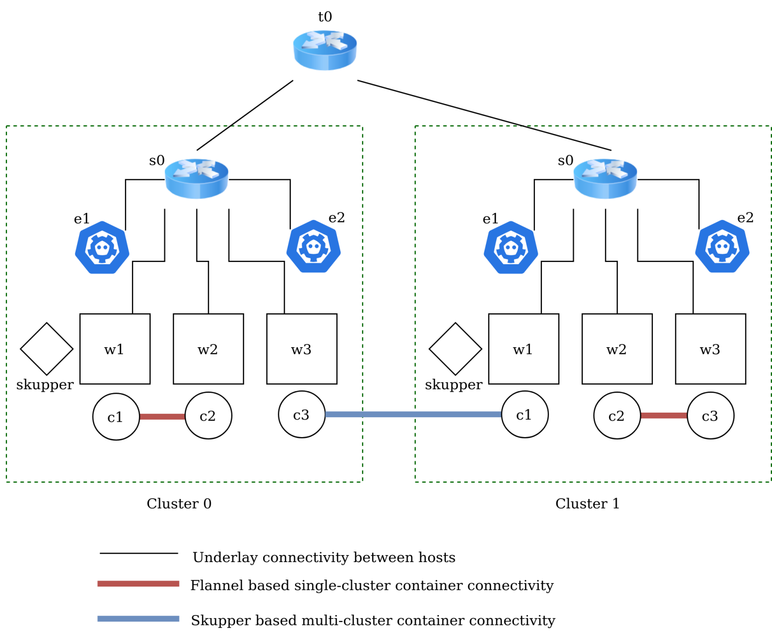
```
*** Creating network
...
<ping commands>
*** Starting CLI:
mininet>
```

Workers

- We have 2 clusters with 3 workers each:
- `C0w1`, `C0w2` and `C0w3` are workers => mininet hosts

Run commands on workers:

```
mininet> C0w1 ifconfig
```



Exercise 🔨

1. Ping the workers from each other.

Containers

1. Each worker `w<i>` has a container `c<i>`
2. Exec into containers using this command:

```
mininet> py C0w1.exec_container("c1", "ifconfig")
```

Exercise 🔨

1. Run a few commands in the container. See that only the network namespace is different from the underlying worker.
2. Create new containers:

```
mininet> py C0w1.create_container("c10")
```

(ignore the error it throws)
3. Delete the new container:

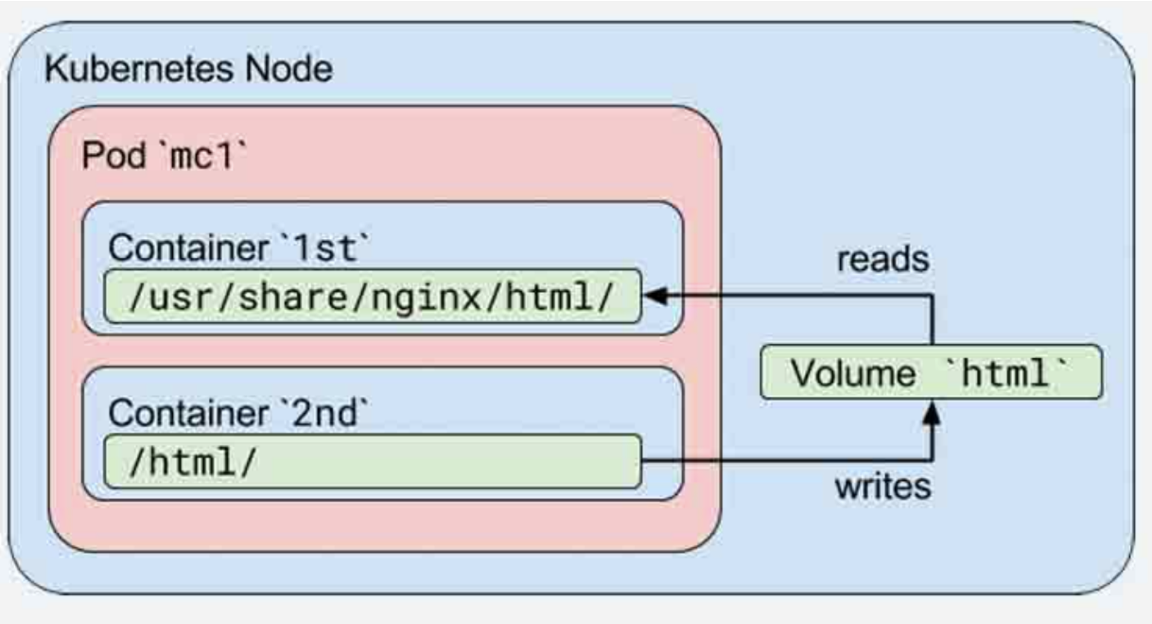
```
mininet> py C0w1.delete_container("c10")
```

Progress so far

1. Workers and containers ✓
2. Container-Container communication ⬅️
3. Service abstraction
4. Multi-cluster communication

C2: Container-container communication

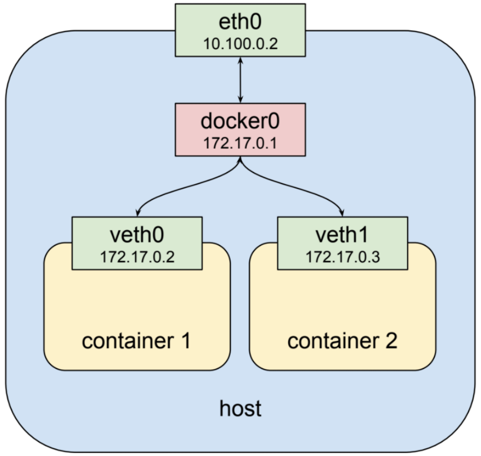
- For the moment, assume 2 services both have one replicas each
- 2 pods need to communicate
- Pod: group of containers with shared storage and network resources.



C2: How does it work?

In brief:

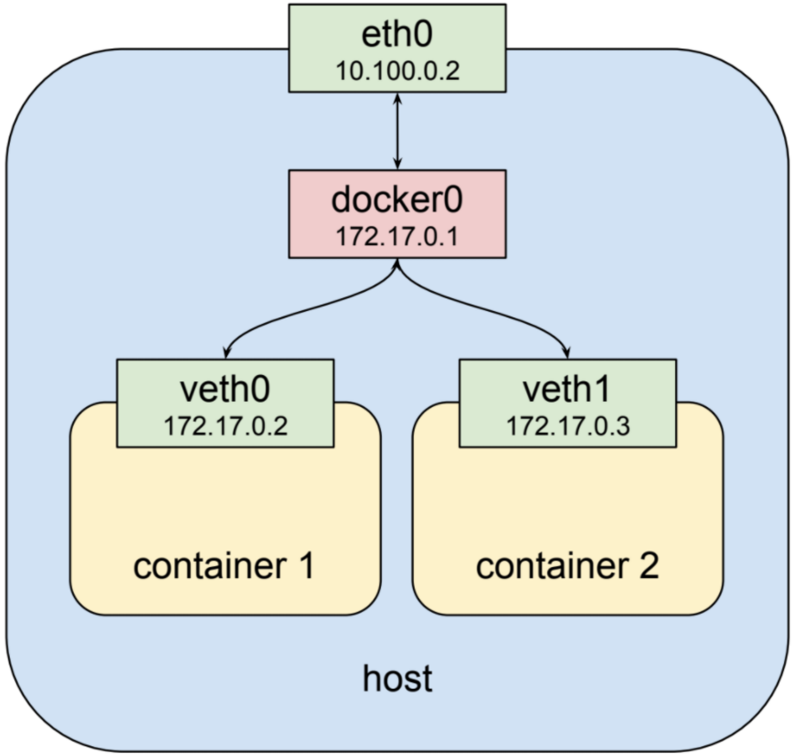
- The network namespace of the container is assigned an interface (eg eth0)
- A pod is assigned an IP
- Packets from one pod needs to be routed to another pod



C2: Pods on the same host

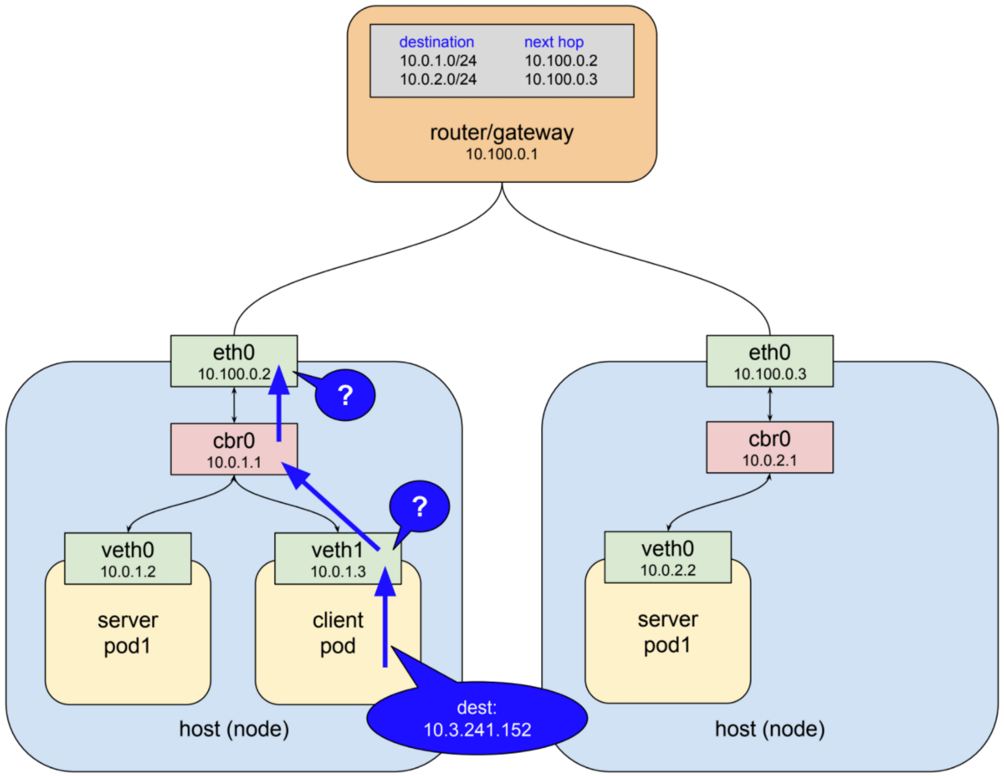
- Docker also needs the same setup
- Routing is much easier with a L2 bridge

But what about pods across 2 workers?



C2: Need for CNI

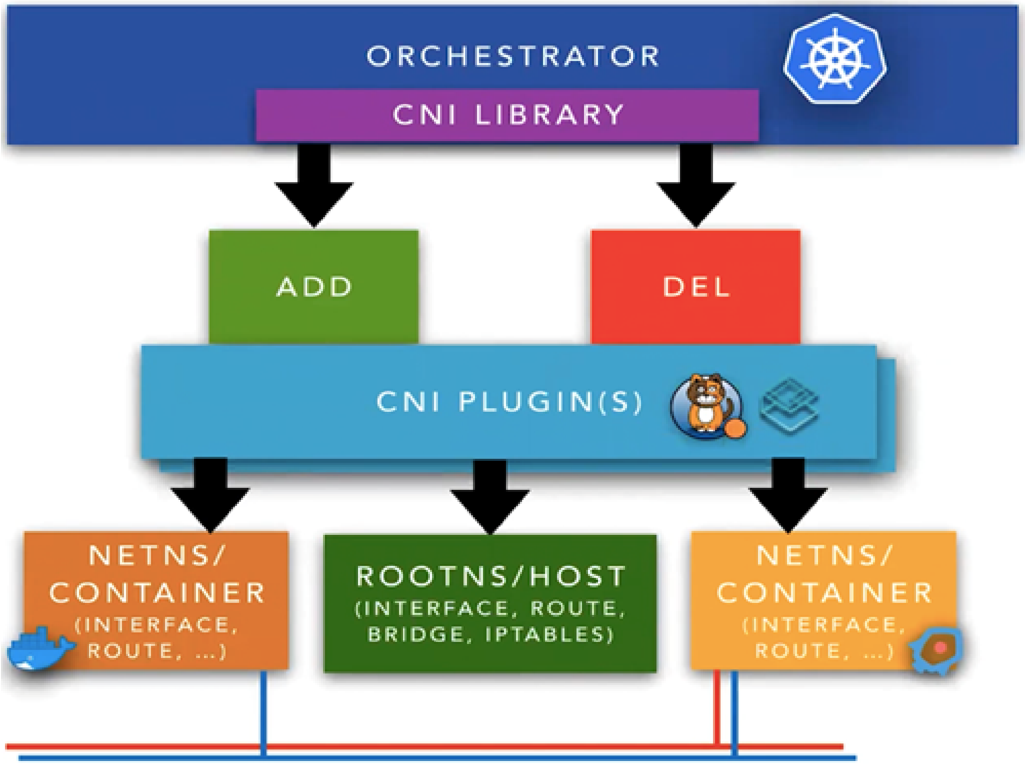
- Many different ways:
  - Tunnel overlays like VXLAN (underlying network is unaware)
  - Plain routing like BGP
- Challenges:
  - Different techniques for different clusters
  - Pods keep changing



CNI: Container Networking Interface

CNI

- Standard interface to manage container networking
- Different "plugins" that conform to the spec



Some CNI plugins

Out of the box plugins: bridge, ipvlan, macvlan, dhcp

A lot of other plugins:

cilium

flannel

CALICO

Plugin Classification

- Interface plugins: create interface for containers
- IP Address Management (IPAM) plugins: allocate IP address for a container
- Meta plugins: portmap, firewall etc

Can chain any number of plugins one after another.

Spec

Spec: https://www.cni.dev/docs/spec/

A plugin must be called with the following env variables:

- COMMAND: ADD, DEL, CHECK, VERSION
- CONTAINERID: id of the container
- NETNS: path to the net namespace of the container (eg. /run/netns/namespace1)
- IFNAME: name of intf to create inside the container
- ARGS: extra args
- PATH: where to search for the plugin

Example configuration

```
{
    "cniVersion": "1.0.0",
    "name": "dbnet",
    "type": "bridge",
    "bridge": "cni0",
    "keyA": ["some more", "plugin specific", "configuration"],
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1"
    },
    "dns": {
        "nameservers": [ "10.1.0.1" ]
    }
}
```

C2: How do we do it?

2 aspects to it:

- The CNI plugin - flannel
- Who calls the CNI plugin? (since we dont have a real k8s runtime)

cnitool

- Development tool that allows you to run cni plugins for a namespace
- See: https://www.cni.dev/docs/cnitool/

```
CNI_PATH=/opt/cni/bin
NETCONFPATH=/tmp/knetsim/<name>
cnitool add|del <name> /var/run/netns/<nsname>
```

Flannel

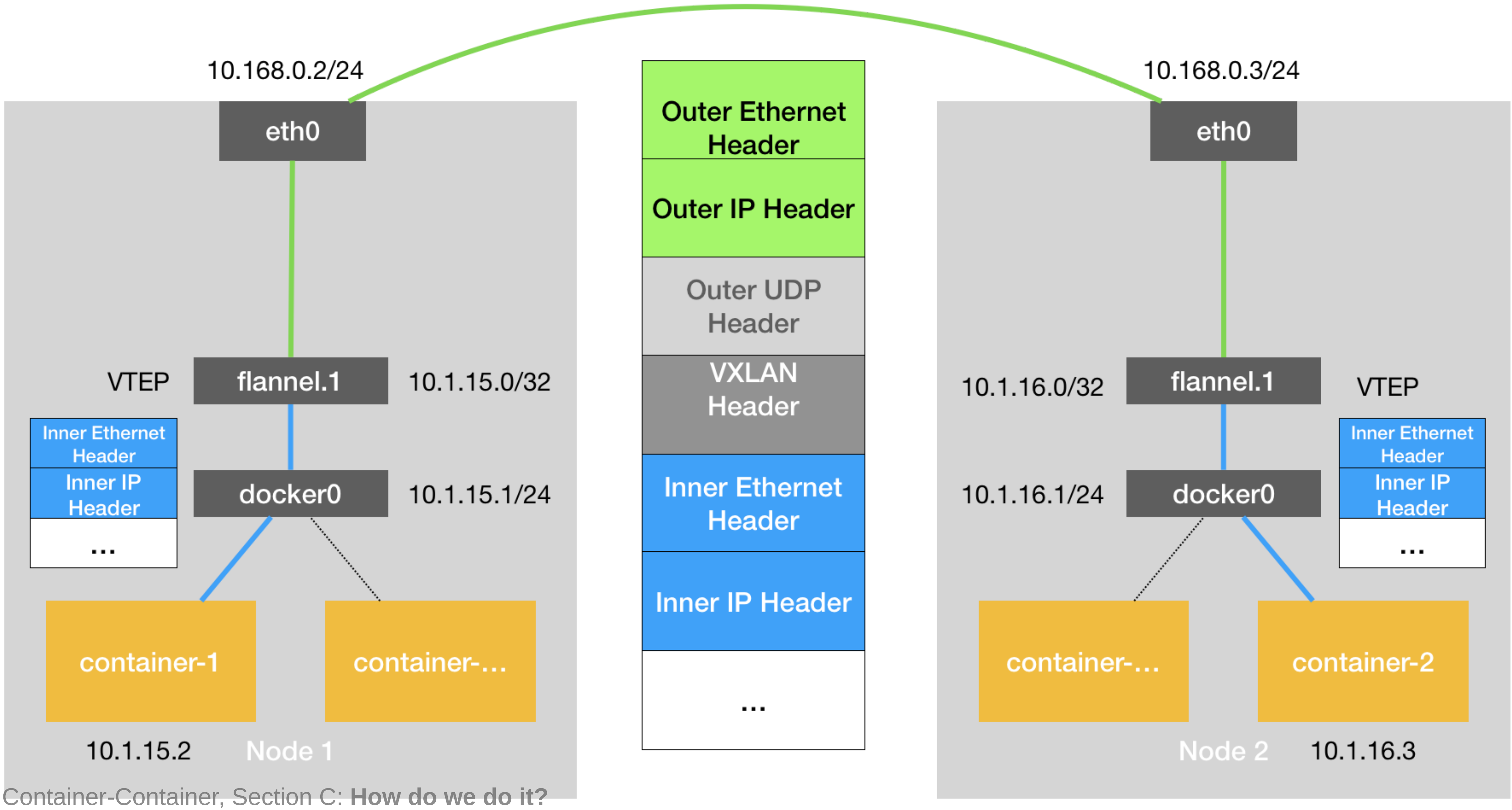
How do we run it?

- A `flanneld` binary running per worker connected to an etcd cluster for the k8s cluster
  - etcd is a key-value store
  - k8s comes with its own etcd setup
- The `flannel` CNI plugin available on each worker to be called by `cnitool` whenever containers go up/down

Startup sequence

- Cluster level configuration loaded into etcd first
- Bring up flannel daemons on each workers
- Setup CNI configuration to be used for container creation on all workers
- Create containers

Working


10.168.0.2/24
eth0
VTEP
flannel.1
10.1.15.0/32
Inner Ethernet Header
Inner IP Header
...
docker0
10.1.15.1/24
container-1
container-...
10.1.15.2
Node 1
Outer Ethernet Header
Outer IP Header
Outer UDP Header
VXLAN Header
Inner Ethernet Header
Inner IP Header
...
10.168.0.3/24
eth0
10.1.16.0/32
flannel.1
VTEP
Inner Ethernet Header
Inner IP Header
...
10.1.16.1/24
docker0
container-...
container-2
Node 2
10.1.16.3

Top level configuration

Loaded into etcd from `./conf/flannel-network-config.json`:

```
{
    "Network": "11.0.0.0/8",
    "SubnetLen": 20,
    "SubnetMin": "11.10.0.0",
    "SubnetMax": "11.99.0.0",
    "Backend": {
        "Type": "vxlan",
        "VNI": 100,
        "Port": 8472
    }
}
```

Per node generated subnet configuration

Autogenerated by the flannel daemon on each worker:

```
FLANNEL_NETWORK=11.0.0.0/8
FLANNEL_SUBNET=11.10.128.1/20
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

CNI configuration

```
{
  "name": "C0w1",
  "type": "flannel",
  "subnetFile": "/tmp/knetsim/C0w1/flannel-subnet.env",
  "dataDir": "/tmp/knetsim/C0w1/flannel",
  "delegate": {"isDefaultGateway": true}
}
```

C2: Hands on 🔨

1. Examine IPs of w1c1 and w2c2.
2. Ping w2c2 from w1c1. Note: use the `ping <ip> -c 5` command.
3. (Optional) Create a new container on one of the workers and see the IP assigned to it and check if you can connect to it.

C2: Hands on 🔨

The traffic flow here is: c1 > host bridge > vxlan tunnel > host bridge > c2

Let us trace the flow of the ping.

To obtain pid of worker processes:

```
ps aux | grep "mininet"
```

To run a command in a particular network namespace (using pid of worker nodes):

```
nsenter --net -t <pid> bash
```

To check for icmp packets on an intf:

```
tcpdump -i <intf> icmp
```

Check the bridge interfaces "flannel.100" on both hosts.

C2: Hands on 🔨

To check for packets on the vxlan port:

```
tcpdump port 8472
```

But, how to confirm if this is indeed VXLAN?

Use tshark protocol decoding:

```
tshark -V -d udp.port==8472,vxlan port 8472 |& less
```

C2: Optional Exercises

1. Examine the logs of flannel in the `/tmp/knetsim` folder.
2. Change the parameters of the flannel config in `conf` folder and re-run and see the change in IPs.

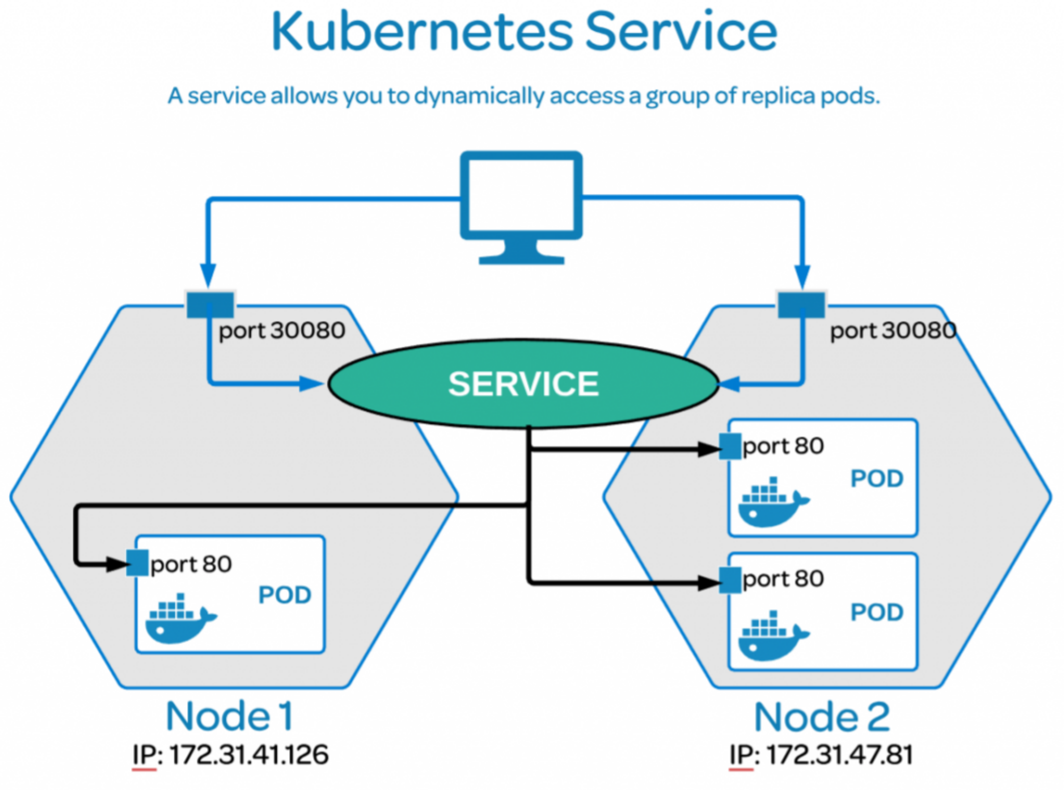
Progress so far

1. Workers and containers ✓
2. Container-Container communication ✓
3. Service abstraction ⬅
4. Multi-cluster communication

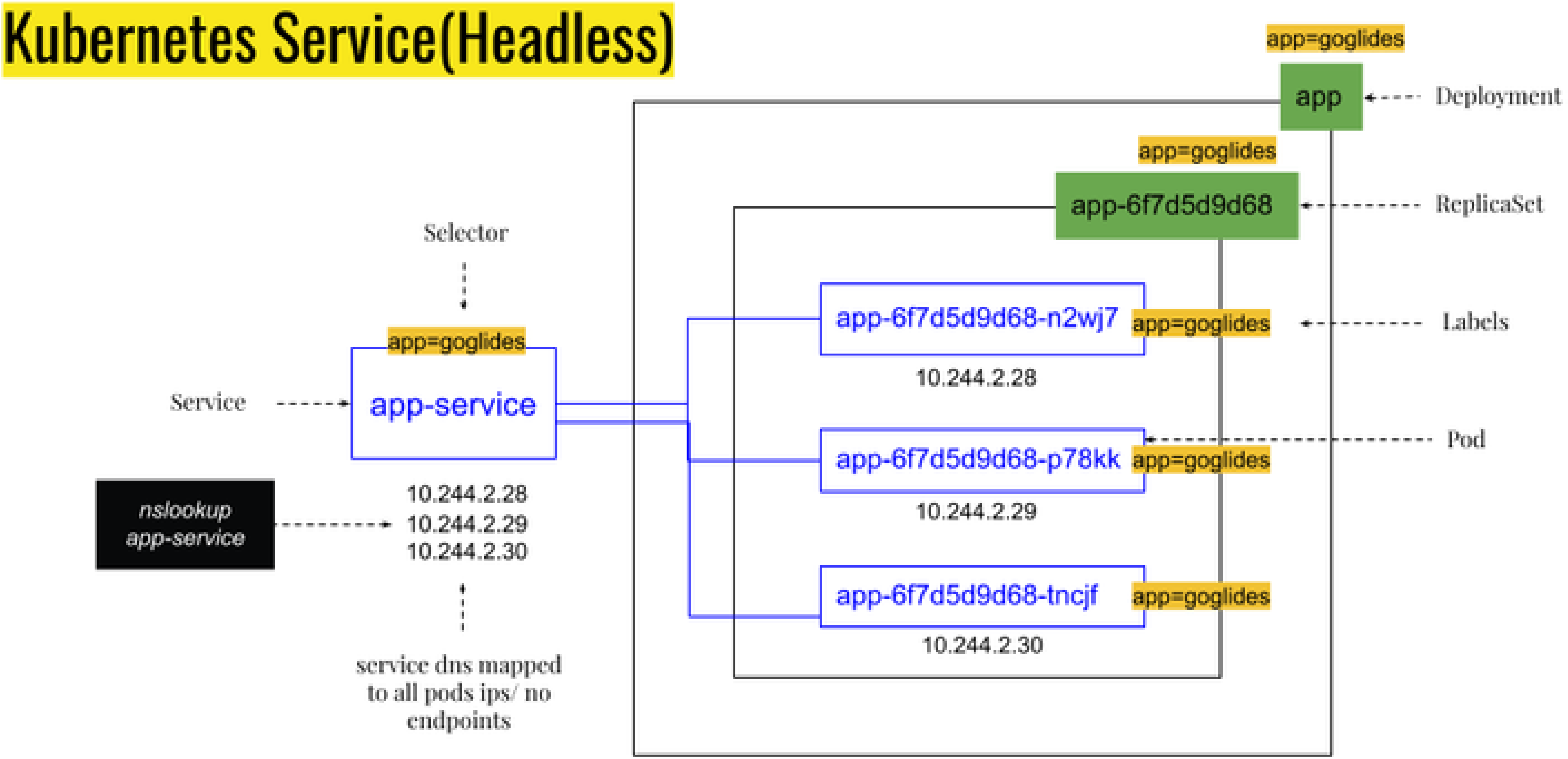
C3: Service Abstraction

Users consume services, not pods

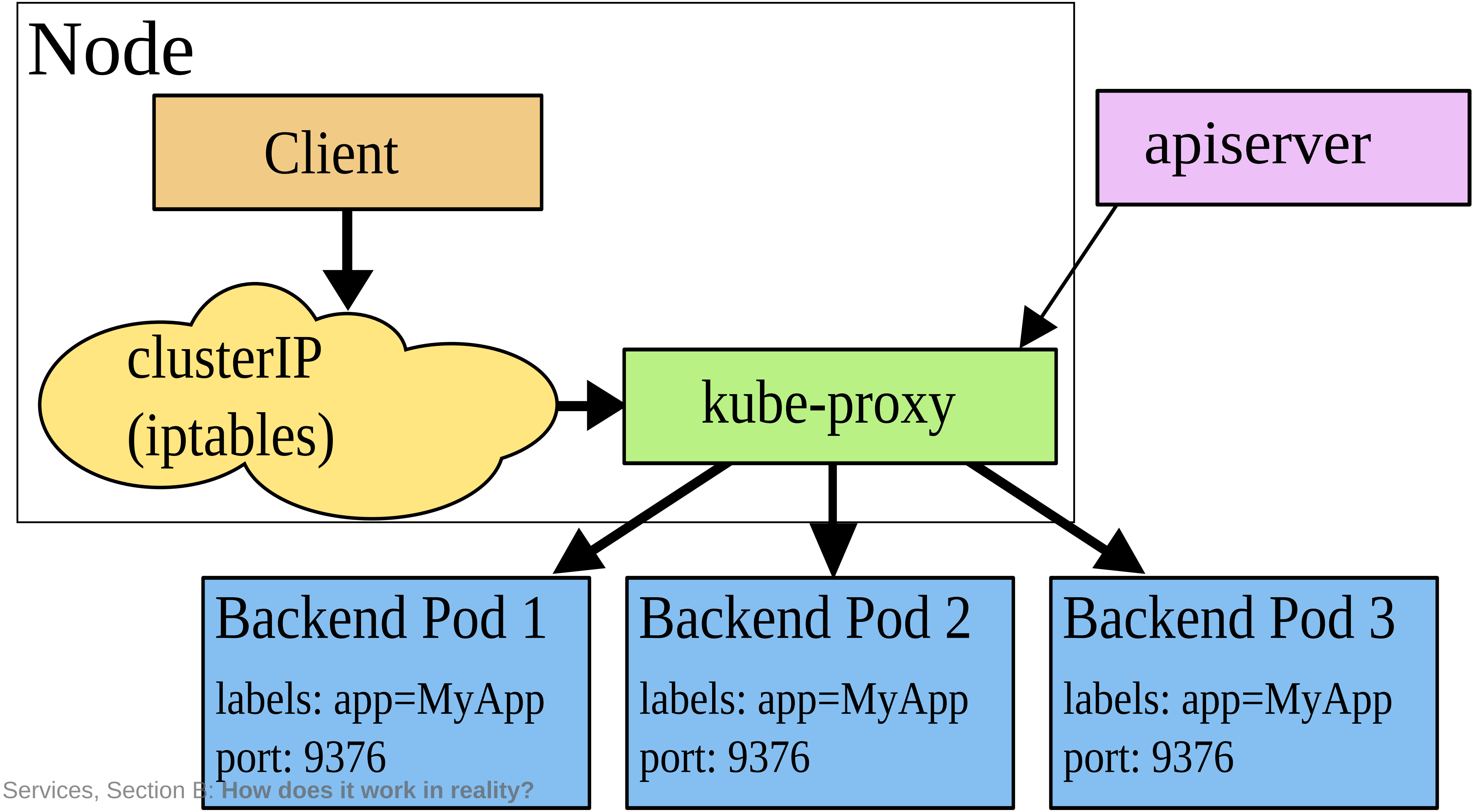
We already know about pod ips, how do services work?



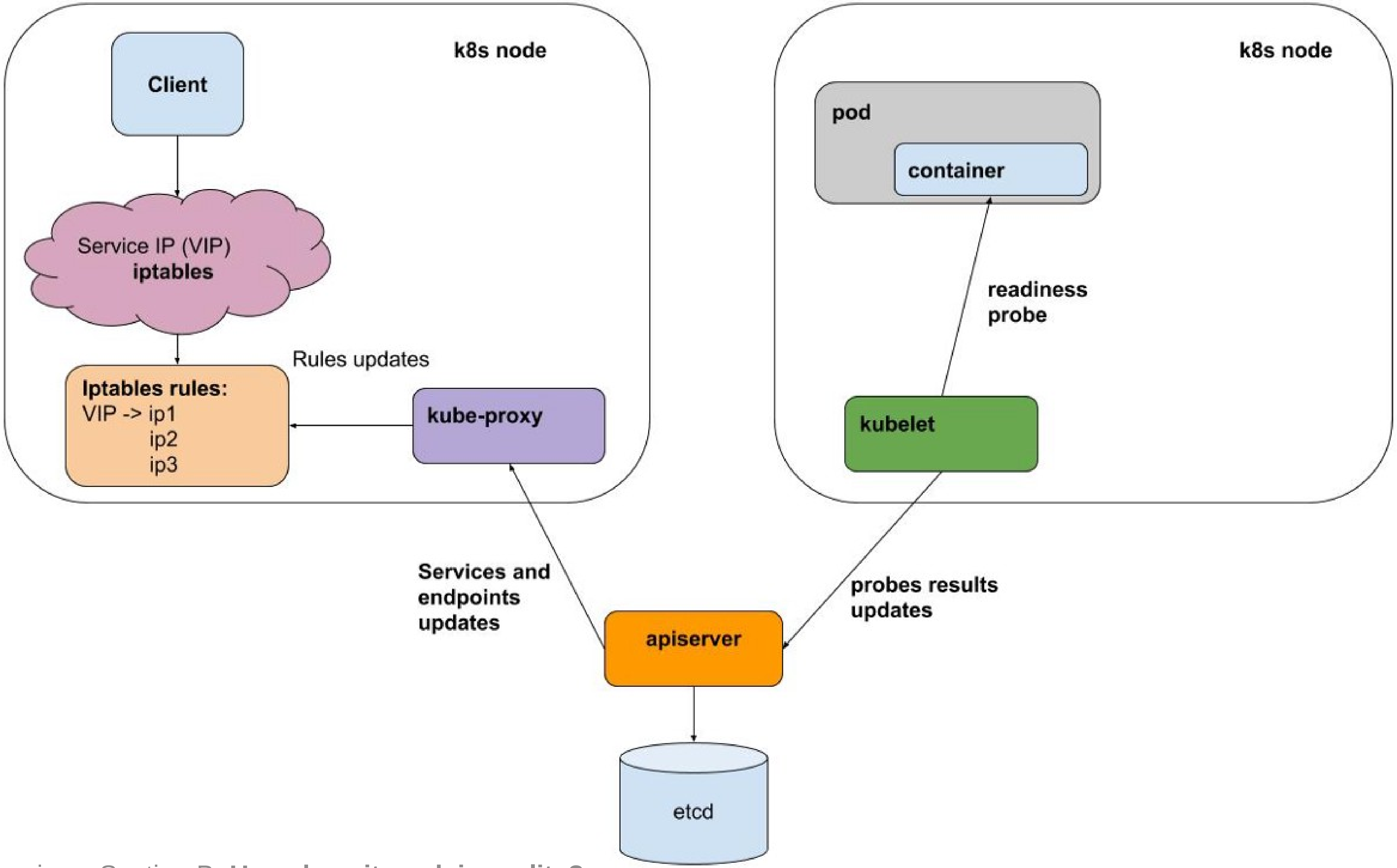
Services: using DNS



Services: using kube-proxy



Kubeproxy

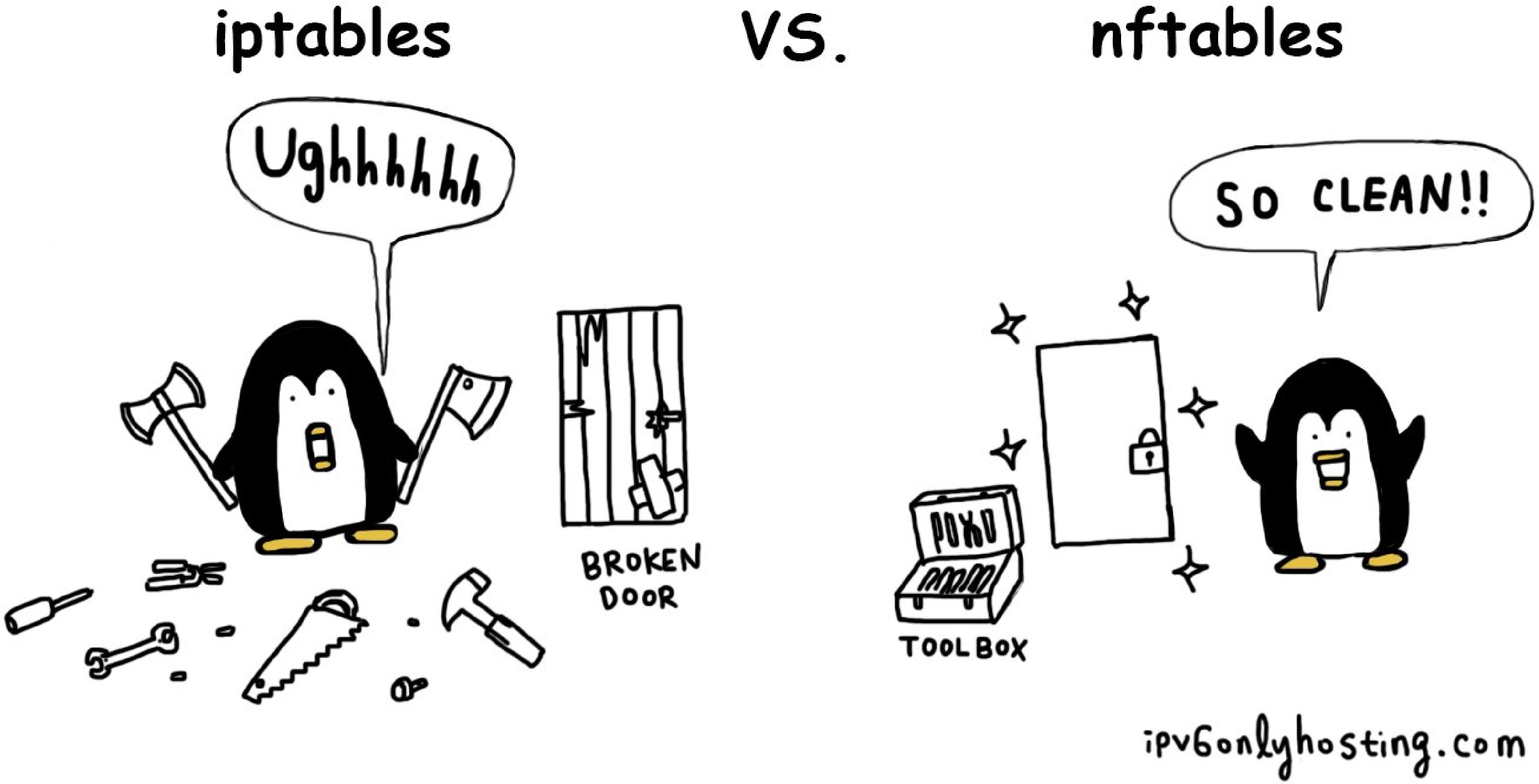

k8s node
Client
Service IP (VIP)
iptables
Rules updates
Iptables rules:
VIP -> ip1
ip2
ip3
kube-proxy
k8s node
pod
container
readiness
probe
kubelet
Services and
endpoints
updates
apiserver
probes results
updates
etcd

C3: How do we do it?

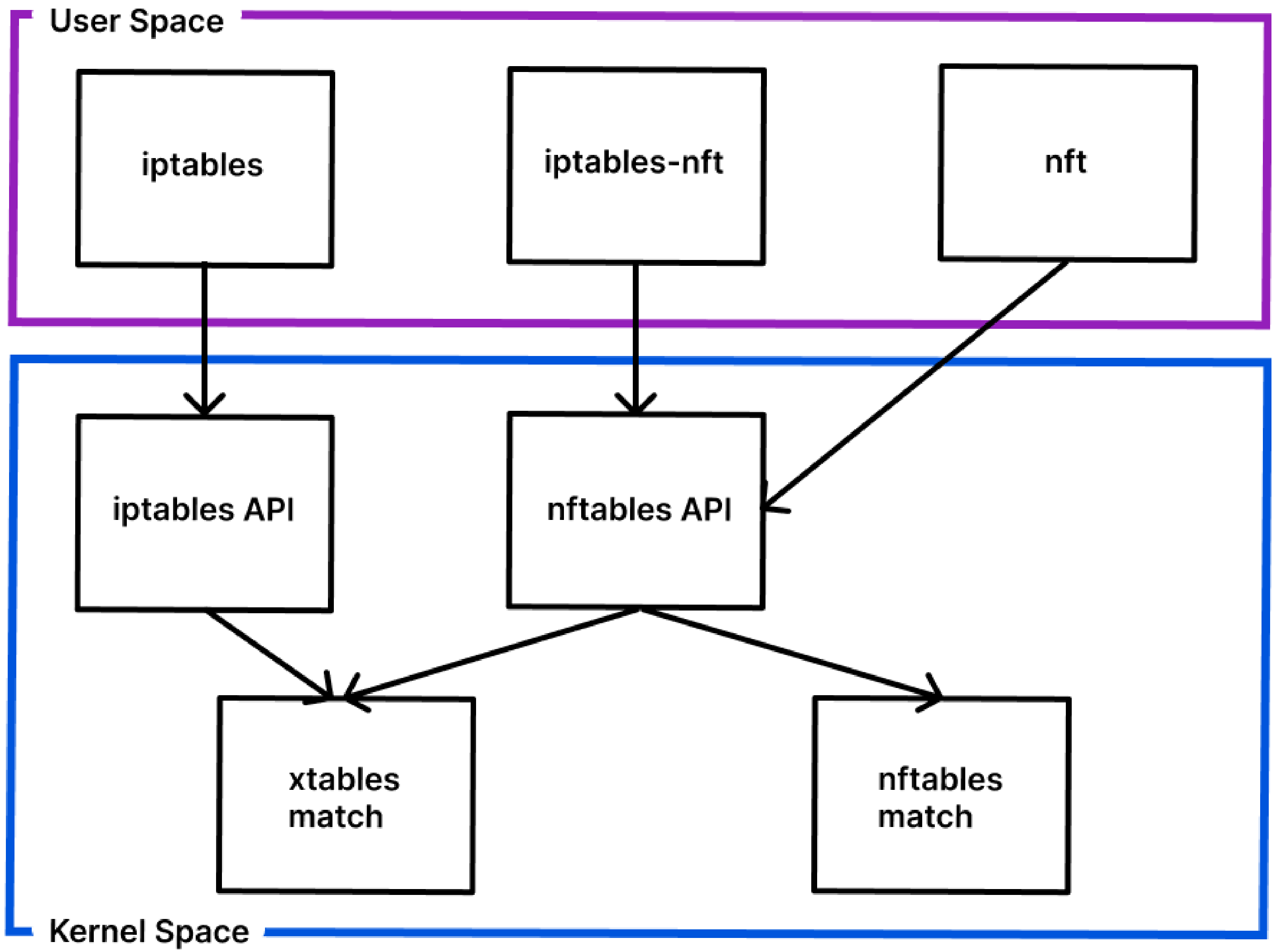
Using nftables to program static DNAT rules

Aside: nftables

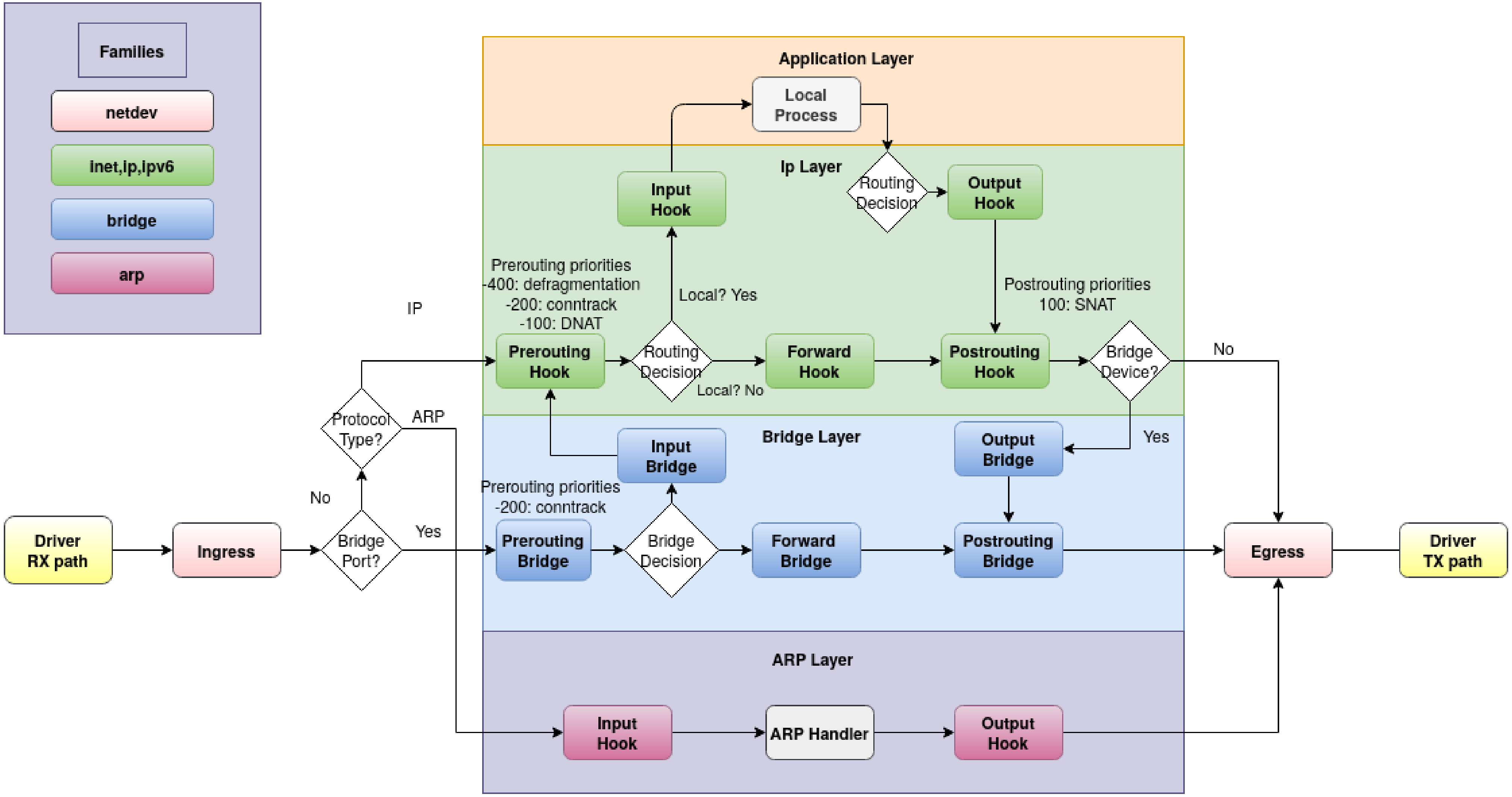
- nftables is the replacement to iptables



Landscape


User Space
iptables
iptables-nft
nft
iptables API
nftables API
xtables match
nftables match
Kernel Space

Hook points



Families
netdev
inet,ip,ipv6
bridge
arp
Application Layer
Local Process
Ip Layer
Input Hook
Routing Decision
Output Hook
Prerouting priorities
-400: defragmentation
-200: conntrack
-100: DNAT
Local? Yes
Postrouting priorities
100: SNAT
IP
Prerouting Hook
Routing Decision
Local? No
Forward Hook
Postrouting Hook
Bridge Device?
No
Protocol Type?
ARP
Bridge Layer
Input Bridge
Output Bridge
Yes
No
Prerouting priorities
-200: conntrack
Driver RX path
Ingress
Bridge Port?
Yes
Prerouting Bridge
Bridge Decision
Forward Bridge
Postrouting Bridge
Egress
Driver TX path
ARP Layer
Input Hook
ARP Handler
Output Hook

Mini `nftables` tutorial

Let us run a new container for this experiment:

```
docker run -it --privileged --rm --entrypoint bash knetsim
```

Check the rules (will be empty):

```
nft list ruleset
```

Use this reference: https://wiki.nftables.org/wiki-nftables/index.php/Main_Page

(Go to the Basic Operations section)

Creating a table

```
nft add table ip table1
nft list ruleset # or
nft list table table1
```

Table family types: ip, arp, ip6, bridge, inet, netdev

Creating a chain

```
nft add chain ip table1 chain1 { type filter hook output priority 0 \; policy accept \; }
nft list ruleset
```

- ip refers to the table family (can be omitted)
- table1 refers to the table we just created
- chain1 is the name of the new chain
- type is one of `filter`, `route` or `nat`

Creating a rule

```
nft add rule ip table1 chain1 ip daddr 8.8.8.8 counter
nft list ruleset
```

- You can match based on anything in the packet. Check: https://wiki.nftables.org/wiki-nftables/index.php/Quick_reference-nftables_in_10_minutes#Matches
- `counter` is a statement

What can you do with Rule statements?

- Verdict statements: `accept`, `drop`, `queue` (to userspace), `continue`, `return`, `jump`, `goto`
- `counter`
- `limit`: rate limiting
- `nat`: dnat to or snat to

Refer to: https://wiki.nftables.org/wiki-nftables/index.php/Quick_reference-nftables_in_10_minutes#Statements

Testing our rule

Counters should be empty:

```
nft list ruleset
```

Send a single packet:

```
ping -c 1 8.8.8.8
```

Counter should now be incremented:

```
nft list ruleset
```

Cleaning up

Using handles to delete rules

```
nft -a list ruleset
nft delete rule ip table1 chain1 handle #handleno
nft list ruleset
```

```
nft delete chain ip table1 chain1
nft delete table ip table1
```

Exit the container.

Other useful features

- Intervals: `192.168.0.1-192.168.0.250`, `nft add rule filter input tcp ports 1-1024 drop`
- Concatenations (`.` syntax)
- Math operations: hashing, number series generators
- Sets and Maps (with Named variants)
- Quotas: a rule which will only match until a number of bytes have passed
- Flowtables: net stack bypass

Nft summary

- A better `iptables`
- tables -> chains -> rules
- Chains have to be created for a purpose: `filter`, `route` or `nat`
- Rules can have statements:
  - Verdict statements: `accept`, `drop`, `jump` etc
  - `counter`, `limit`, `nat`
- A lot of other handy features

C3: Hands on

In the code, we have:

```
C0.kp_vip_add("100.64.10.1", ["c2", "c3"])
```

This means, we can access the containers c2 and c3 using this VIP.

1. Ping this virtual ip from c1 and see that it works.
2. Delete one of the 2 containers c2/c3 and see what happens. (Hint: repeat the ping a few times)

C3: Exercises 🔨

1. Create a few containers and expose them using a new VIP.
2. Examine the `nft` rules added to the hosts.

Progress so far

1. Workers and containers ✓
2. Container-Container communication ✓
3. Service abstraction ✓
4. Multi-cluster communication ⬅️

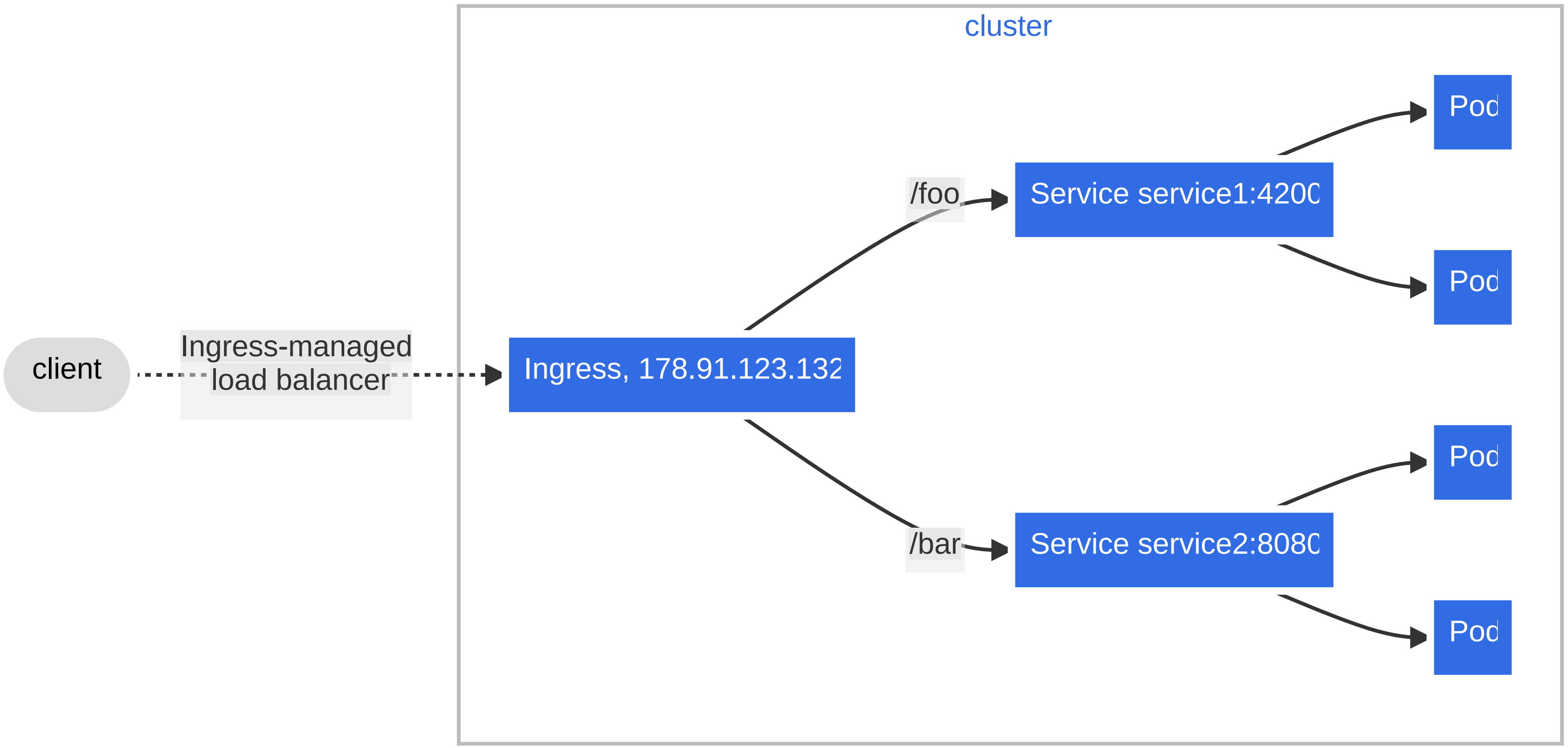
C4: Multi-cluster communication

We have seen how containers and services within a cluster communicate.

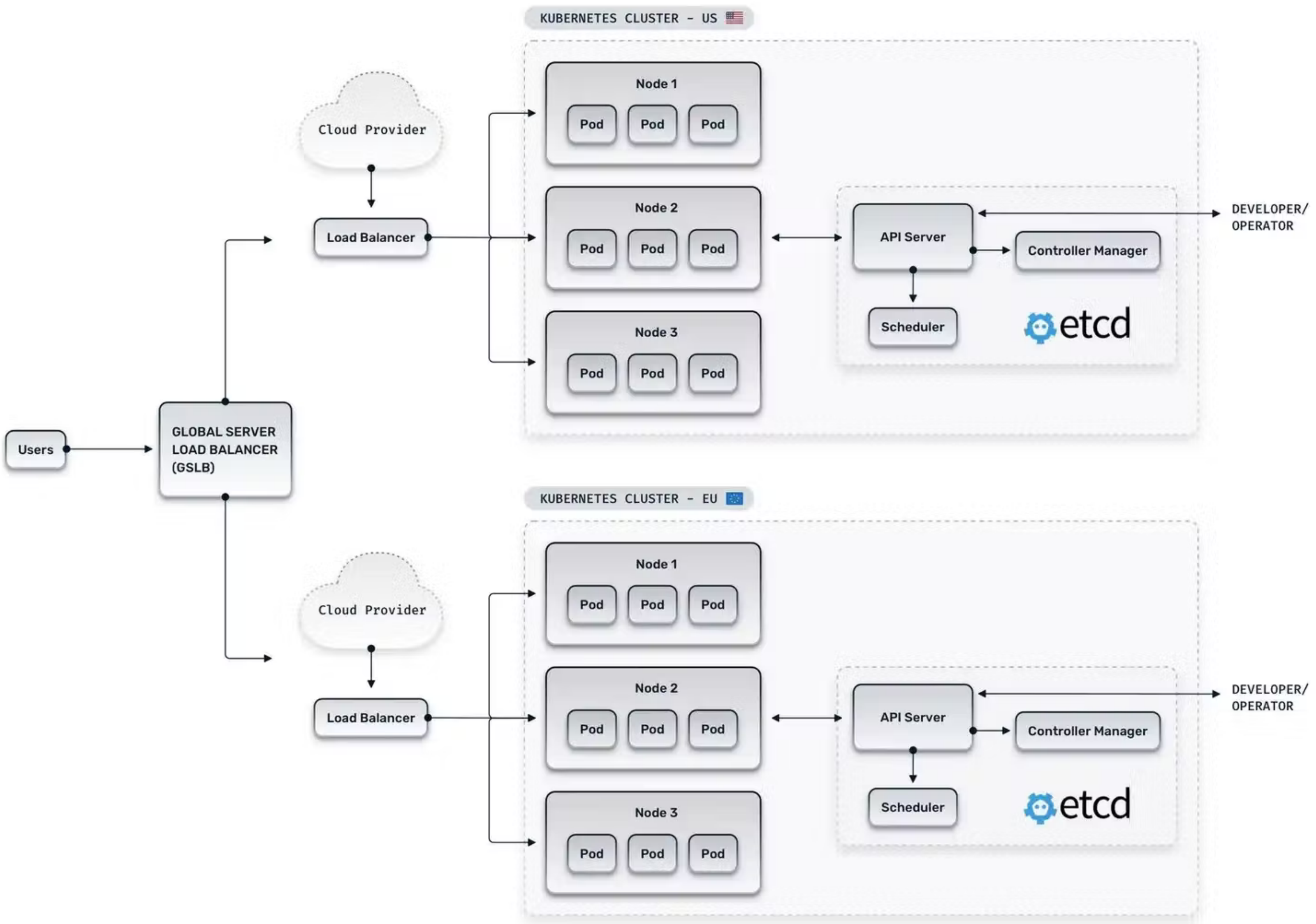
What about services across clusters?

Exposing services

- Ingress



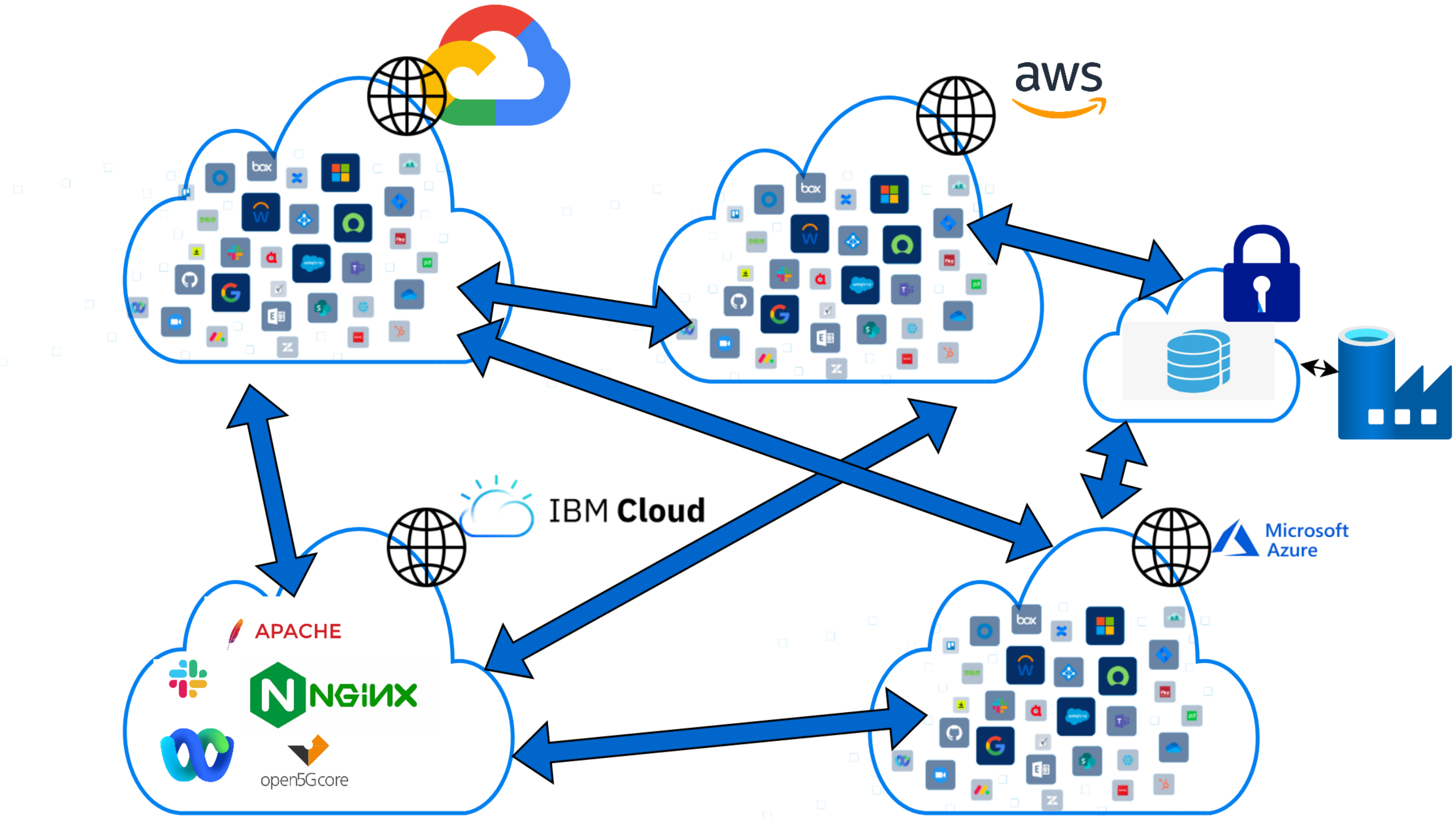
Sharded workloads



Why Multi-cluster?

- Same provider: different zones, data centers
- Different providers: multi-cloud, hybrid-cloud, edge/telco clouds

Multi-cloud



Hybrid-cloud

- Connect public-cloud deployments with on-prem enterprise data centers
    - Allows cloud transformation journey
- Allows mix and match of legacy applications with new apps
- Allows moving workloads from on-prem to clouds (and vice-versa) as situation changes

Edge clouds

Multi-cluster Networking Requirements

- Allow containers/services to talk across clusters
- Features:
  - Routing: how to find the pathway from a container to another
  - Management: adding/removing clusters, pods, services
  - Security: encryption of cross-cluster traffic
  - Policies: which apps can talk to which other apps
- These are features required for within cluster too - but a number of solutions exist for this

C4: How does it work?

It doesn't yet!

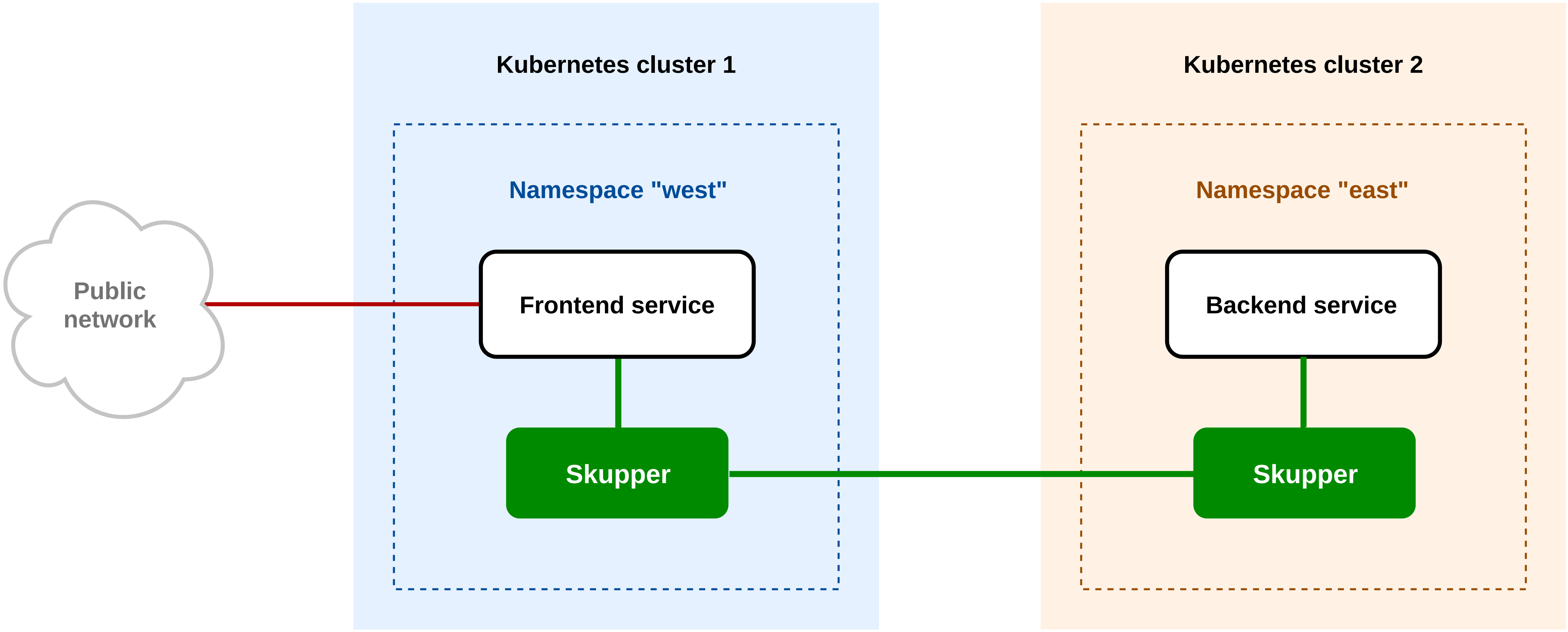
- Active area of research - not solved yet
- Some existing solutions: Cilium Multi-cluster, Istio/Consul Multi-cluster, Submariner, Skupper
- Each solution trade-offs various aspects
- IBM is working on a product

Skupper

- Layer 7 service interconnect
- Open-source project: skupper.io

Let's go into some details.

Skupper Overview



Skupper Usage

Linking sites:

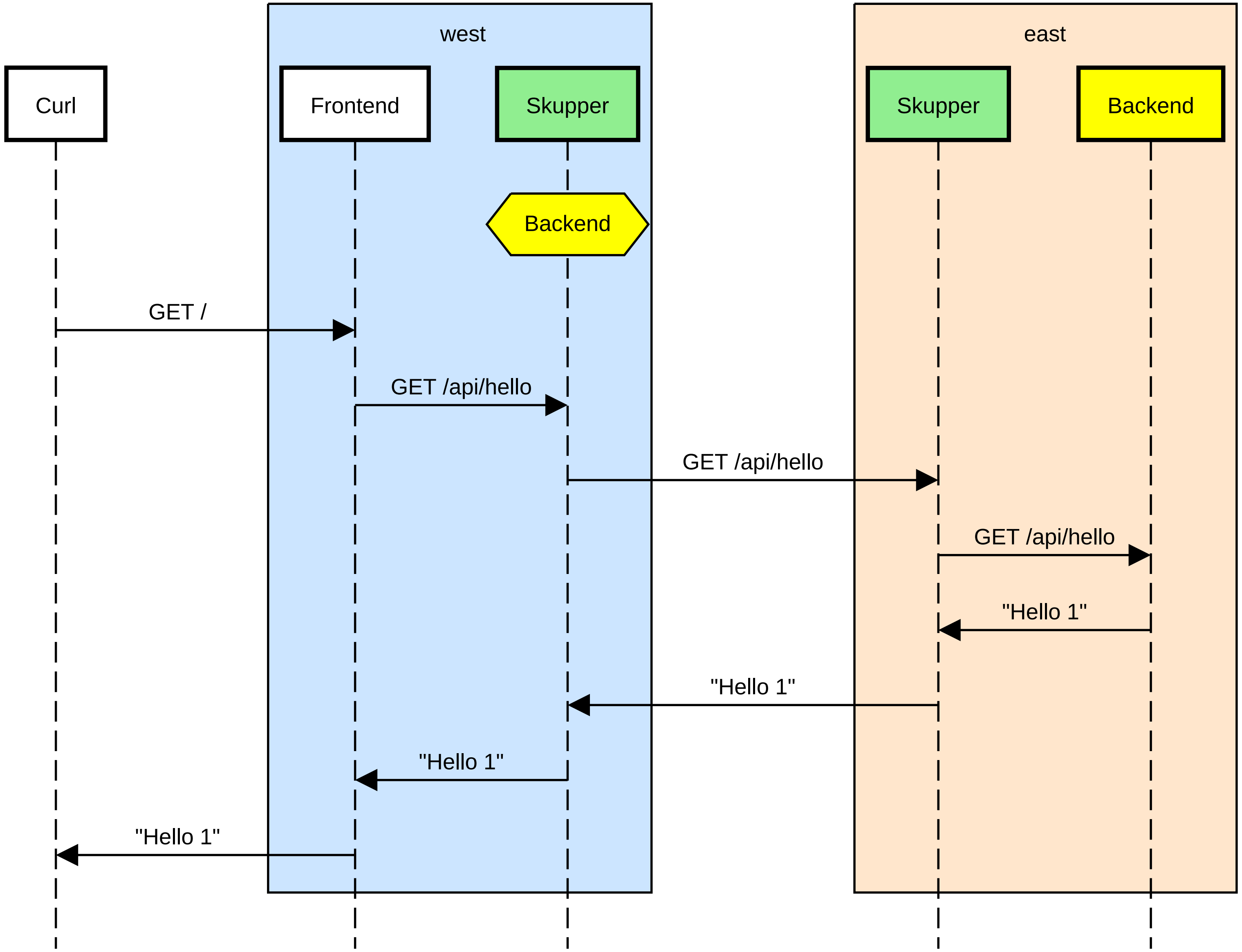
```
skupper init
# on one end
skupper token create site1.token
# on the other end
skupper link create site1.token
```

Exposing a service:

```
skupper expose deployment/backend --port 8080
```

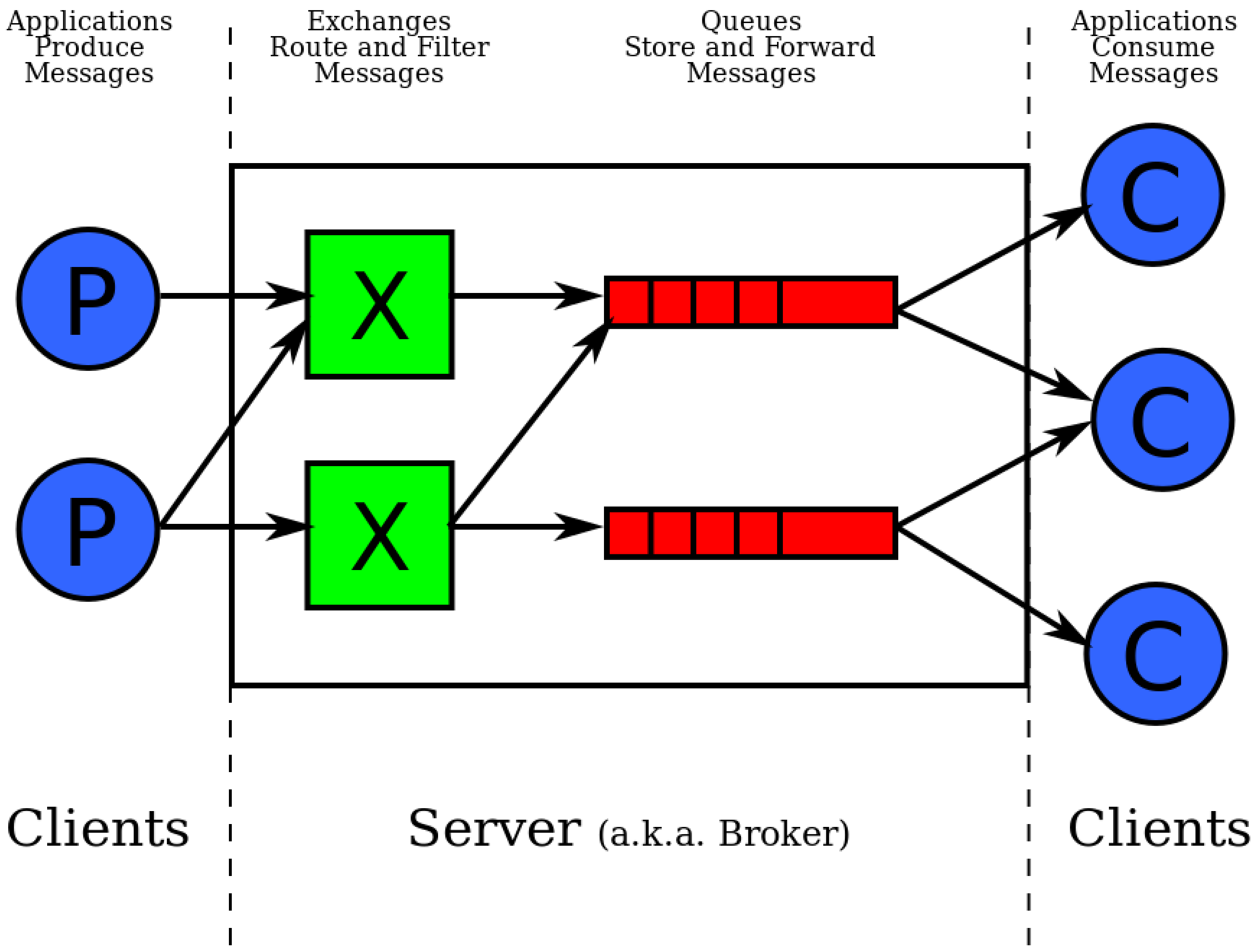
Message pattern

- Any number of services communicate over the same skupper router



How does it work?

- Based on the AMQP Messaging queue system
- Skupper Routers are modified Qpid Dispatch Routers



Skupper in Kubernetes

A bit more complicated:

- Skupper Router to do the actual routing
- Site-Controller: to manage the links between various clusters
- Service-Controller: to manage services as they come up/down

C4: How do we do it?

We also use skupper-router, with manual configuration.

Let us look at how the config file looks like.

(Example is available in `conf/` folder as `solo0.json` and `solo1.json`)

Left side

```
        "router", {
            "id": "c0",
            "mode": "interior",
            ...
        }
```

```
        "listener", {
            "name": "interior-listener",
            "role": "inter-router",
            "port": 55671,
            "maxFrameSize": 16384,
            "maxSessionFrames": 640
        }
```

Right side

```
        "router",
        {
            "id": "c1",
            "mode": "interior",
            ...
        }
```

```
        "connector",
        {
            "name": "link1",
            "role": "inter-router",
            "host": "localhost",
            "port": "55671",
            "cost": 1,
            "maxFrameSize": 16384,
            "maxSessionFrames": 640
        }
```

Service Management

```
        "tcpListener",
        {
            "name": "backend:8080",
            "port": "1028",
            "address": "backend:8080",
            "siteId": "c0"
        }
```

```
        "tcpConnector",
        {
            "name": "backend",
            "host": "localhost",
            "port": "8090",
            "address": "backend:8080",
            "siteId": "c1"
        }
```

How do we do it?

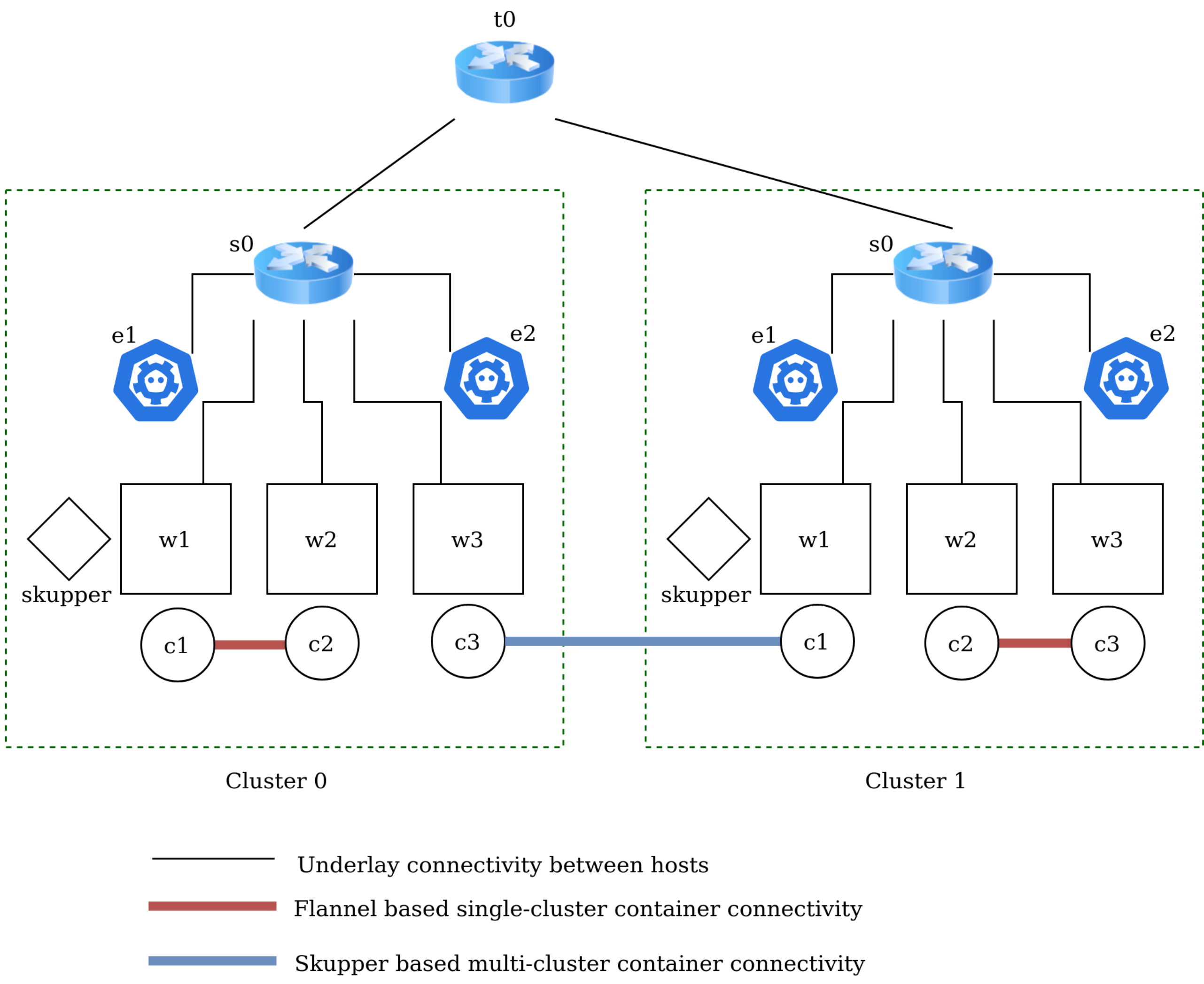
1. Generate conf json files based on user provided list of services to expose.
2. Run one `skupper-router` in each cluster on worker0.
3. Containers in the cluster should connect to the skupper-router at a particular port for a given service.

C4: Hands on

Go read line number 79-94 in `main.py`.

Understand and reproduce it.

Examine the generated conf files in `/tmp/knetsim/skupper` folder.



Skupper: benefits and limitations

Pros:

- Allow any number of services to talk across clusters with a single port exposed
- Provides encryption for cross-service links

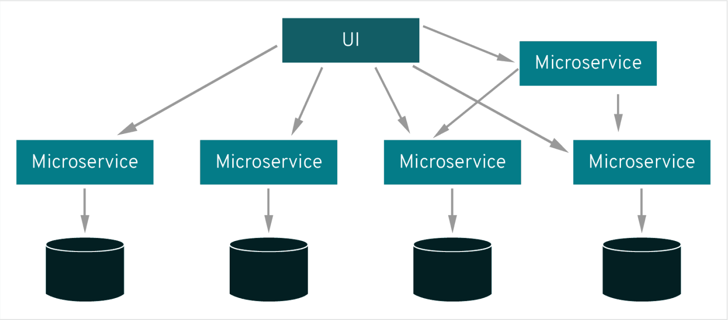
Cons:

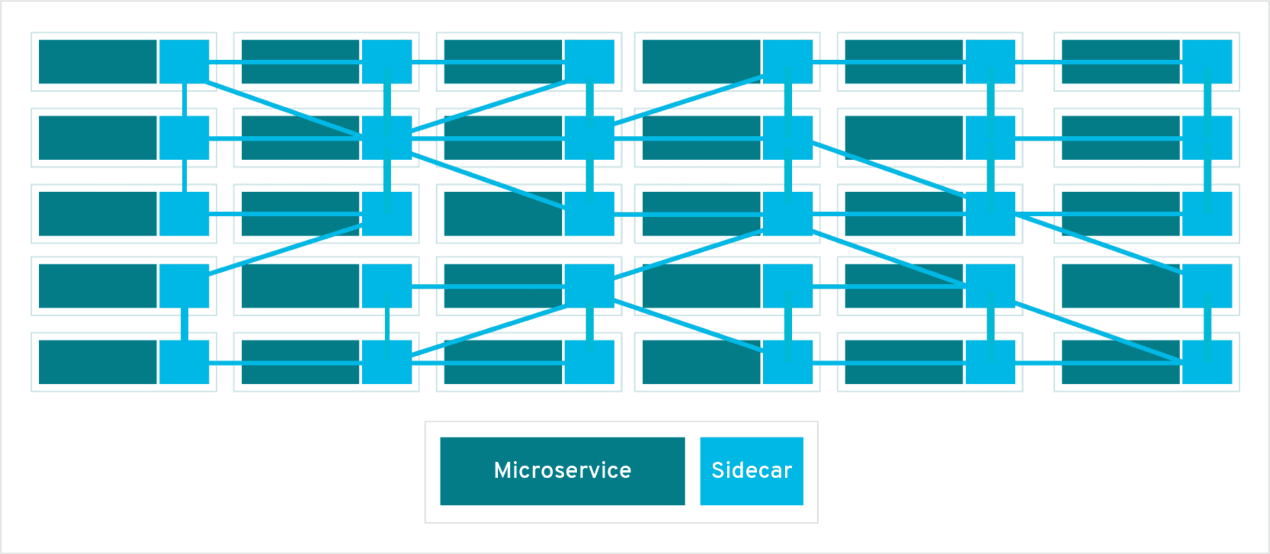
1. Not efficient: in terms of performance overheads
2. No provision for fine-grained policies

There are other solutions in the market (including our own!)

Bonus round: Service Meshes

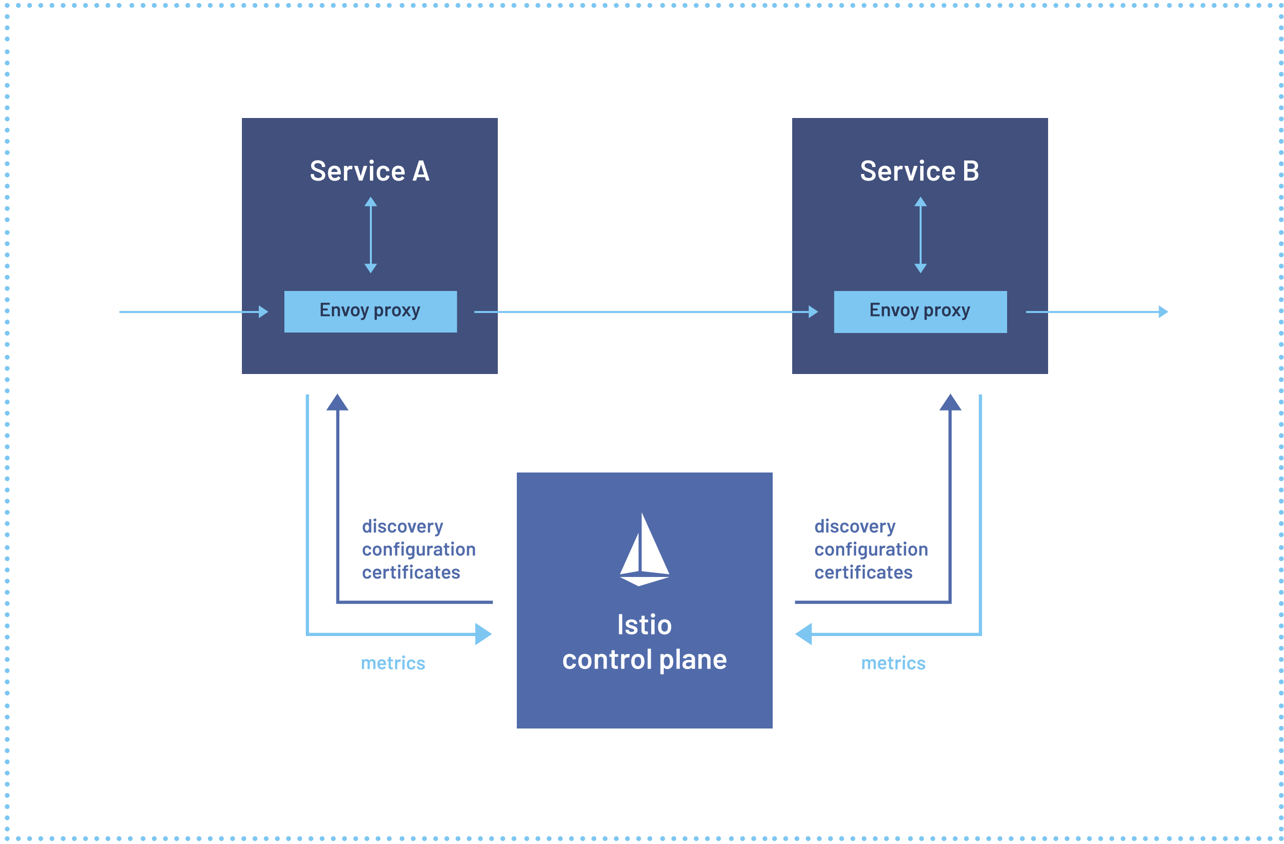
- Mesh of services
- Sidecars to manage service-service communication
- Handle:
  - service discovery
  - TLS certificate issuing
  - metrics aggregation





Istio Service Mesh

- load balancing, service-to-service authentication, and monitoring – with few or no service code changes



Multi-cluster Istio

(https://istio.io/v1.2/docs/concepts/multicluster-deployments/)

- Multiple control plane topology
- Single-network shared control plane
- Multi-network shared control plane: using istio gateways
- Summary: tight coupling across clusters

Retrospective

What did we learn today?

1. About the various layers in Kubernetes networking:

- Network namespaces
- Pod-pod communications: CNI spec and CNI plugins
- Service abstraction using `kube-proxy`

2. Intro to the world of multi-cluster networking

- Saw how Skupper works in connecting clusters

What next?

1. Setup and configure your own kubernetes clusters more confidently
2. Be able to compare and select CNI plugin solutions like Calico, Cilium
3. Extend CNI functionalities
4. Understand the CNCF networking landscape and how things fit with each other
5. Plan multi-cluster/edge deployments

And of course,

Conduct research in the exciting space of cloud-native networking

References

- https://kubernetes.io/docs/
- https://www.redhat.com/architect/multi-cluster-kubernetes-architecture
- https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/