SIG-Network Intro & Deep-dive

Bowei Du <@bowei>
Rich Renner <@eth0xfeed>
Tim Hockin <@thockin>

Part 1: Intro

- An overview of the SIG and the “basics”
- If you are new to Kubernetes or not very familiar with the things that our SIG deal with - this is for you!

Part 2: Deep-dive

- A deeper look at some of the newest work that the SIG has been doing
- If you are already comfortable with Kubernetes networking concepts, and want to see what’s next - this is for you!

Part 1: Intro

What, When, Where

Responsible for the Kubernetes network components

- Pod networking within and between nodes
- Service abstractions
- Ingress and egress
- Network policies and access control

Zoom meeting: Every other Thursday, at 21:00 UTC
Slack: #sig-network (`slack.k8s.io`)
https://git.k8s.io/community/sig-network

(Don’t worry, we’ll show this again at the end)

APIs

Service, Endpoints, EndpointSlice

- Service registration & discovery

Ingress

- L7 HTTP routing

Gateway

- Next-generation HTTP routing and service ingress

NetworkPolicy

- Application “firewall”

Components

Kubelet CNI implementation

- Low-level network drivers and how they are used

Kube-proxy

- Implements Service API

Controllers

- Endpoints and EndpointSlice
- Service load-balancers
- IPAM

DNS

- Name-based discovery

Networking model

All Pods can reach all other Pods, across Nodes

Sounds simple, right?

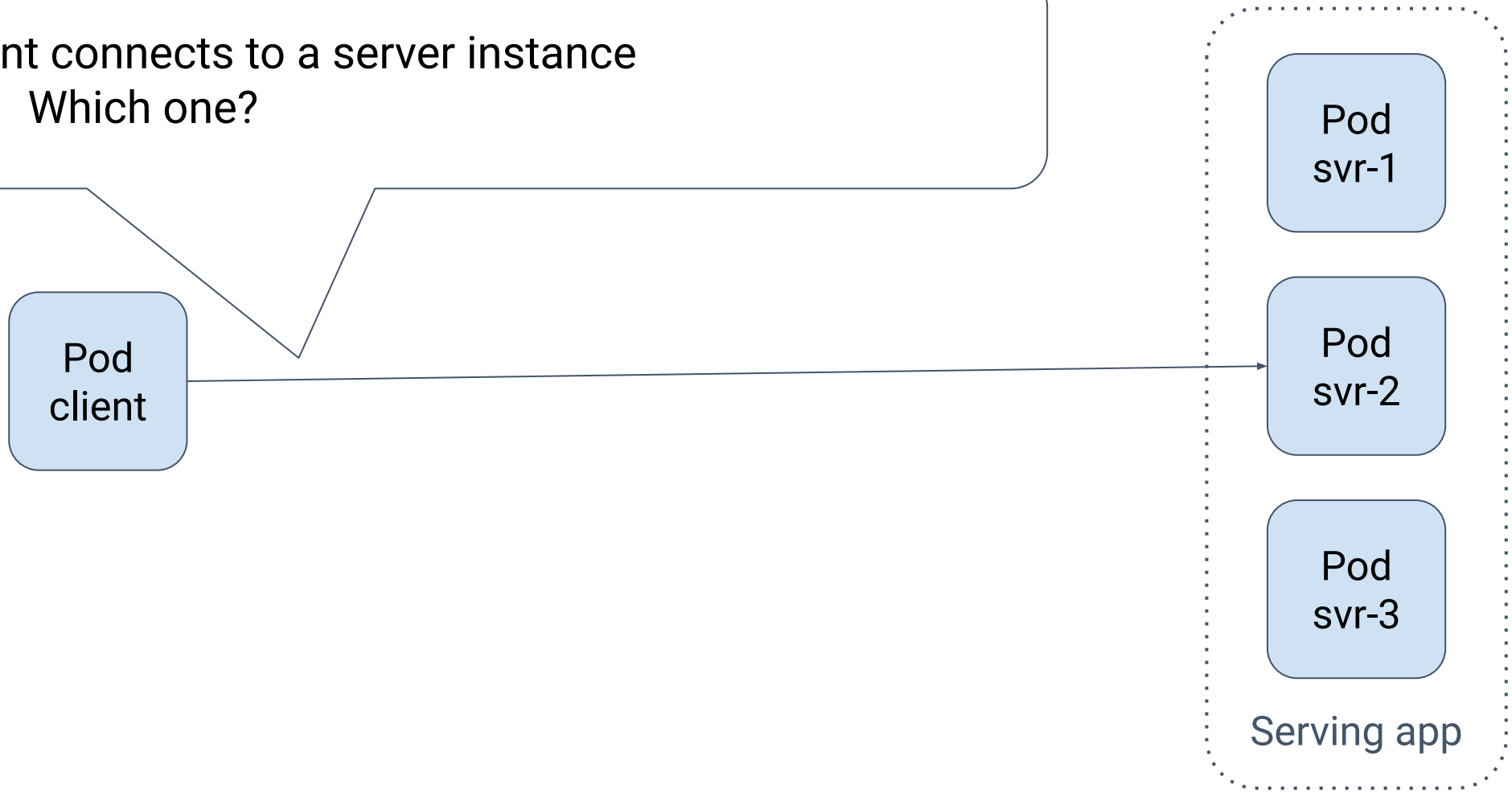
Many implementations

- Flat
- Overlays (e.g. VXLAN)
- Routing config (e.g. BGP)

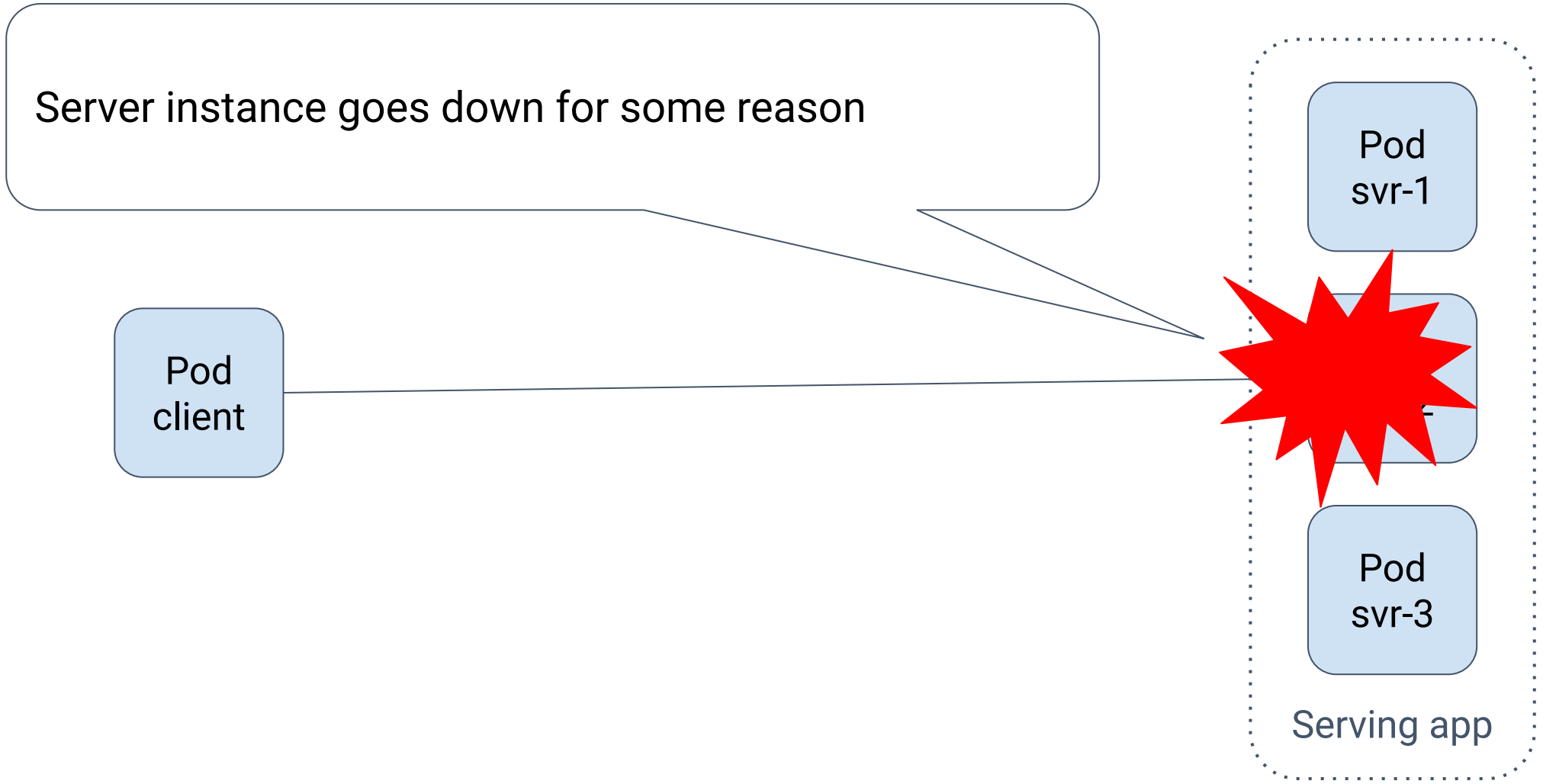
One of the more common things people struggle with

Services: problem

Client connects to a server instance
- Which one?



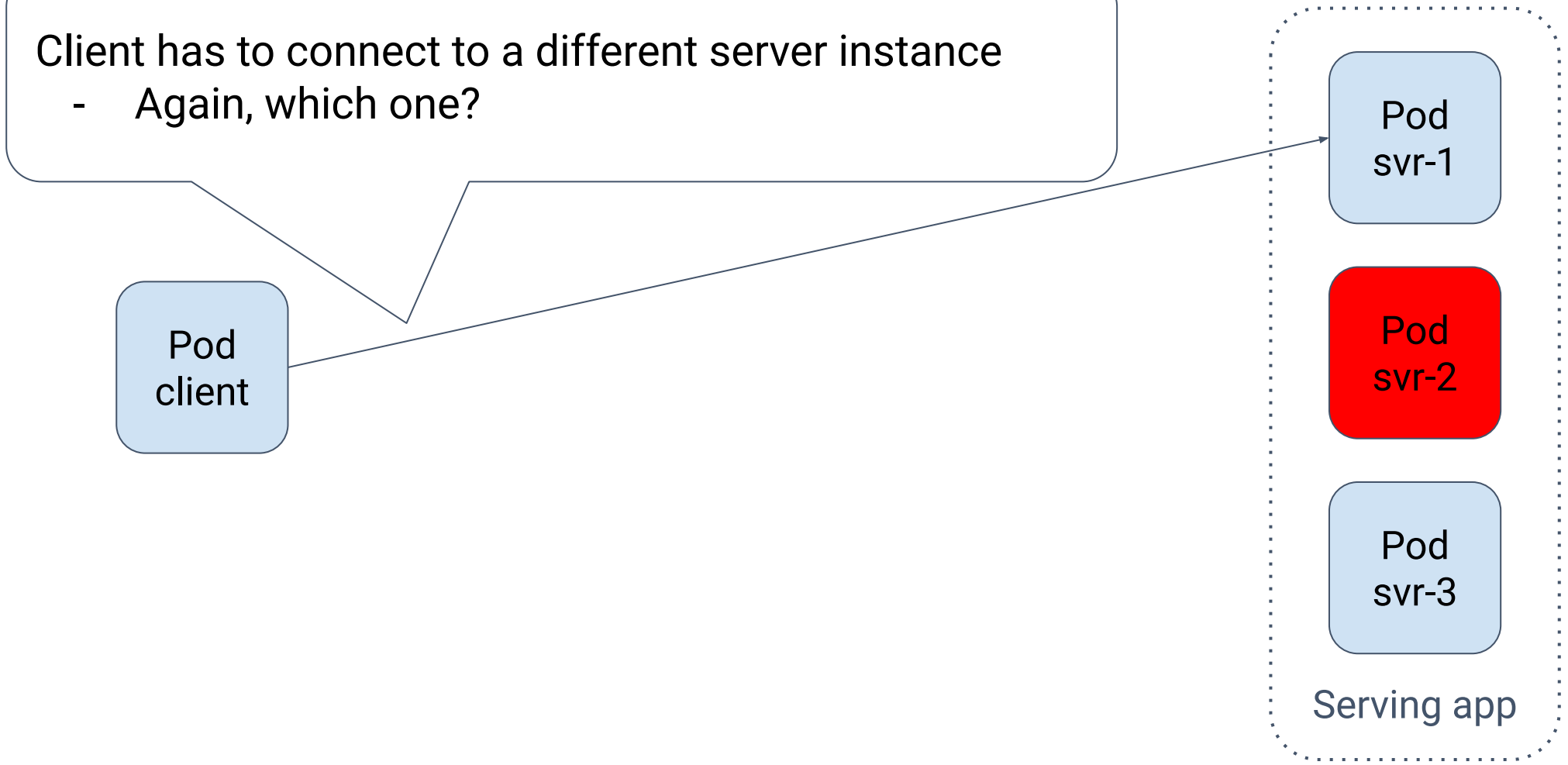
Services: problem



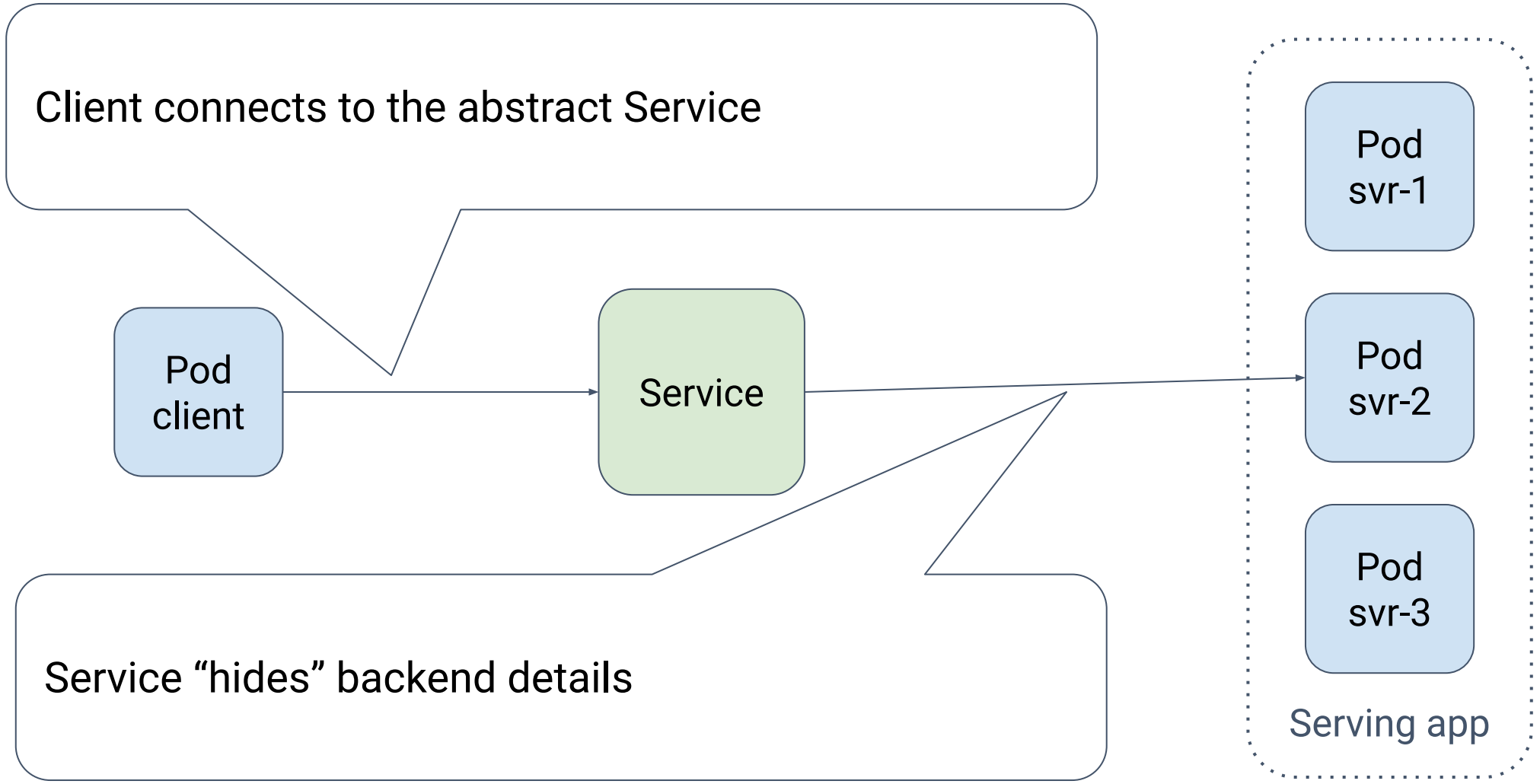
Services: problem

Client has to connect to a different server instance

- Again, which one?



Services: abstraction



Services

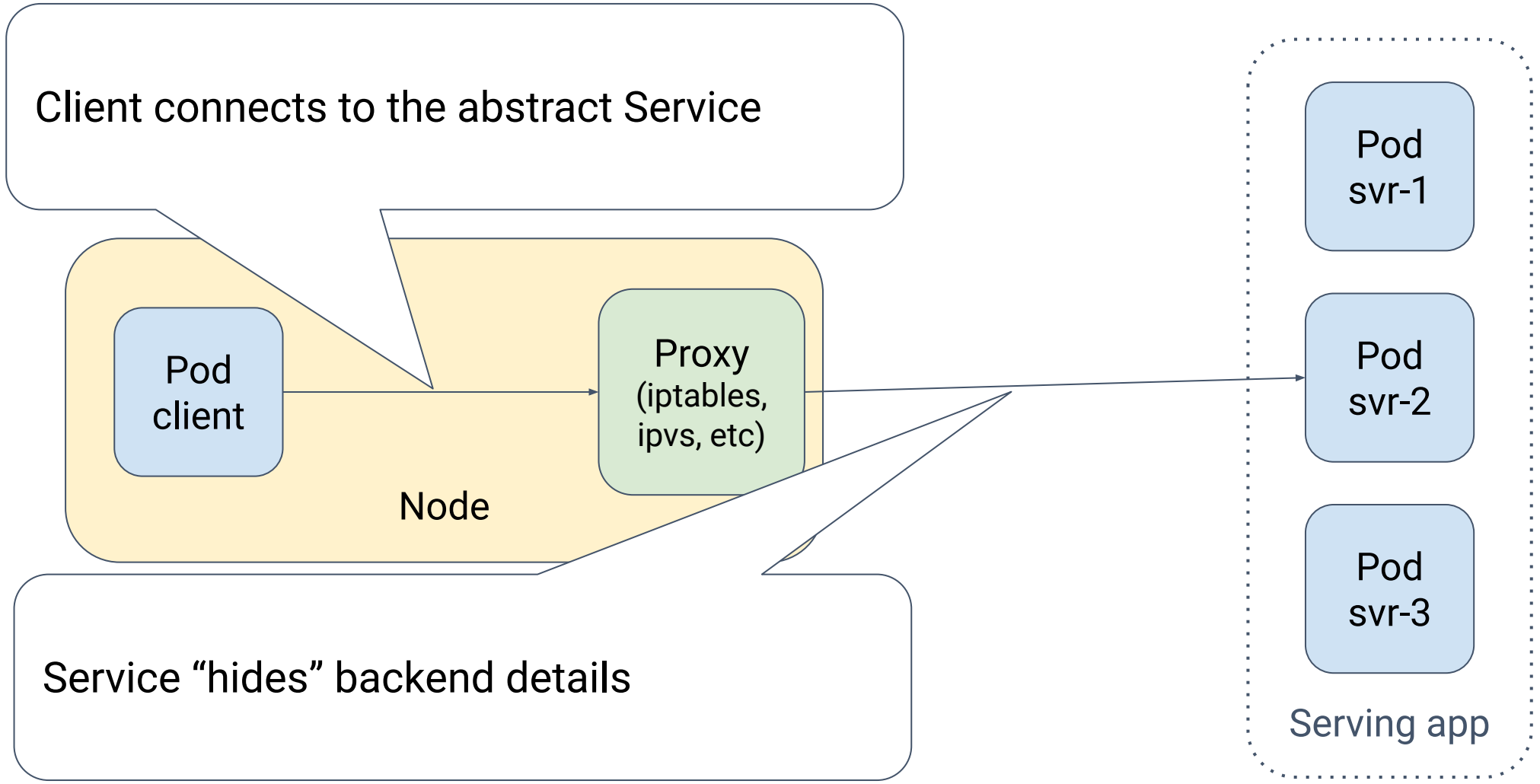
Pod IPs are ephemeral

“I have a group of servers and I need clients to find them”

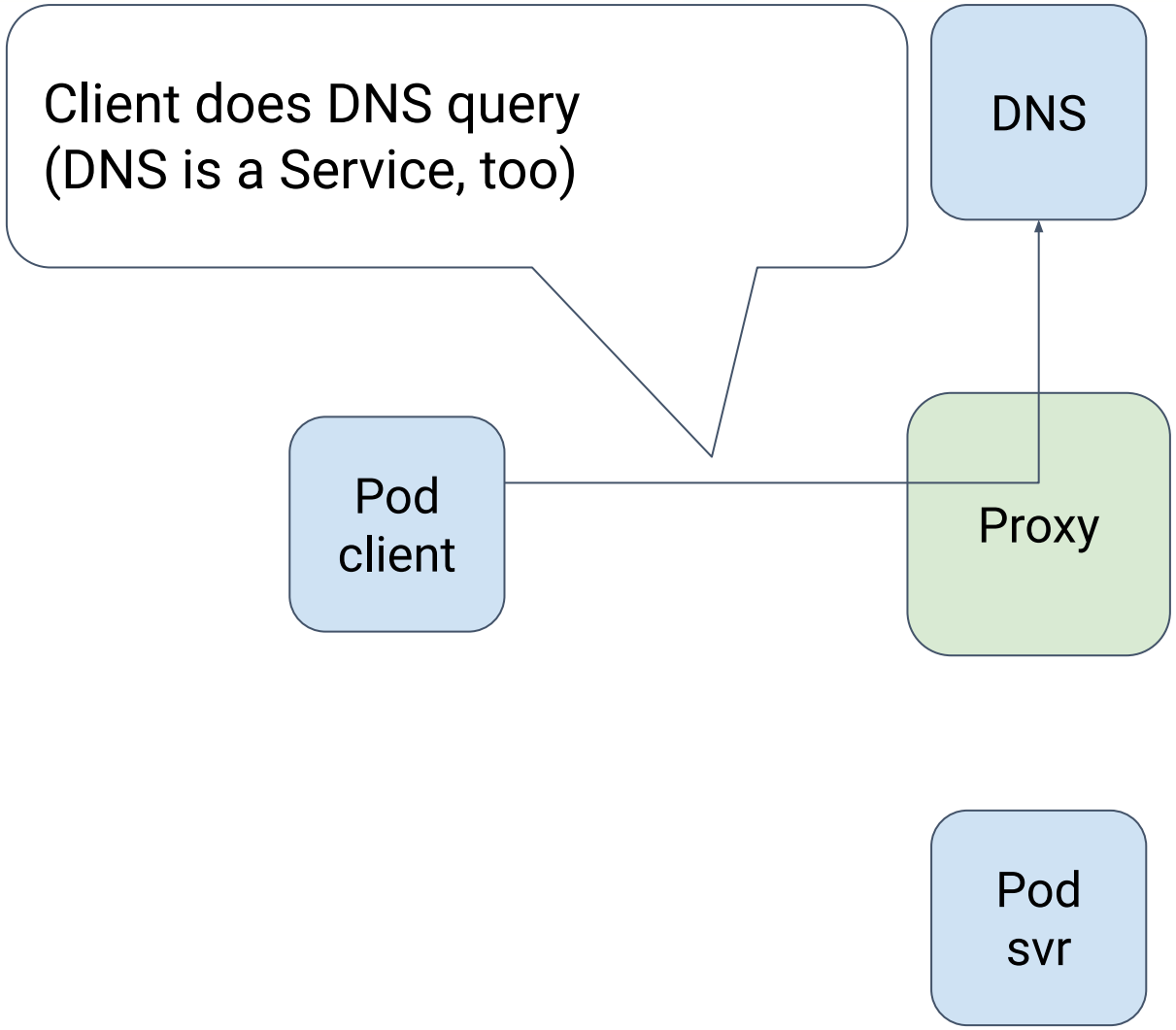
Services “expose” a group of pods

- Durable VIP (or not, if you choose)
- Port and protocol
- Used to build service discovery
- Can include load balancing (but doesn’t have to)

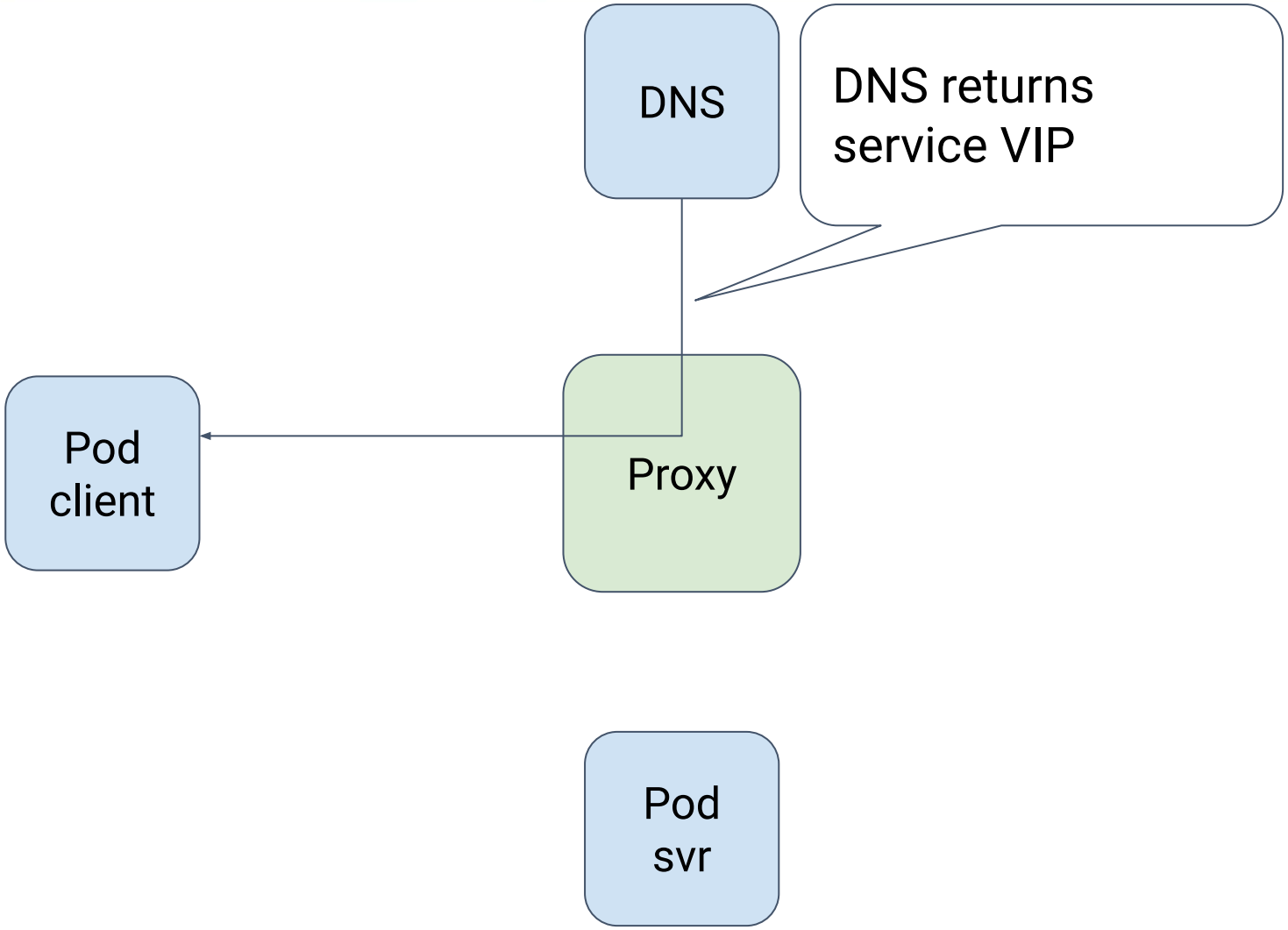
Services: what really happens?



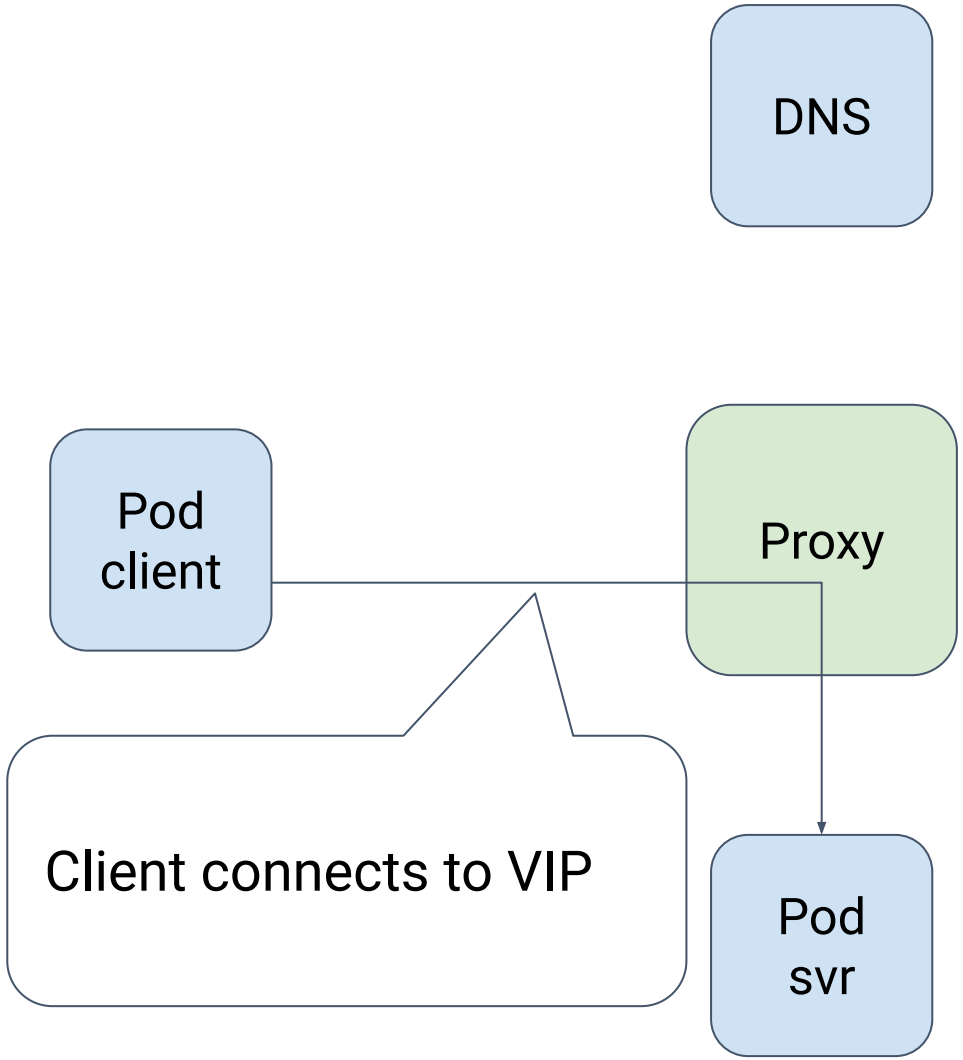
Services: what really happens?



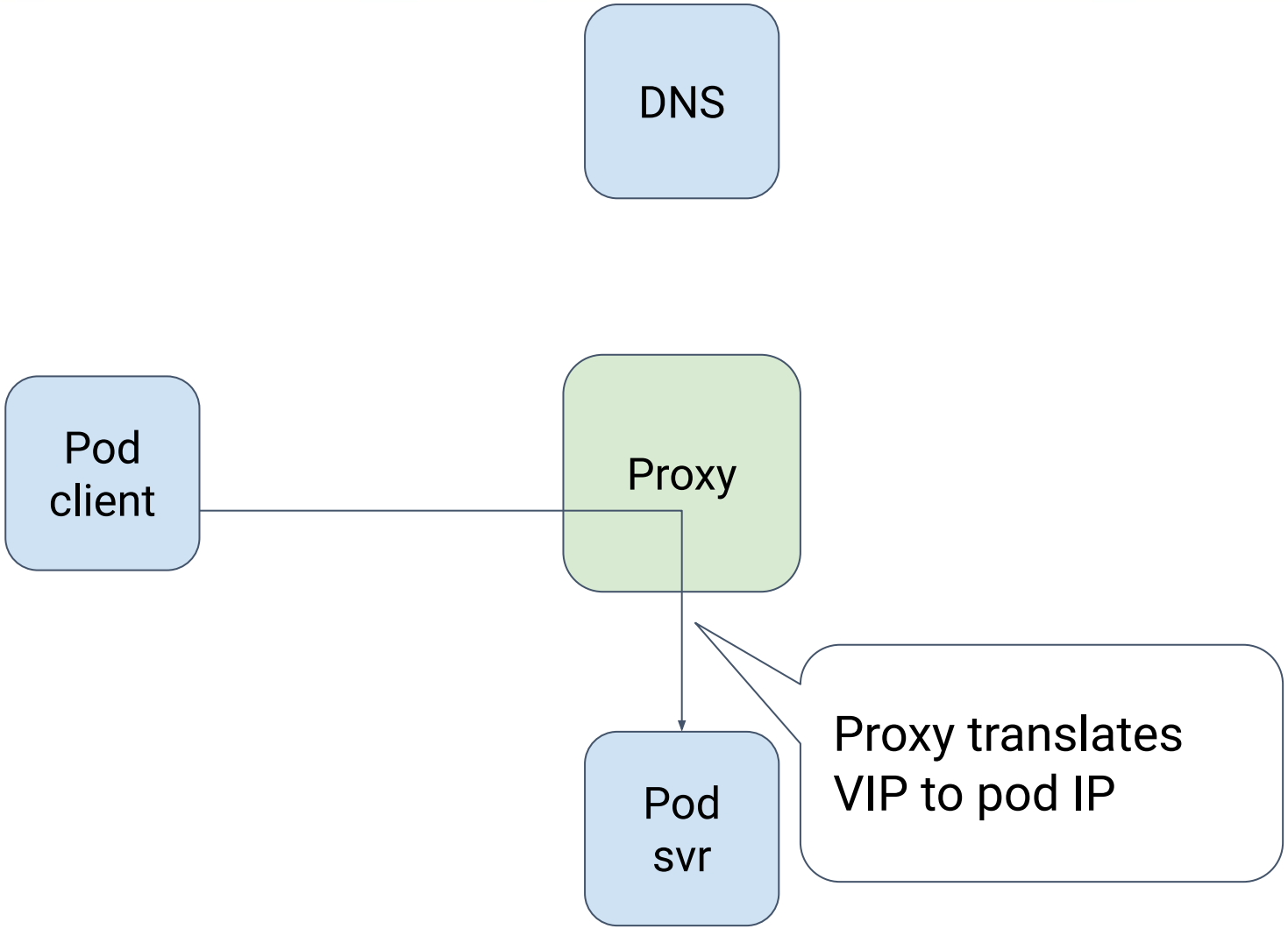
Services: what really happens?



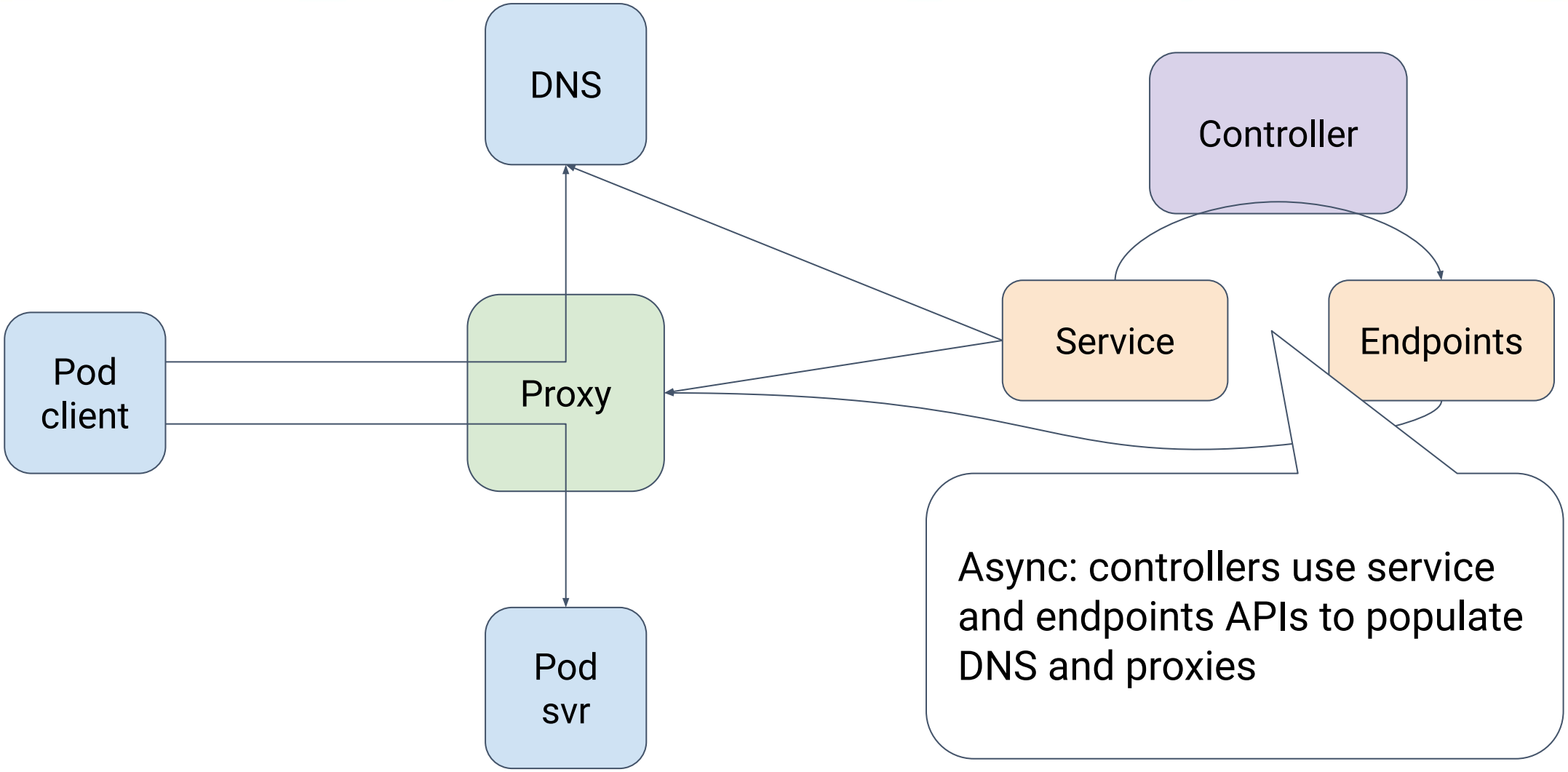
Services: what really happens?



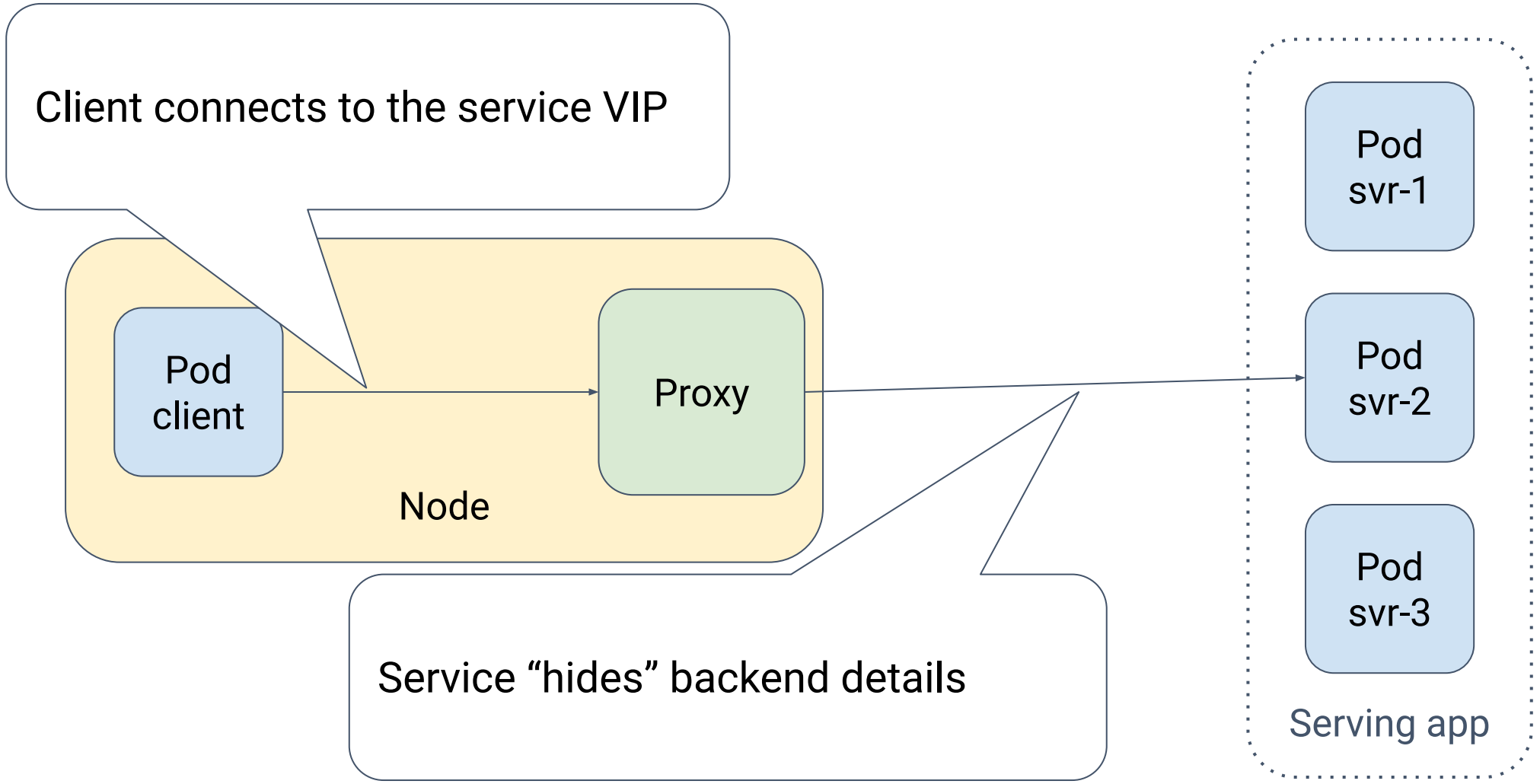
Services: what really happens?



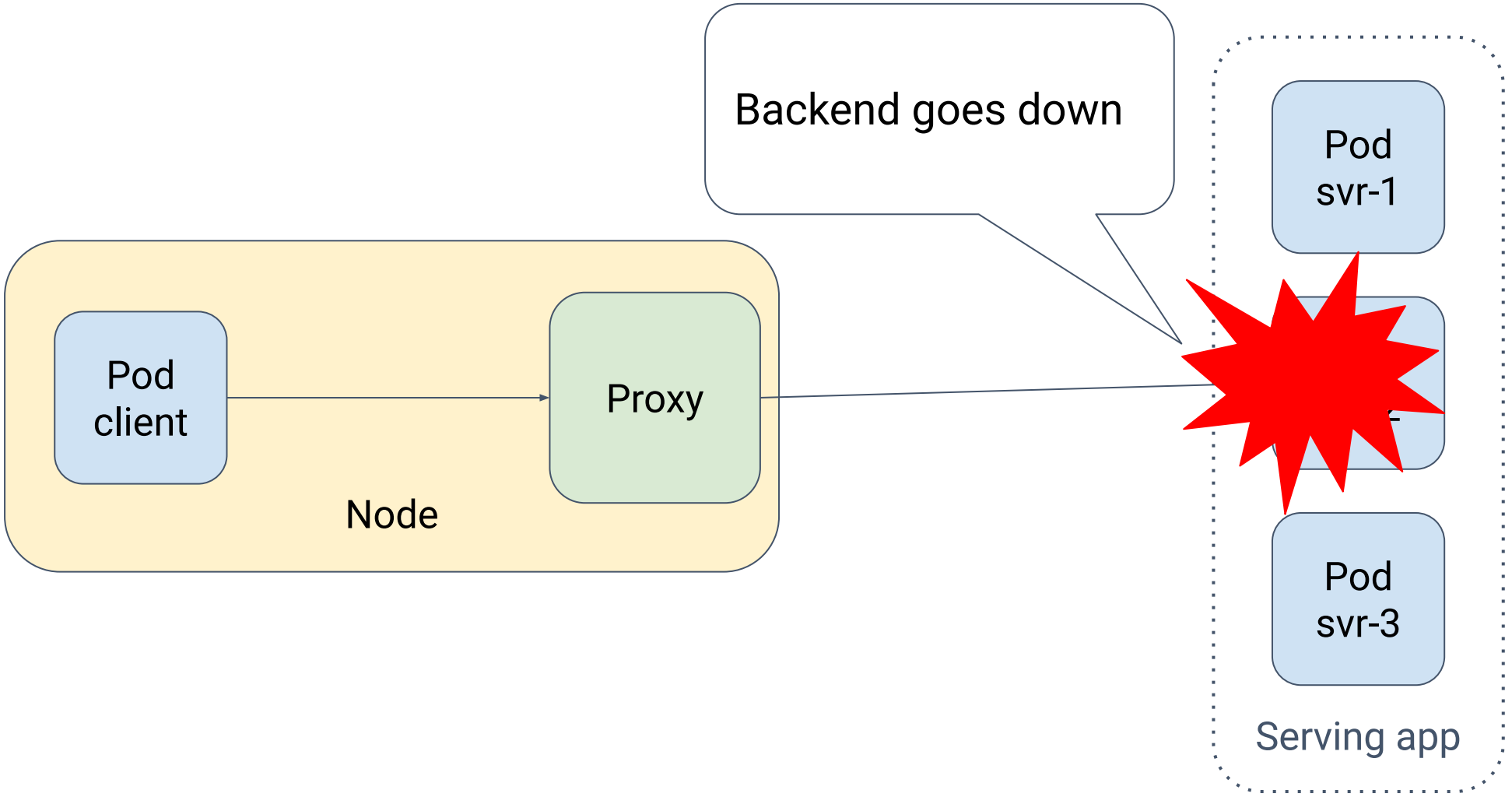
Services: what really happens?



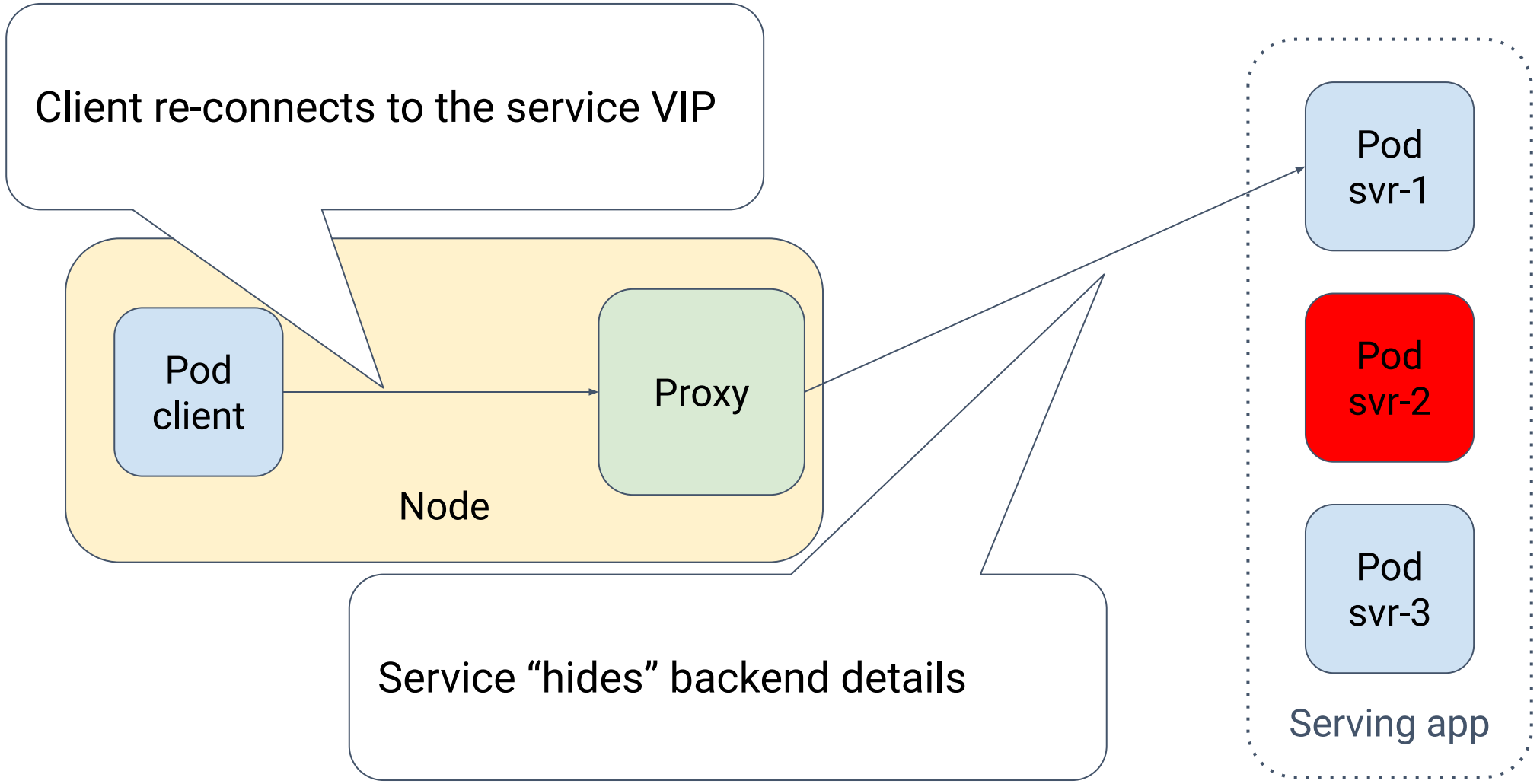
Services: what really happens?



Services: what really happens?



Services: what really happens?



Services: what you specify


```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: default
Spec:
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 9376
```

Used for discovery (e.g. DNS)

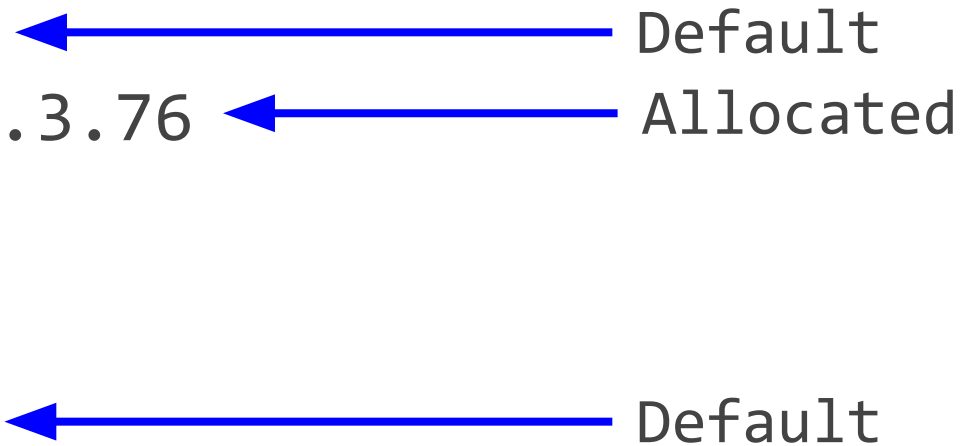
Which pods to use

Logical port (for clients)

Port on the backend pods

Services: what you get


```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: default
Spec:
  type: ClusterIP
  clusterIP: 10.9.3.76
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```



Endpoints

Represents the list of IPs “behind” a Service

- Usually Pods, but not always

Recall that Service had `port` and `targetPort` fields

- Can “remap” ports

Generally managed by the system

- But can be manually managed in some cases

Endpoints controller(s)


```
Service {
  name: foo
  selector:
    app: foo
  ports:
  - port: 80
    targetPort: 9376
}
```

Endpoints controller(s)


```
Pod {
  labels:
    app: foo
  ip: 10.1.0.1
}
```

```
Pod {
  labels:
    app: bar
  ip: 10.1.0.2
}
```

```
Pod {
  labels:
    app: foo
  ip: 10.1.9.3
}
```

```
Pod {
  labels:
    app: qux
  ip: 10.1.1.8
}
```

```
Pod {
  labels:
    app: foo
  ip: 10.1.7.6
}
```

```
Service {
  name: foo
  selector:
    app: foo
  ports:
  - port: 80
    targetPort: 9376
}
```

Endpoints controller(s)


```
Pod {
  labels:
    app: foo
  ip: 10.1.0.1
}
```

```
Pod {
  labels:
    app: bar
  ip: 10.1.0.2
}
```

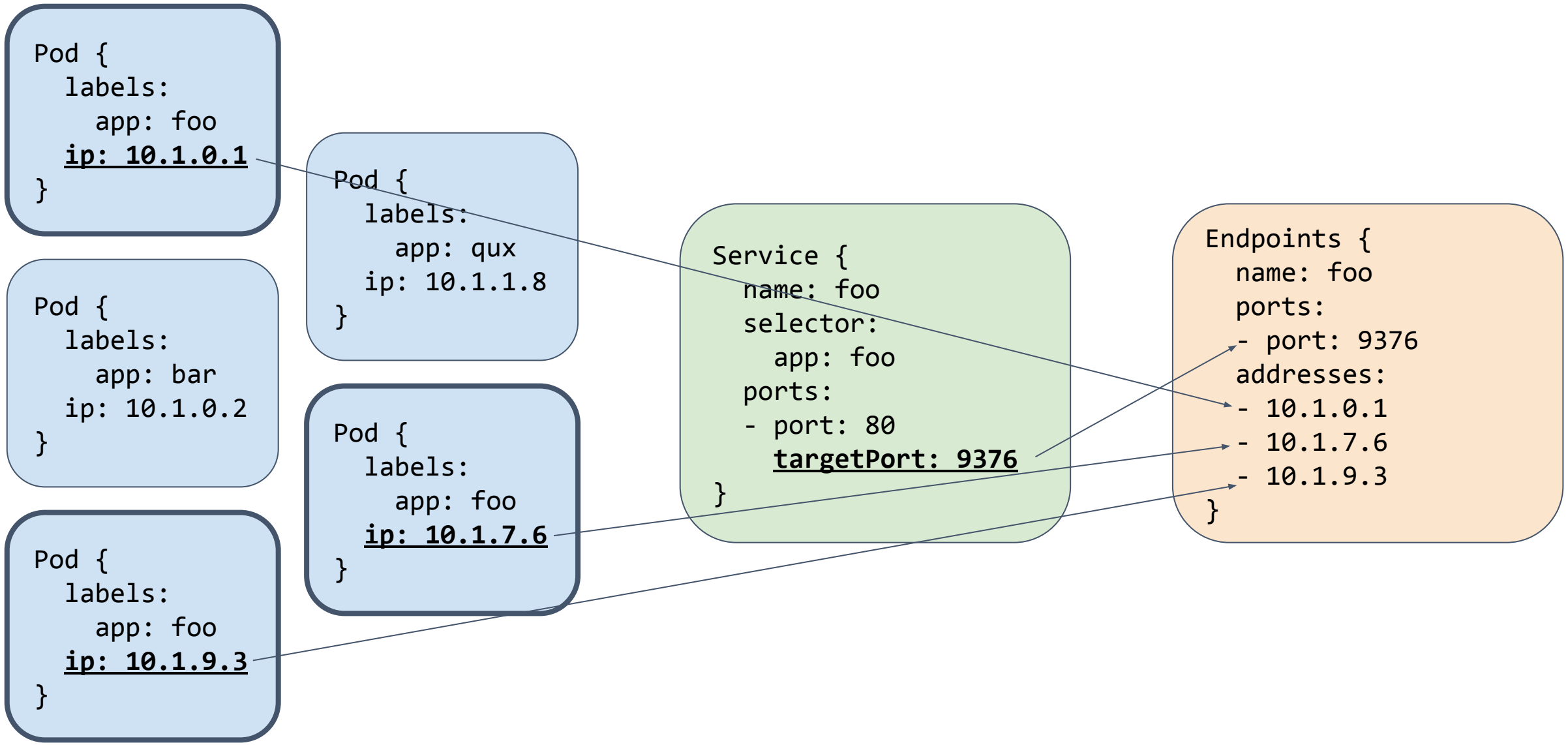
```
Pod {
  labels:
    app: foo
  ip: 10.1.9.3
}
```

```
Pod {
  labels:
    app: qux
  ip: 10.1.1.8
}
```

```
Pod {
  labels:
    app: foo
  ip: 10.1.7.6
}
```

```
Service {
  name: foo
  selector:
    app: foo
  ports:
  - port: 80
    targetPort: 9376
}
```

Endpoints controller(s)



DNS

Starts with a specification

- A, AAAA, SRV, PTR record formats

Generally runs as pods in the cluster

- But doesn’t have to

Generally exposed by a Service VIP

- But doesn’t have to be

Containers are configured by kubelet to use kube-dns

- Search paths make using it even easier

Default implementation is CoreDNS

Services: DNS

The name of your service

The cluster's DNS zone

`my-service.default.svc.cluster.local`

The namespace your service lives in

Indicates a service name

kube-proxy

Default implementation of Services

- But can be replaced!

Runs on every Node in the cluster

Uses the node as a proxy for traffic from pods on that node

- iptables, IPVS, winkernel, or userspace options
- Linux: iptables & IPVS are best choice (in-kernel)

Transparent to consumers

Kube-proxy: control path

Watch Services and Endpoints

Apply some filters

- E.g. ignore “headless” services

Link Endpoints (backends) with Services (frontends)

Accumulate changes to both

Update node rules

Kube-proxy: data path

Recognize service traffic

- E.g. Destination VIP and port

Choose a backend

- Consider client affinity if requested

Rewrite packets to new destination (DNAT)

Un-DNAT on response

Kube-proxy: FAQ

Q: Why not just use DNS-RR?

A: DNS clients are generally “broken” and don’t handle changes to DNS records well. This provides a stable IP while backends change

Q: My clients are enlightened, can I opt-out?

A: Yes! Headless Services get a DNS name but no VIP.

Service LoadBalancers

Services are also how you configure L4 load-balancers

Different LBs work in different ways, too broad for this talk

Integrations with most cloud providers

Ingress

Describes an HTTP proxy and routing rules

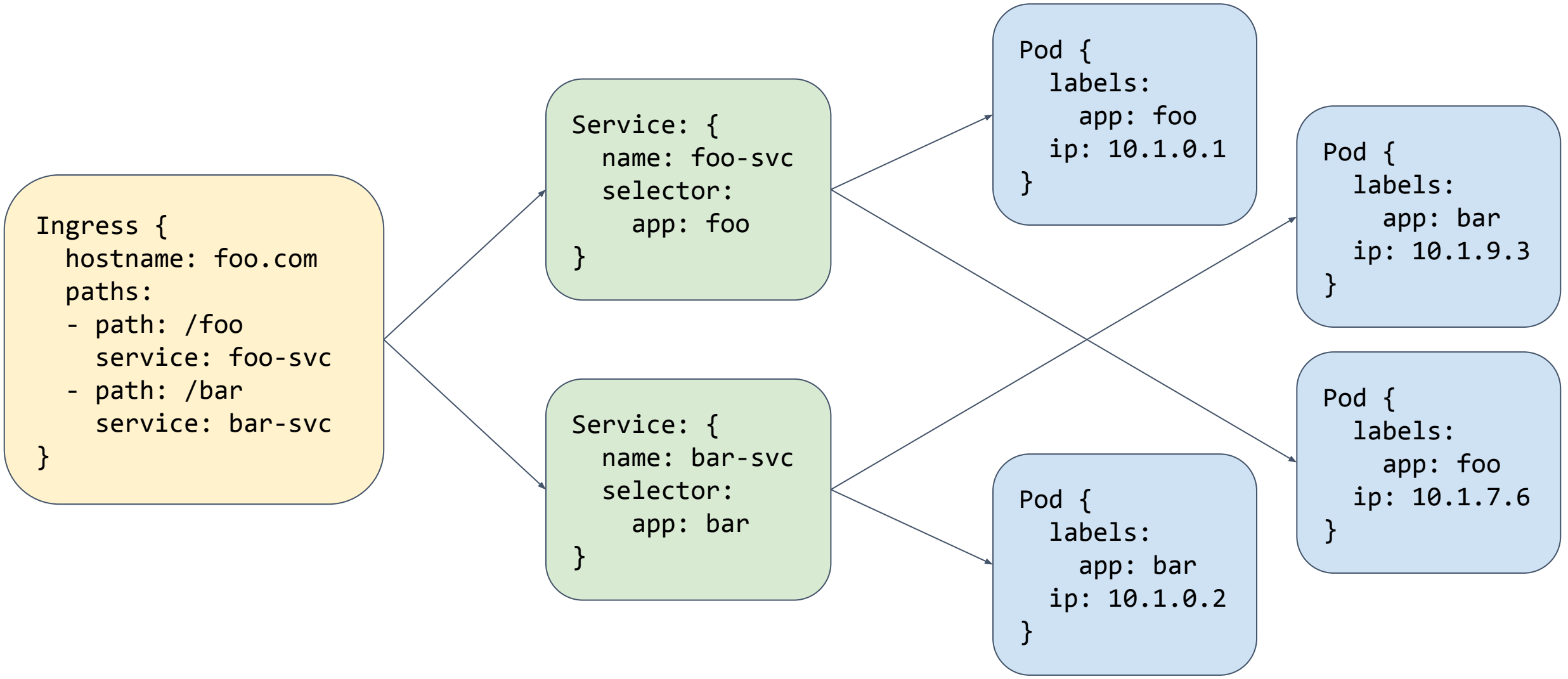
- Simple API - match hostnames and URL paths
- Too simple, more on this later

Targets a Service for each rule

Kubernetes defines the API, but implementations are 3rd party

Integrations with most clouds and popular software LBs

Ingress


Ingress {
hostname: foo.com
paths:
- path: /foo
service: foo-svc
- path: /bar
service: bar-svc
}
Service: {
name: foo-svc
selector:
app: foo
}
Service: {
name: bar-svc
selector:
app: bar
}
Pod {
labels:
app: foo
ip: 10.1.0.1
}
Pod {
labels:
app: bar
ip: 10.1.0.2
}
Pod {
labels:
app: bar
ip: 10.1.9.3
}
Pod {
labels:
app: foo
ip: 10.1.7.6
}

Ingress FAQ

Q: How is this different from Service LoadBalancer?

A: Service LB API does not provide for HTTP - no hostnames, no paths, no TLS, etc.

Q: Why isn't there a controller "in the box"?

A: We didn't want to be "picking winners" among the software LBs. That may have been a mistake, honestly.

NetworkPolicy

Describes the allowed call-graph for communications

- E.g. frontends can talk to backends, backends to DB, but never frontends to DB

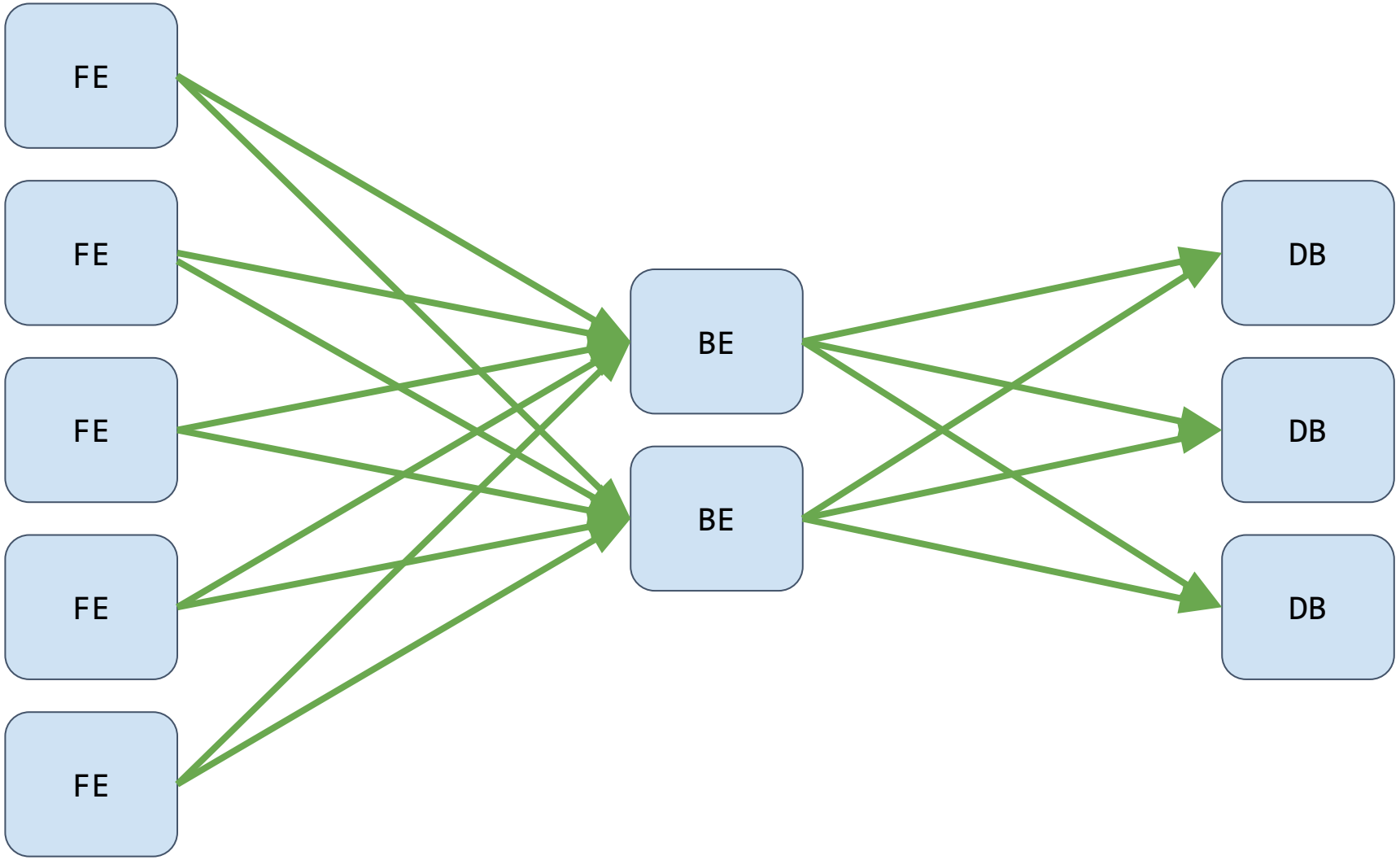
Like Ingress, implementations are 3rd-party

- Often highly coupled to low-level network drivers

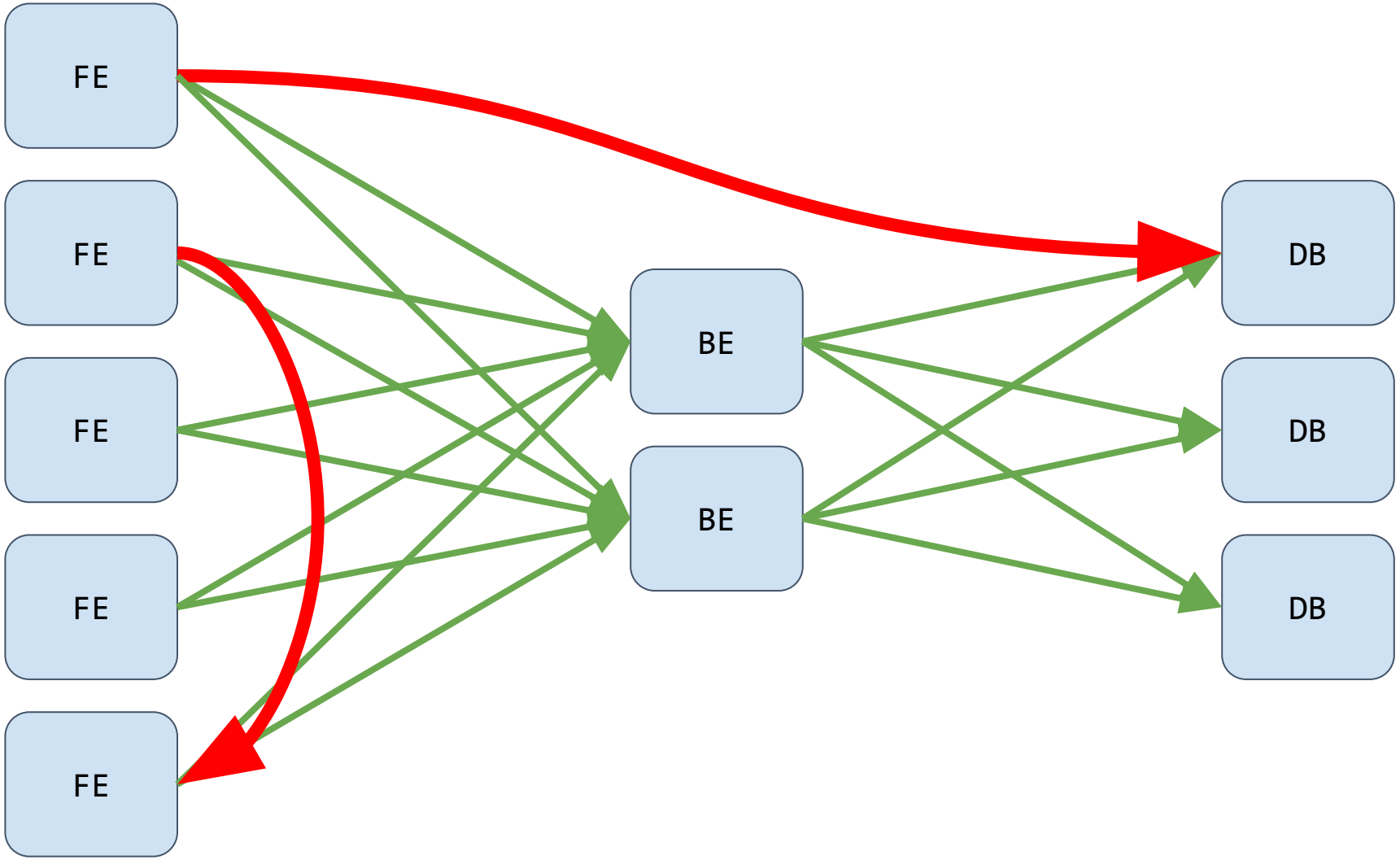
Very simple rules - focused on app-owners rather than cluster or network admins

- We may need a related-but-different API for the cluster operators

NetworkPolicy



NetworkPolicy



Part 2: Deep-Dive

Deep-dive

On-going work in the SIG:

- NodeLocal DNS
- EndpointSlice
- Services (Gateway API, MultiClusterService)
- IPv{4,6} Dual stack

NodeLocal DNS

Kubernetes DNS resource cost is high:

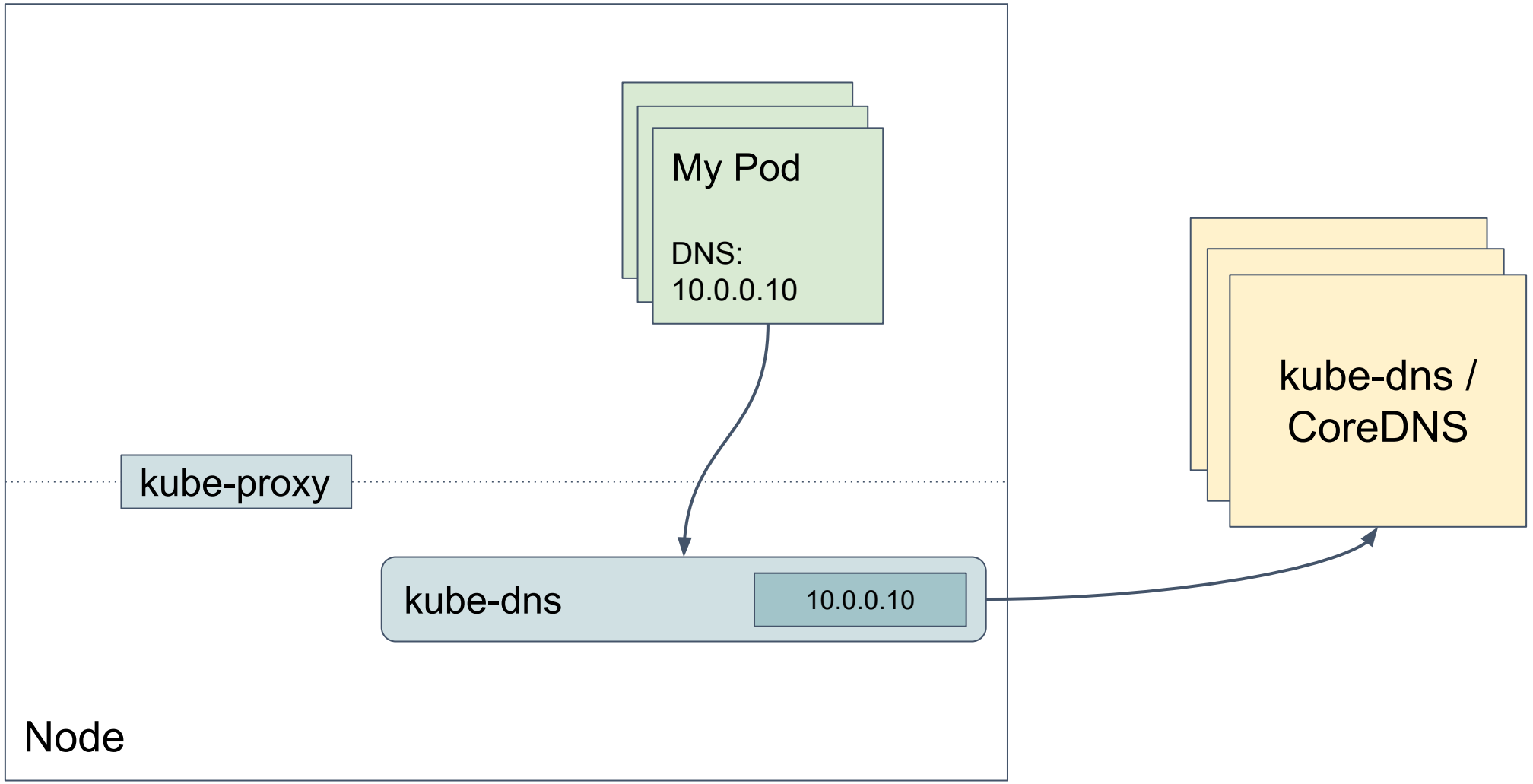
- Expansion due to alias names (“my-service”, “my-service.ns”, ...)
- Application density (e.g. microservices)
- DNS-heavy application libraries (e.g. Node.JS)
- CONNTRACK entries due to UDP

Solution? NodeLocal DNS (GA v1.18)

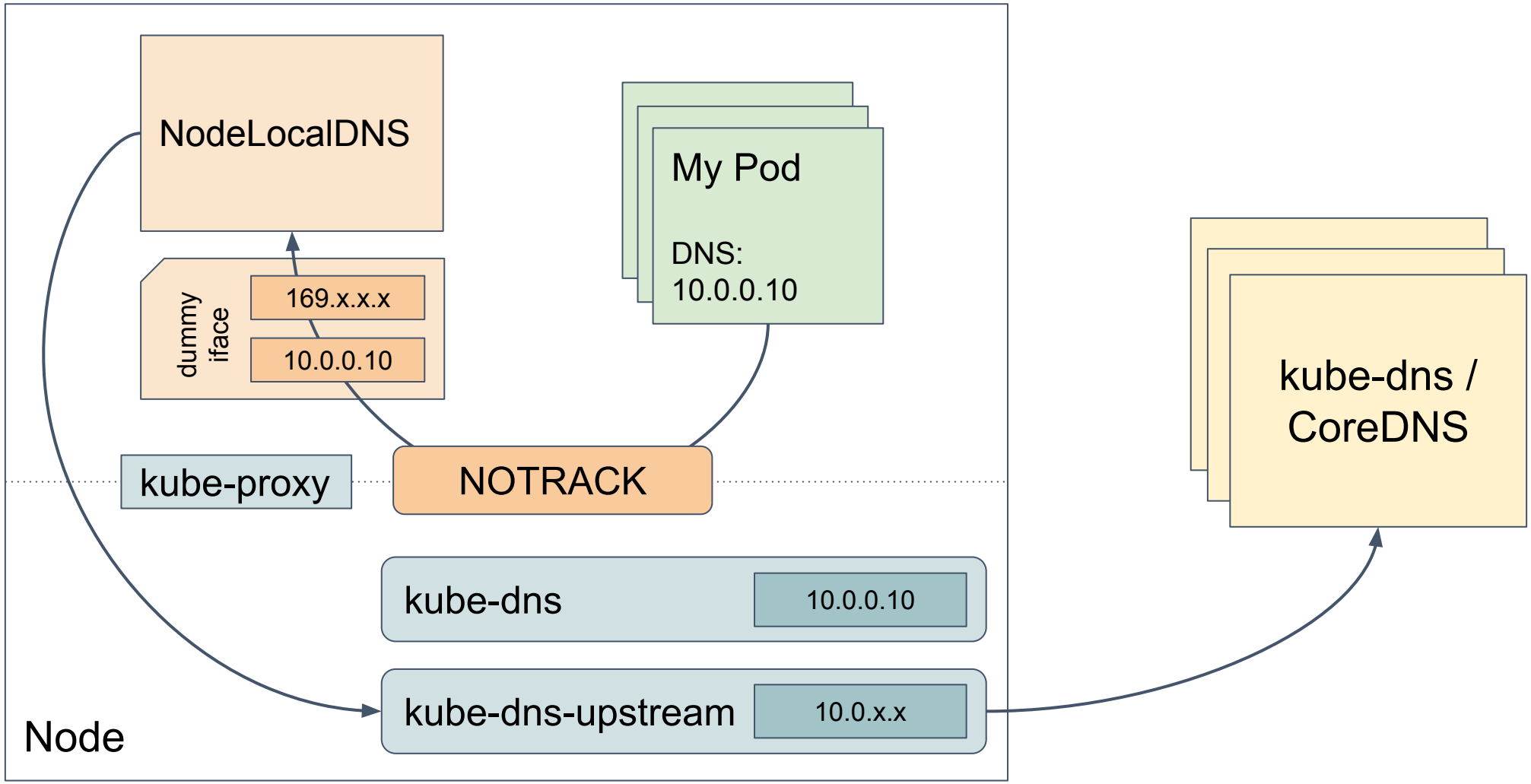
- Run a cache on every node
- Careful: per-node overhead can easily dominate in large clusters

As a system-critical service in a Daemonset, we need to be careful about high-availability during upgrades, failures.

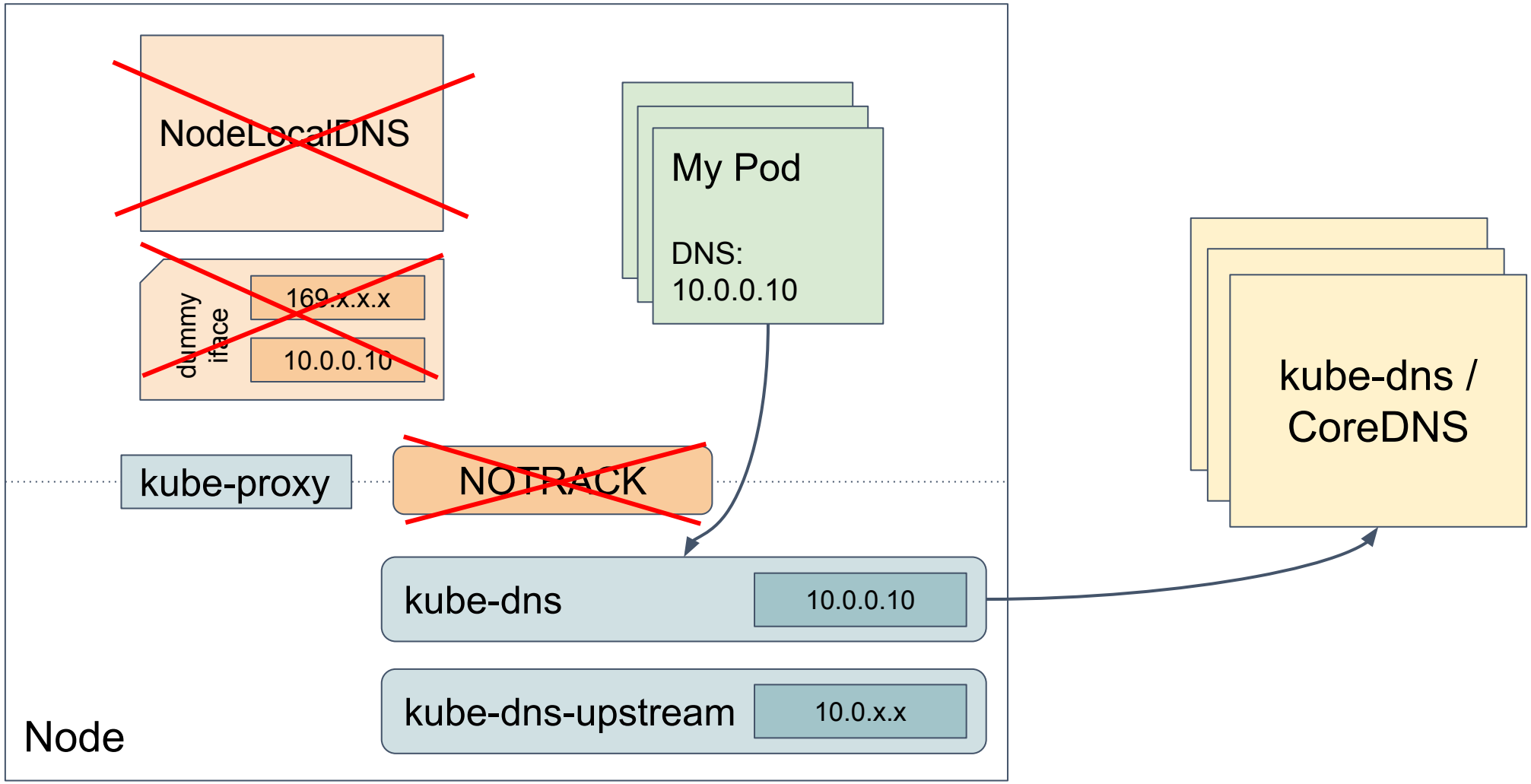
NodeLocal DNS



NodeLocal DNS



NodeLocal DNS



DNS

We can do better though:

- Proposal: push alias expansion into the server as an API (enhancements/pull/967)
- Refactor the DNS naming scheme altogether?

EndpointSlice

Larger clusters (think 15k nodes) and very large Services lead to API scalability issues:

- Size of a single object in etcd
- Amount of data sent to watchers
- etcd DB activity

# of nodes: 5000

Size of Endpoints object: 1 MB

total bytes transmitted per update:

5000 X 1 MB = 5GB DVD?

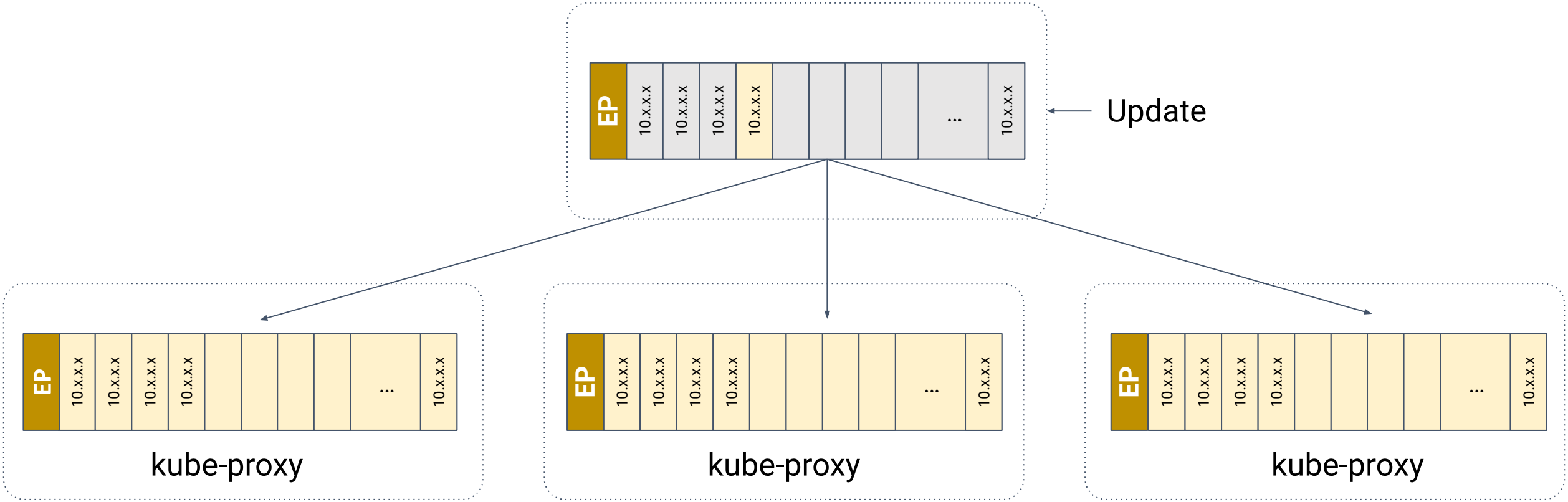
total bytes transmitted per update: 5GB

rolling update?

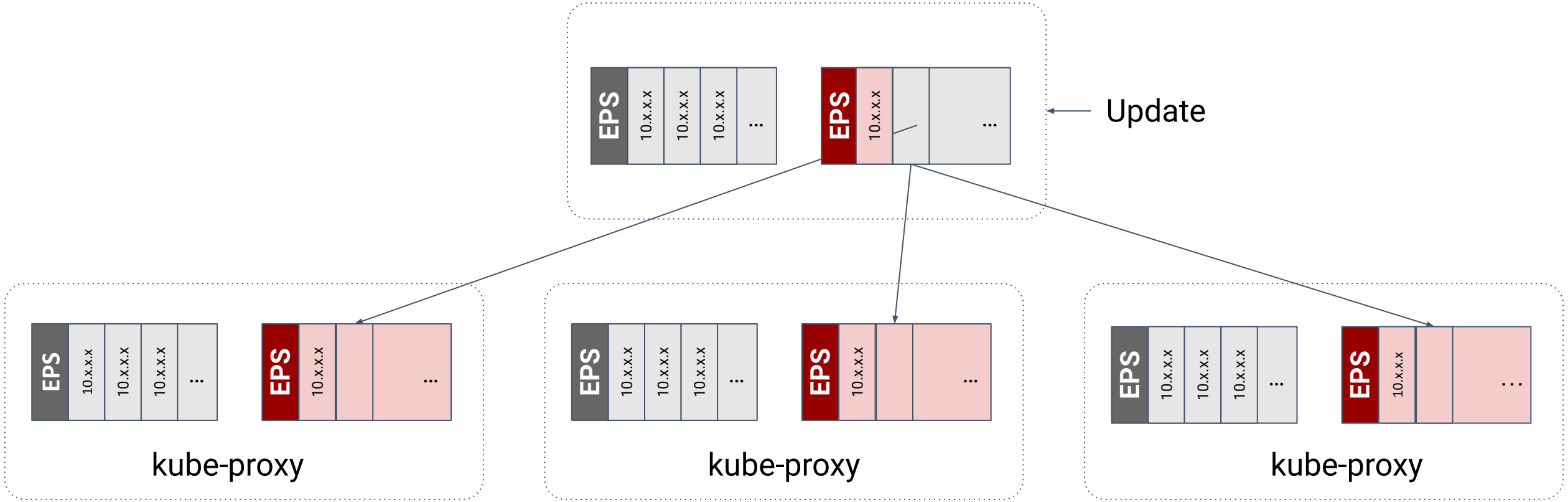
~5000 X 5 GB = 25TB !

Source: Scale Kubernetes Service Endpoints 100X, (Tyczynski, Xia)

EndpointSlice



EndpointSlice



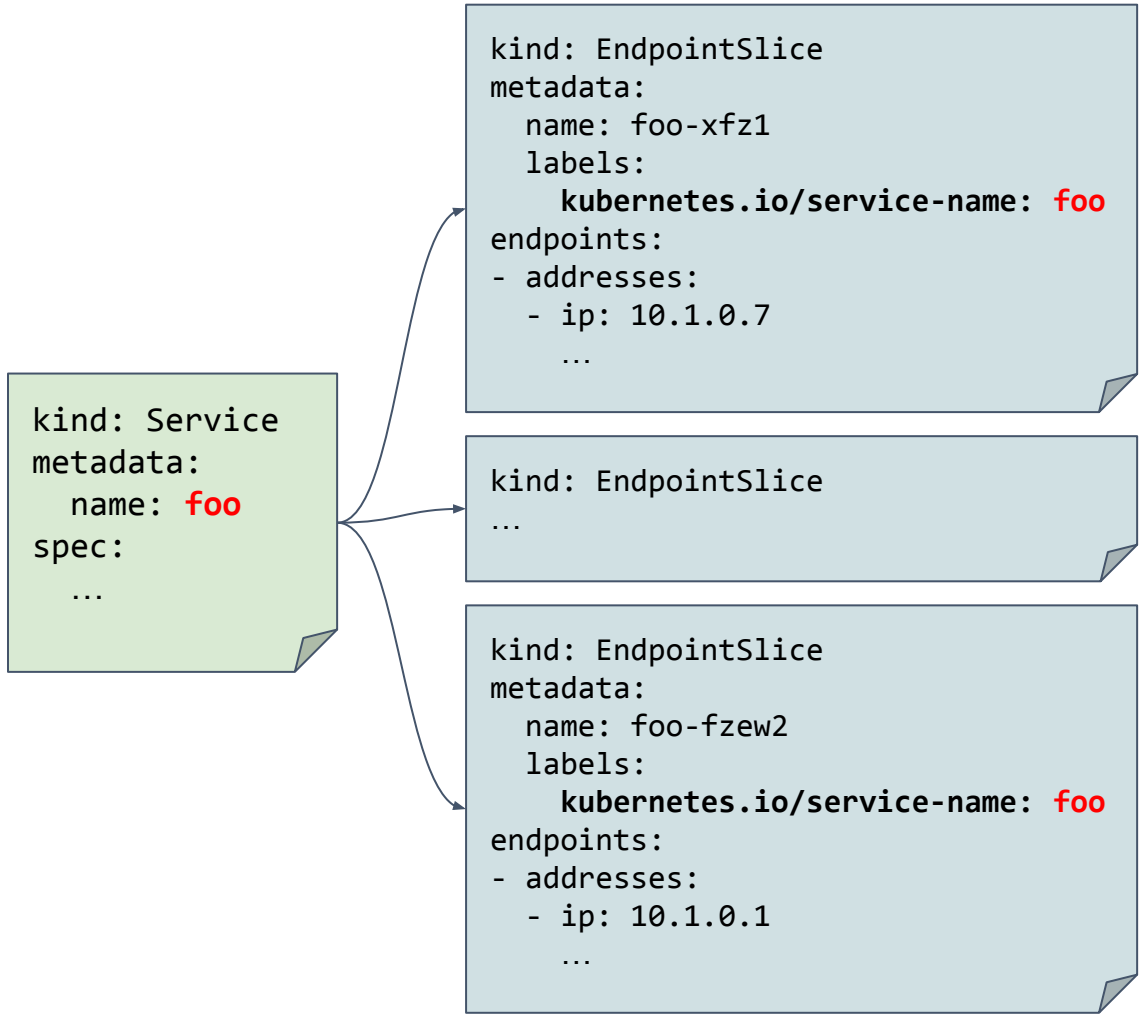
EndpointSlice

Controllers

EndpointSlices controller: slices from Service selector. Linked to the Service via `kubernetes.io/service-name` label

EndpointSliceMirroring controller: slices from selectorless Service's

Other users can set `endpointslice.kubernetes.io/managed-by`



EndpointSlice

Update algorithm is an optimization problem:

- Keep number of slices low
- Minimize changes to slices per update
- Keep amount of data sent low

Current algorithm

1. Remove stale endpoints in existing slices
2. Fill new endpoints in free space
3. Create new slices only if no more room

No active rebalancing -- claim: too much churn, open area

EndpointSlice

v1.17	v1.18	v1.19	v1.20+
Beta	**Beta**	**Beta**	**GA**
EndpointSlice controller available	EndpointSlice controller enabled no kube-proxy	EndpointSlice controller EndpointSliceMirror Windows kube-proxy enabled	

Services across Clusters

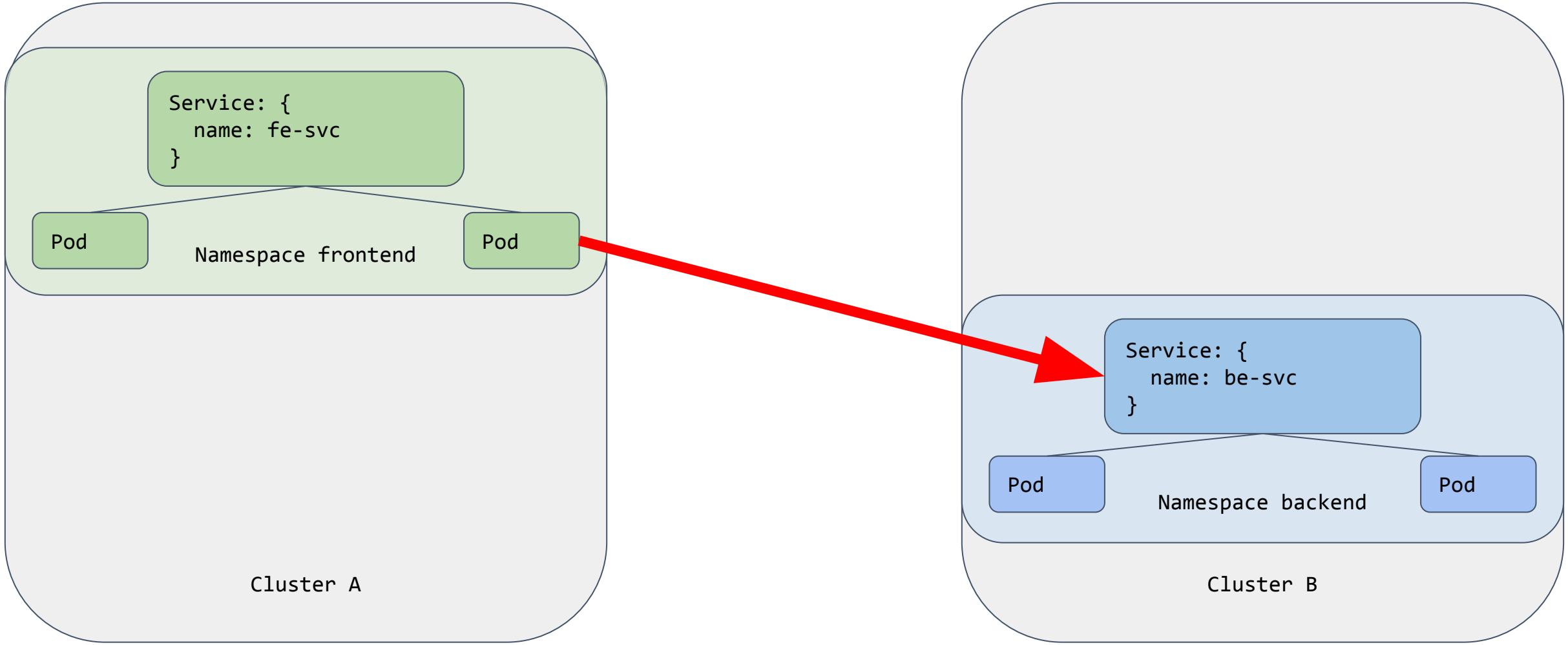
As Kubernetes installations get bigger - multiple clusters is becoming the norm

- LOTS of reasons for this: HA, blast radius, geography, etc.

Services have always been a cluster-centric abstraction

Starting to work through how to export and extend Services across clusters

Services across Clusters

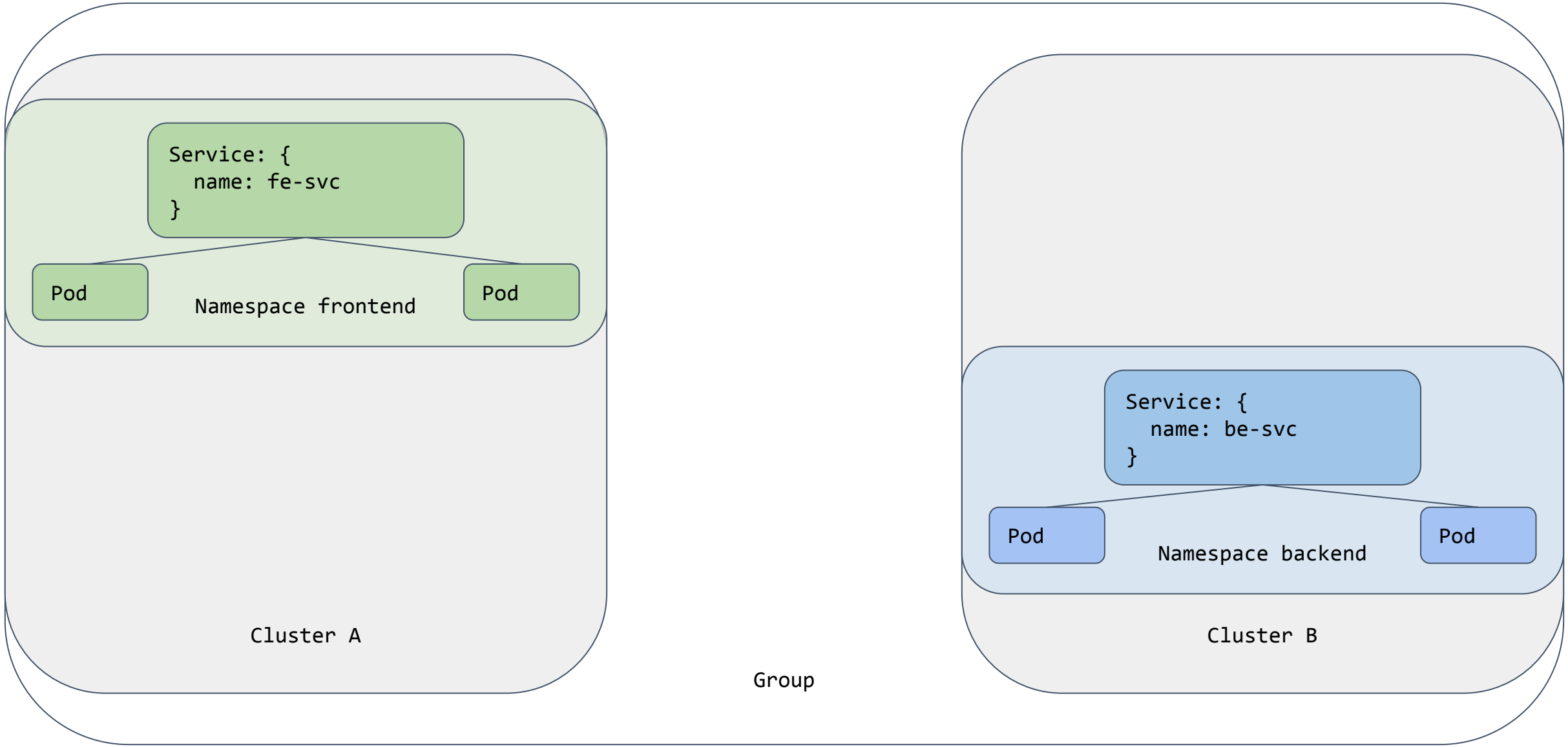


ServiceExport

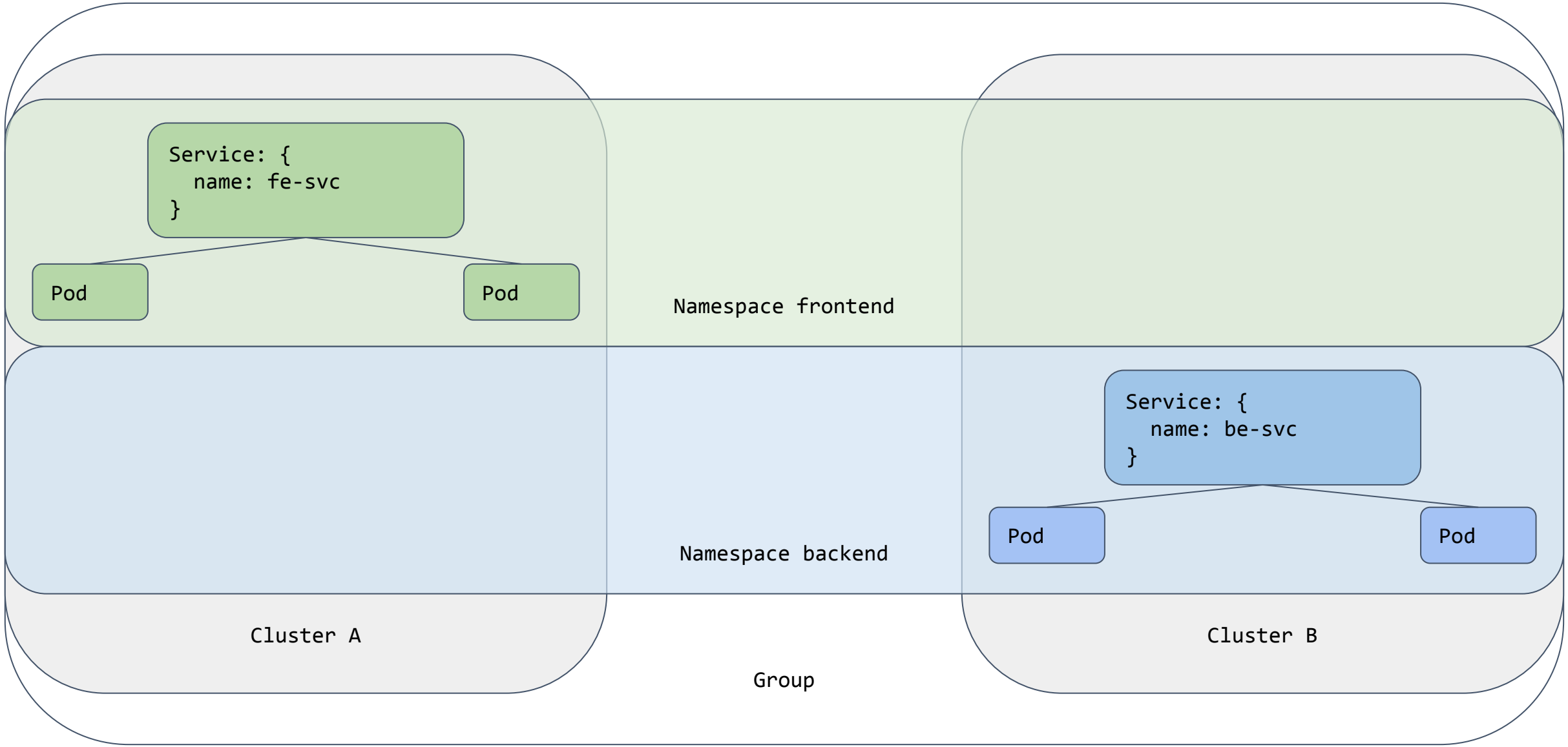

```
Service {
   metadata:
        Name: be-svc
   spec:
        type: ClusterIP
        clusterIP: 1.2.3.4
}
```

```
ServiceExport {
   metadata:
        Name: be-svc
}
```

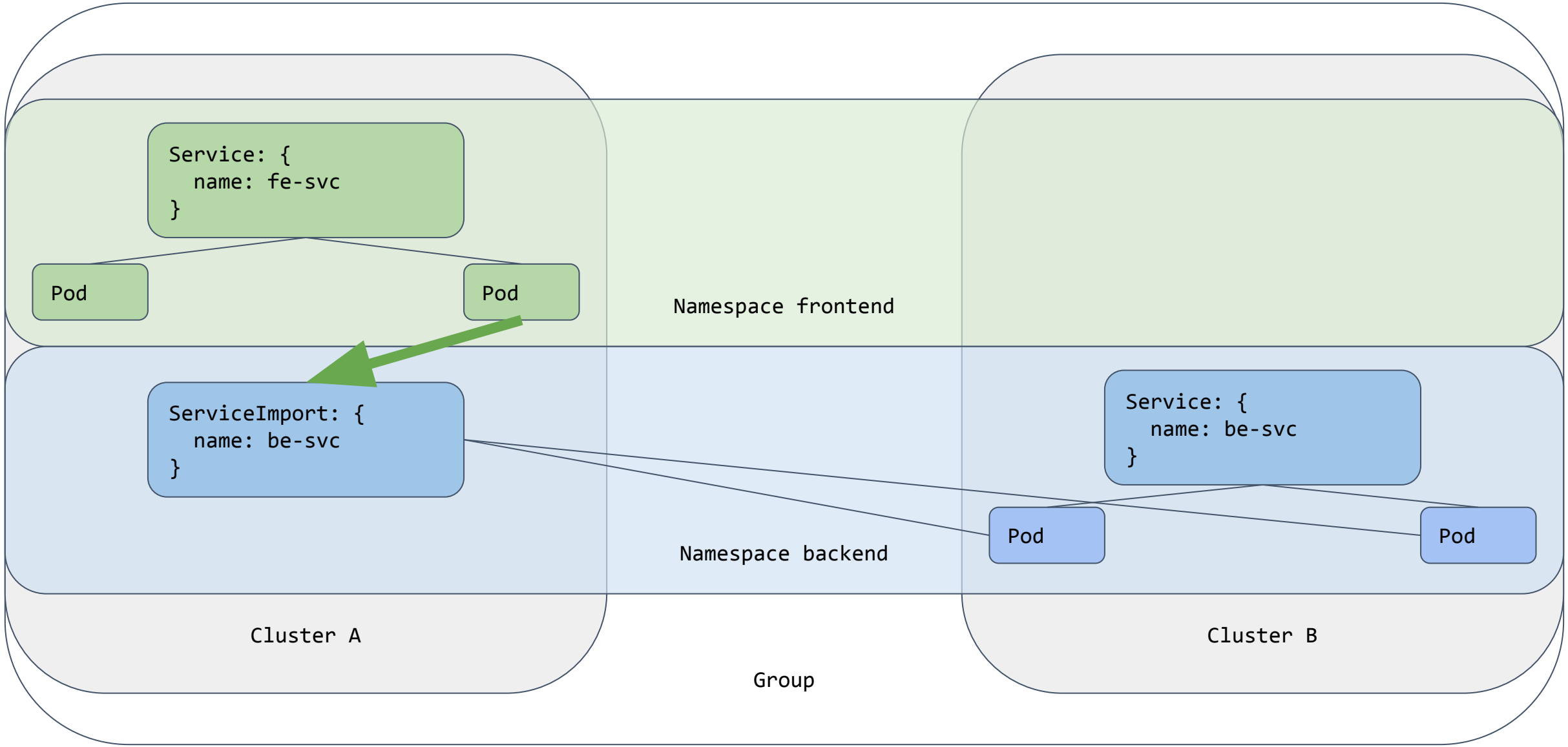
Services across Clusters



Services across Clusters



Services across Clusters



ServiceImports: DNS

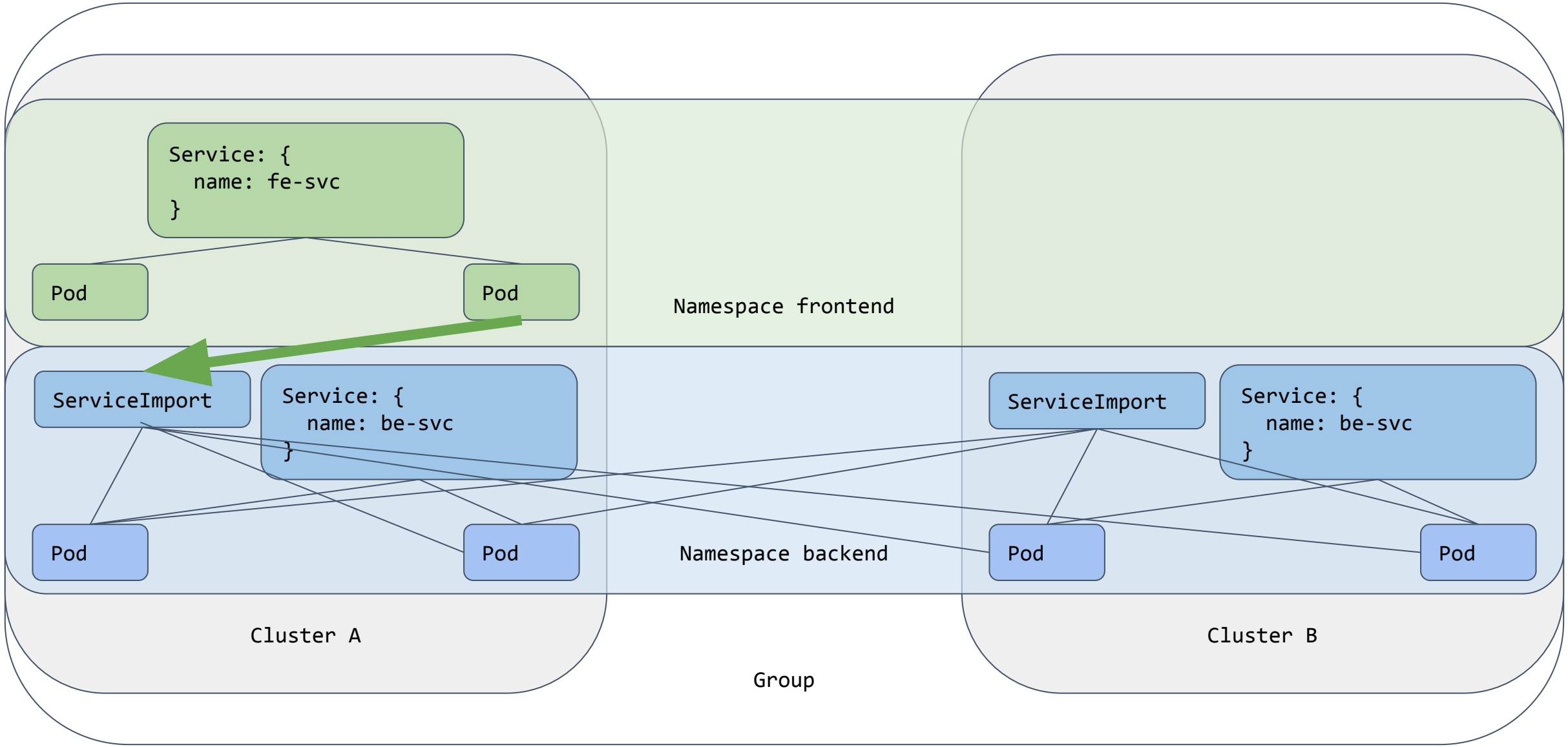
The name of your service

The multi-cluster DNS zone (TBD)

```
be-svc.backend.supercluster.local
```

The namespace your service lives in

Services across Clusters



Services across Clusters

This is mostly KEP-ware right now

Still hammering out API, names, etc

Still working out some semantics (e.g. conflicts)

IPv{4,6} Dual Stack

Some users need IPv4 and IPv6 at the same time

- Kubernetes only supports 1 Pod IP

Some users need Services with both IP families

- Kubernetes only supports 1 Service IP

This is a small, but important change to several APIs

Wasn't this work done already? Yes, but we found some problems, needed a major reboot

IPv{4,6} Dual Stack


```
Pod {
   status:
        podIP: 1.2.3.4
}
```

IPv{4,6} Dual Stack


```
Pod {
   status:
        podIP: 1.2.3.4
        podIPs:
        - 1.2.3.4
        - 1234:5678::0001
}
```

same

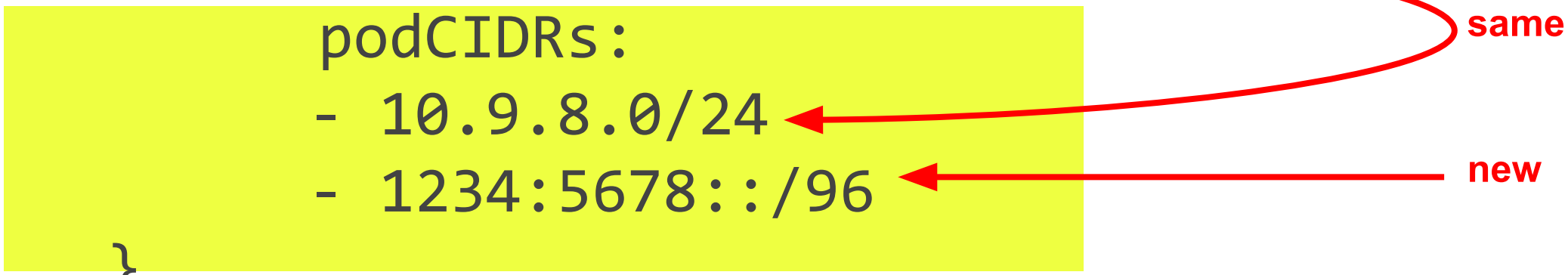
new

IPv{4,6} Dual Stack


```
Node {
   spec:
        podCIDR: 10.9.8.0/24
}
```

IPv{4,6} Dual Stack


```
Node {
   spec:
        podCIDR: 10.9.8.0/24
        podCIDRs:
        - 10.9.8.0/24
        - 1234:5678::/96
}
```



IPv{4,6} Dual Stack


```
Service {
   spec:
        type: ClusterIP
        clusterIP: 1.2.3.4
}
```

IPv{4,6} Dual Stack


```
Service {
  spec:
      type: ClusterIP
      ipFamilyPolicy: PreferDualStack
      ipFamilies: [ IPv4, IPv6 ]
      clusterIP: 1.2.3.4
      clusterIPs:
      - 1.2.3.4
      - 1234::5678::0001
}
```

new

same

new

IPv{4,6} Dual Stack

Can express various requirements:

- “I need single-stack”
- “I’d like dual-stack, if it is available”
- “I need dual-stack”

Defaults to single-stack if users doesn’t express a requirement

Works for headless Services, NodePorts, and LBs (if cloud-provider supports it)

Shooting for second alpha in 1.20

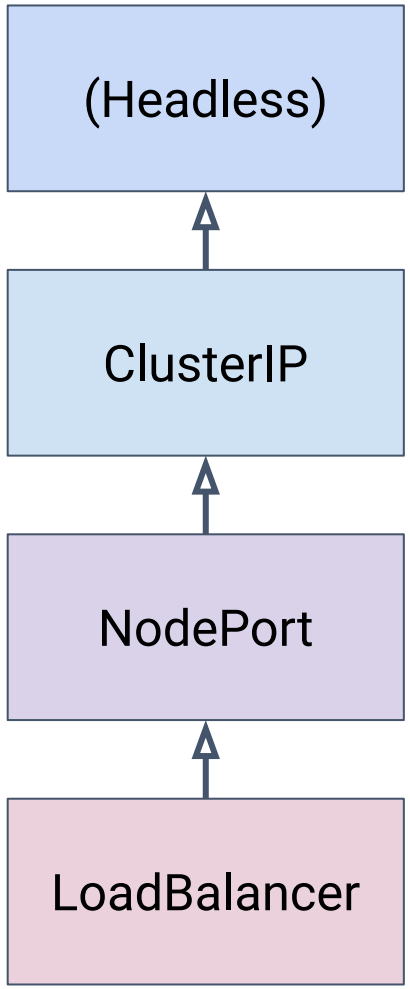
Services V+1

Service resource describes many things:

- Method of exposure (ClusterIP, NodePort, LoadBalancer)
- Grouping of Pods (e.g. selector)
- Attributes (ExternalTrafficPolicy, SessionAffinity, …)

Evolving and extending the resource becomes harder and harder due to interactions between fields…

Evolution of L7 Ingress API: role-based resource modeling, extensibility



Service hierarchy

Services V+1

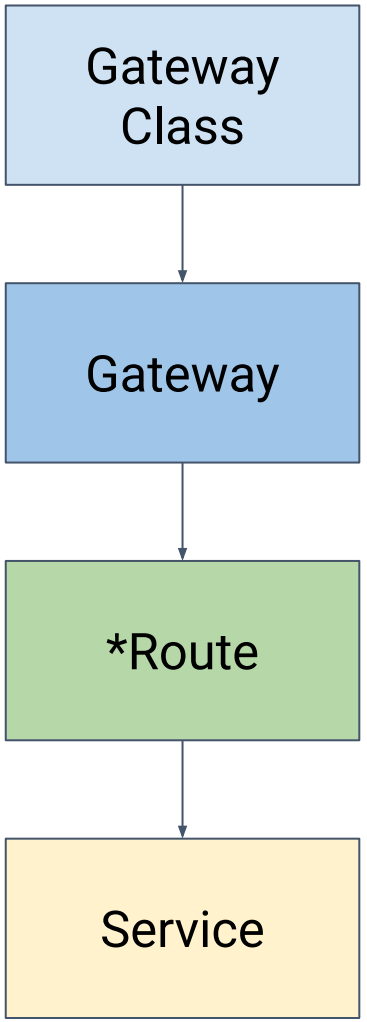
Idea: Decouple along role, concept axes:

Roles:

- 👷 Infrastructure Provider
- 👨‍🔧 Cluster Operator / NetOps
- 👩‍💻 Application Developer

Concepts:

- Grouping, selection
- Routing, protocol specific attributes
- Exposure and access

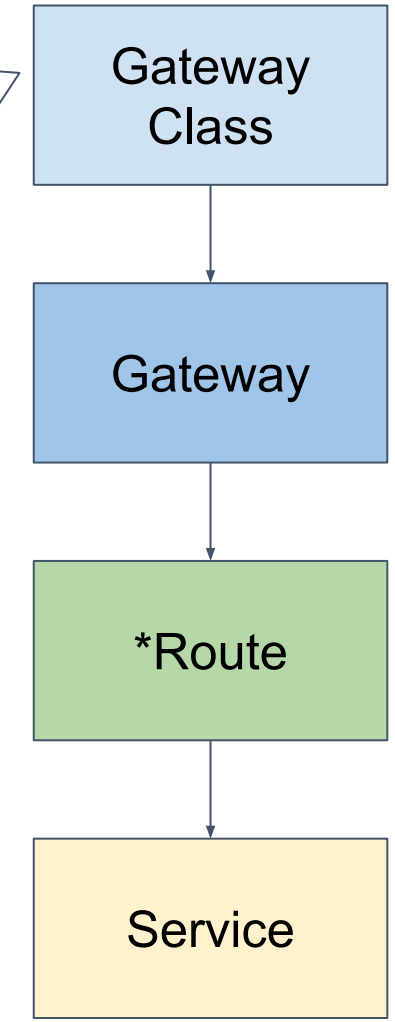


Services V+1

👷 Infrastructure Provider

Defines a kind of Service access for the cluster (e.g. “internal-proxy”, “internet-lb”, …)

Similar to StorageClass, abstracts implementation of mechanism from the consumer.




```
kind: GatewayClass
metadata:
  name: cluster-gateway
spec:
  controller: "acme.io/gateway-controller"
  parametersRef:
    name: internet-gateway
```

Services V+1

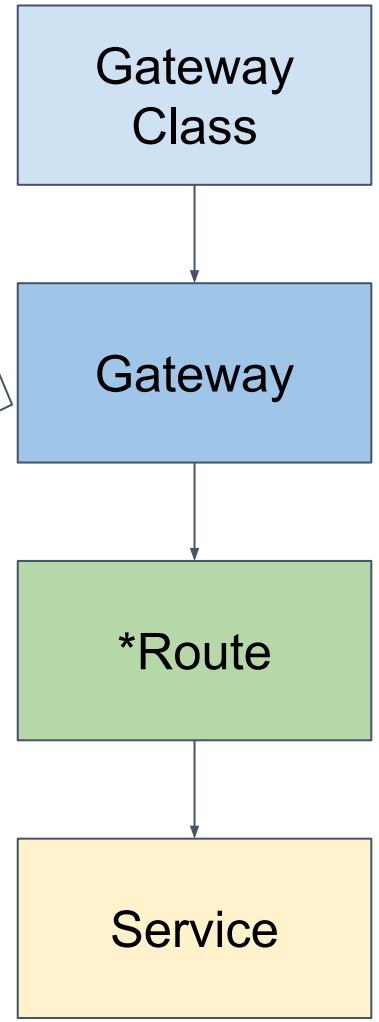
👨‍🔧 Cluster Operator / NetOps

How the Service(s) are access by the user (e.g. port, protocol, addresses)

Keystone resource: 1-1 with configuration of the infrastructure:

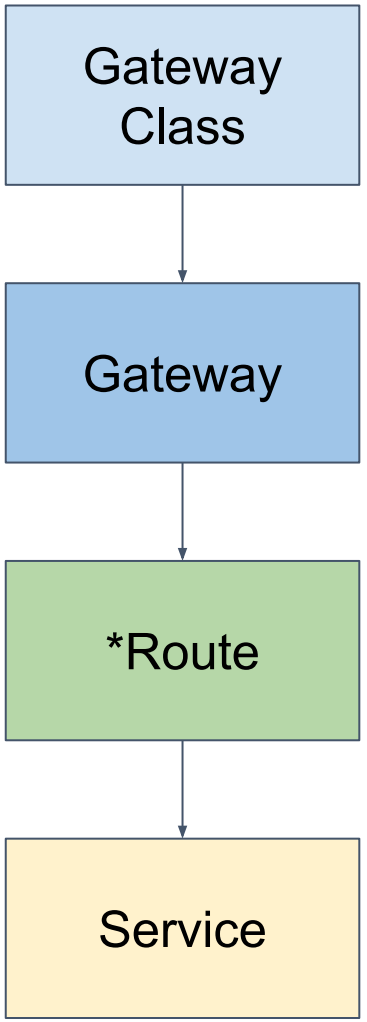
- Spawn a software LB
- Add a configuration stanza to LB.
- Program the SDN

May be “underspecified”: defaults based on GatewayClass.



Services V+1


```
kind: Gateway
metadata:
  name: my-gateway
spec:
  class: cluster-gateway
  # How Gateway is to be accessed (e.g. via Port 80)
  listeners:
  - port: 80
    routes:
    - routeSelector: # Which Routes are linked to this Gateway
      foo: bar
```

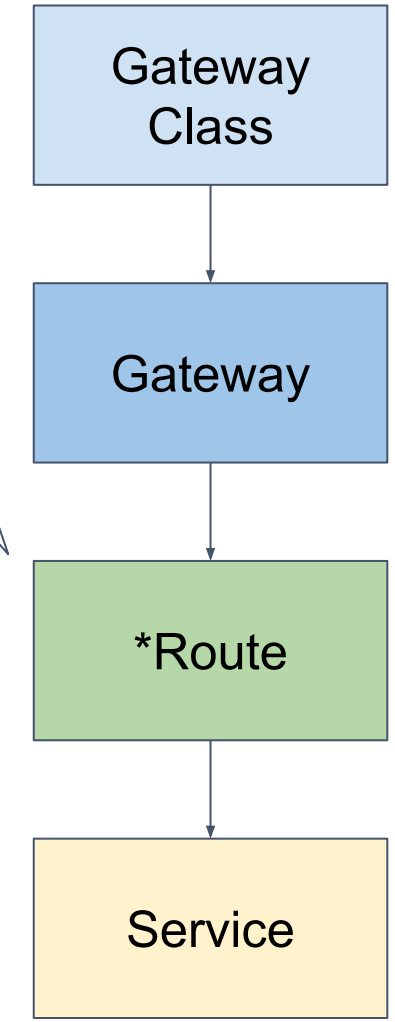


Services V+1

👩‍💻 Application Developer

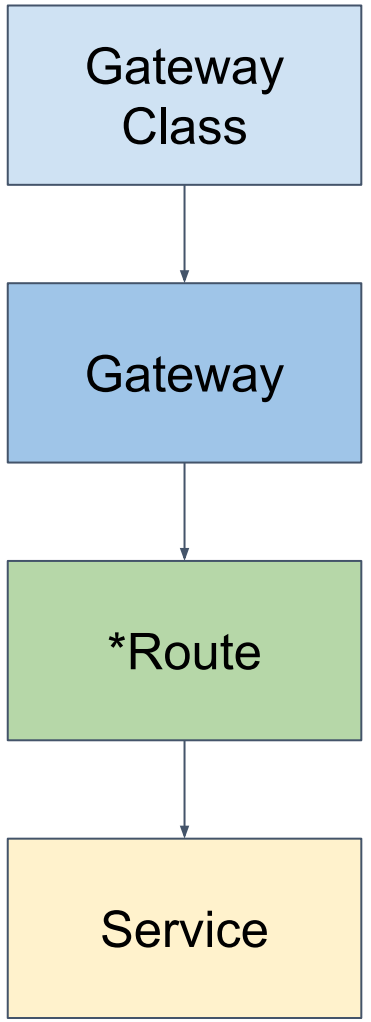
Application routing, composition, e.g. “/search” → service-service, “/store” → store-service.

Family of Resource types by protocol (TCPRoute, HTTPRoute, …) to solve issue of single, closed union type and extensibility.




```
kind: HTTPRoute
metadata:
  name: my-app
spec:
  rules:
  - match: {path: “/store”}
    action: {fowardTo: {targetRef: “store-service”}}
```

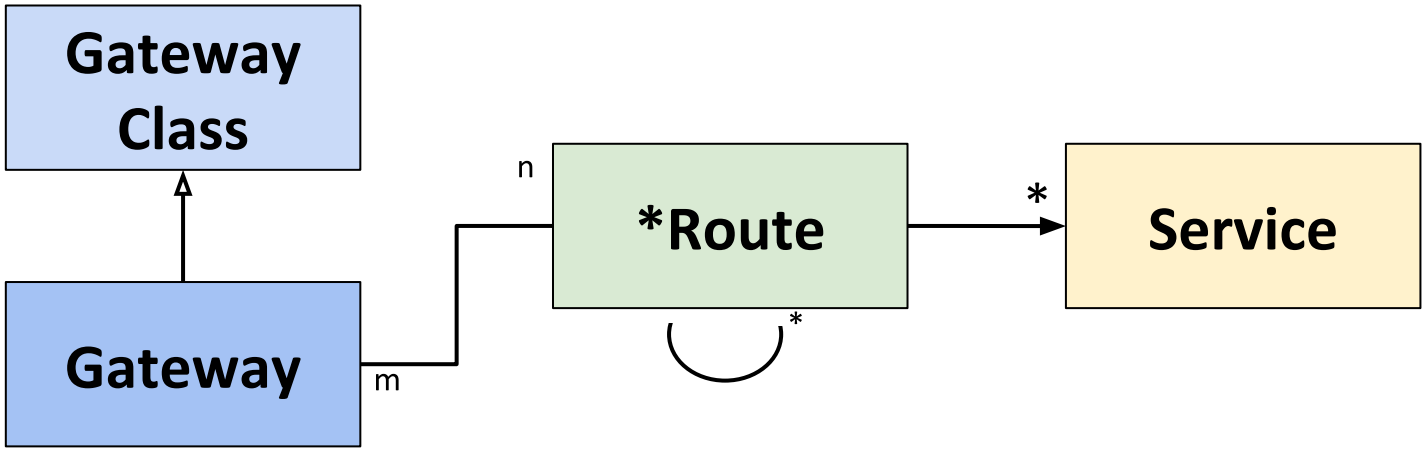
Services V+1



What about Service?

- Grouping, selection
- V1 functionality still works -- but hopefully will not have to add significantly to existing surface area.

Services V+1




```
kind: GatewayClass
name: internet-lb
...
```

```
kind: Gateway
namespace: net-ops
name: the-gateway

class: internet-lb
listeners:
  -   port: 80
      protocol: http
routes:
  -   kind: HTTPRoute
      name: my-app
```

```
kind: HTTPRoute
name: my-app

rules:
  -   path: /my-app
      ...
gateways:
  -   namespace: net-ops
      name: the-gateway
```

```
kind: Service
name: my-app
```

Services V+1

Initial v1alpha1 cut:

- Basic applications, data types
- GatewayClass for interoperation between controllers.
- Gateway + Route
  - HTTP, TCP
  - HTTPS + server certificates+secrets
- Implementability:
  - Merging style (multiple Gateways hosted on single* proxy infra)
  - Provisioning/Cloud (Gateways mapped to externally managed resources)

Wrapping up

Issues

https://issues.k8s.io

File bugs, cleanup ideas, and feature requests

Find issues to help with!

- Especially those labelled “good first issue” and “help wanted”.
- Triage issues (is this a real bug?) labelled “triage/unresolved”.

Enhancements

https://git.k8s.io/enhancements/keps/sig-network

“Enhancements” are user-visible changes (features + functional changes)

- Participate in enhancement dialogue and planning
  - More eyeballs are always welcome
- Submit enhancement proposals of your own!

Get involved!

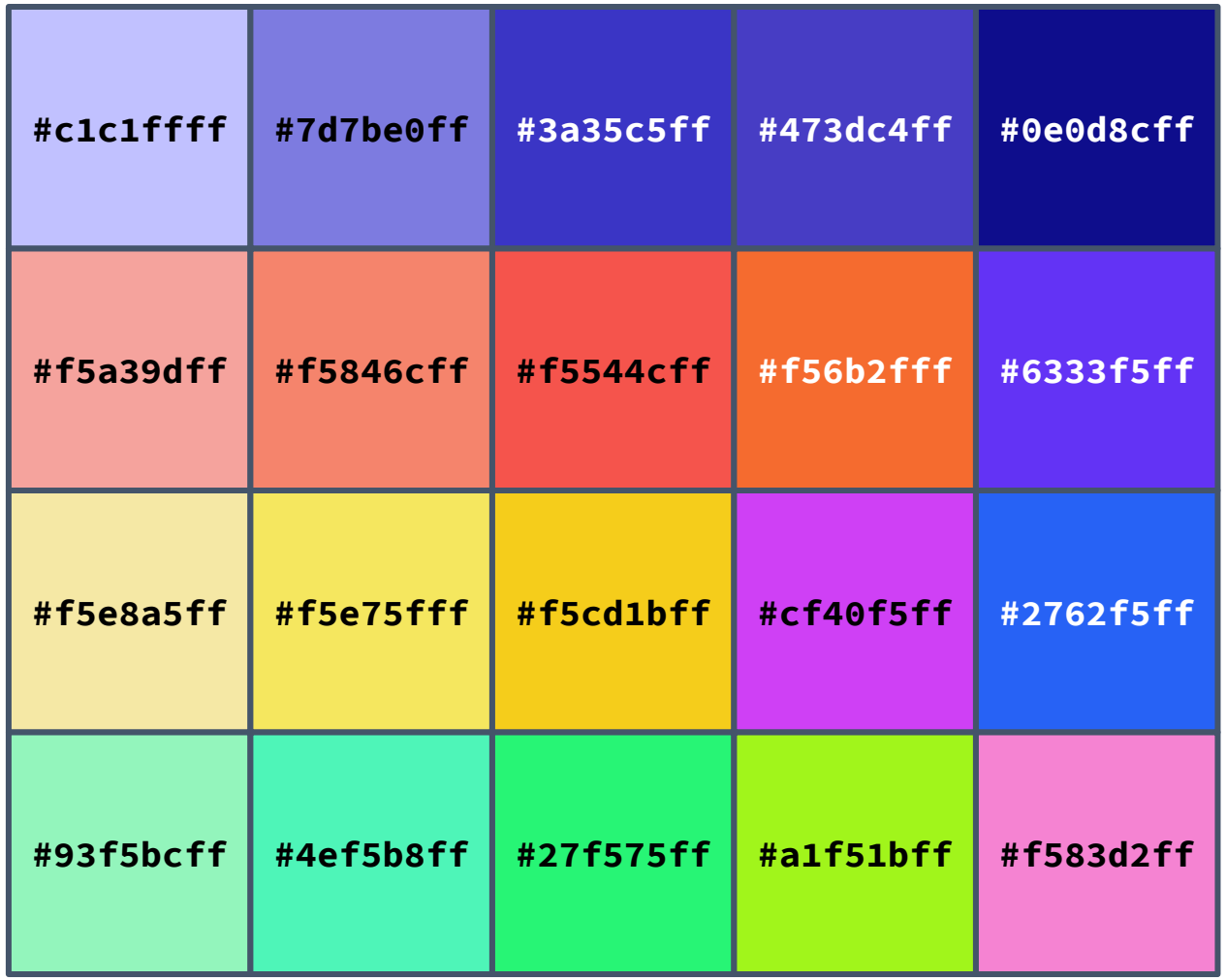
https://git.k8s.io/community/sig-network

Zoom meeting: Every other Thursday, 21:00 UTC

Slack: #sig-network (slack.k8s.io)

Mailing List:
https://groups.google.com/forum/#!forum/kubernetes-sig-network


KEEP CLOUD NATIVE
EVERYWHERE
KubeCon
CloudNativeCon
North America 2020
Virtual


#c1c1ffff
#7d7be0ff
#3a35c5ff
#473dc4ff
#0e0d8cff
#f5a39dff
#f5846cff
#f5544cff
#f56b2fff
#6333f5ff
#f5e8a5ff
#f5e75fff
#f5cd1bff
#cf40f5ff
#2762f5ff
#93f5bcff
#4ef5b8ff
#27f575ff
#a1f51bff
#f583d2ff