

Evolution of kube-proxy

Laurent Bernaille

 @lbernail



Datadog

Monitoring service

Over 350 integrations

Over 1,200 employees

Over 8,000 customers

Runs on millions of hosts

Trillions of data points per day

10000s hosts in our infra

10s of Kubernetes clusters

Clusters from 50 to 3000 nodes

Multi-cloud

Very fast growth

A teal background with a white wireframe cube centered on the left side. The cube is composed of several lines forming its edges, with some lines being thicker than others to create a 3D effect.

What is kube-proxy?

Kube-proxy

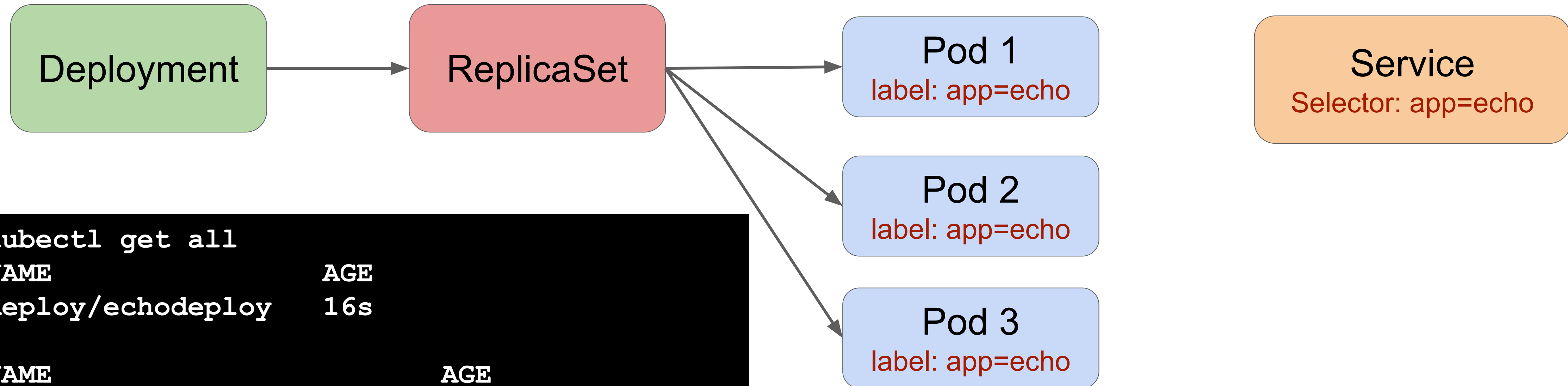
- Network proxy running on each Kubernetes node
- Implements the Kubernetes service abstraction
 - Map service Cluster IPs (virtual IPs) to pods
 - Enable ingress traffic

Deployment and service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echodeploy
  labels:
    app: echo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: echo
  template:
    metadata:
      labels:
        app: echo
    spec:
      containers:
        - name: echopod
          image: lbernail/echo:0.5
```

```
apiVersion: v1
kind: Service
metadata:
  name: echo
  labels:
    app: echo
spec:
  type: ClusterIP
  selector:
    app: echo
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 5000
```

Created resources



```
kubectl get all
```

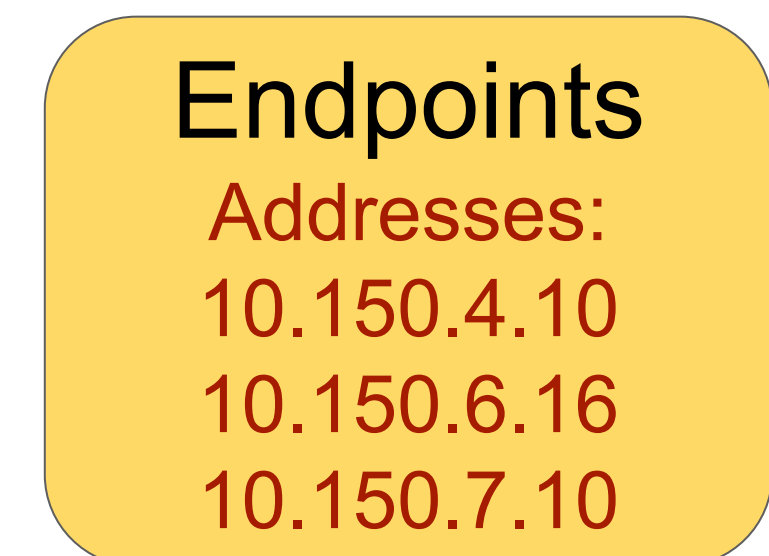
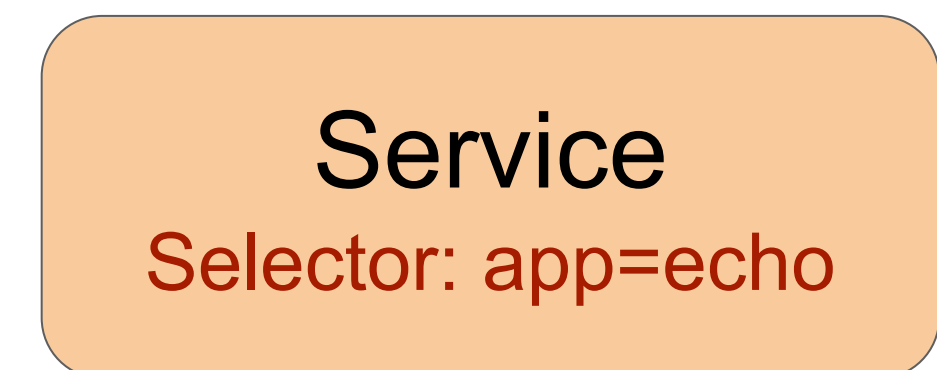
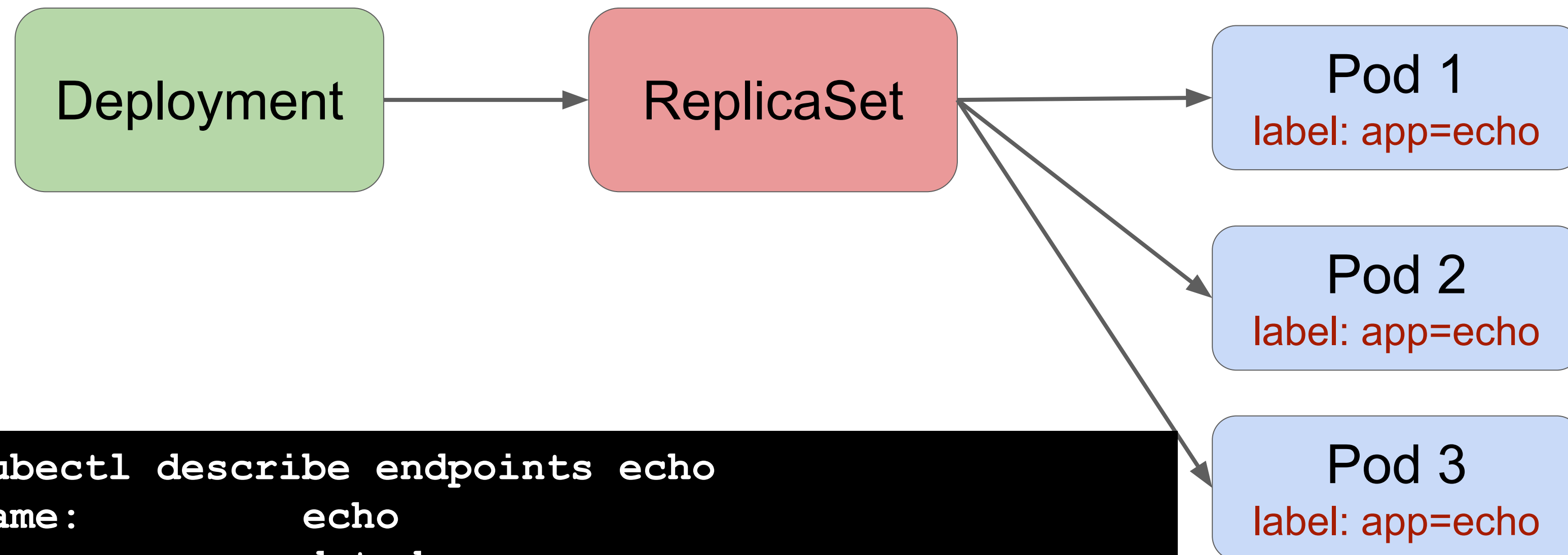
NAME	AGE
deploy/echo deploy	16s

NAME	AGE
rs/echo deploy-75dddcf5f6	16s

NAME	READY
po/echo deploy-75dddcf5f6-jtjts	1/1
po/echo deploy-75dddcf5f6-r7nmk	1/1
po/echo deploy-75dddcf5f6-zvqhv	1/1

NAME	TYPE	CLUSTER-IP
svc/echo	ClusterIP	10.200.246.139

The endpoint object



```
kubectl describe endpoints echo
Name:          echo
Namespace:     datadog
Labels:        app=echo
Annotations:   <none>
Subsets:
  Addresses:    10.150.4.10,10.150.6.16,10.150.7.10
  NotReadyAddresses: <none>
  Ports:
    Name  Port  Protocol
    ----  -
    http  5000  TCP
```

Readiness

- A pod can be started but not ready to serve requests
 - Initialization
 - Connection to backends
- Kubernetes provides an abstraction for this: Readiness Probes

```
readinessProbe:  
  httpGet:  
    path: /ready  
    port: 5000  
  periodSeconds: 2  
  successThreshold: 2  
  failureThreshold: 2
```

Accessing the service

```
kubectl run -it test --image appropriate/curl ash

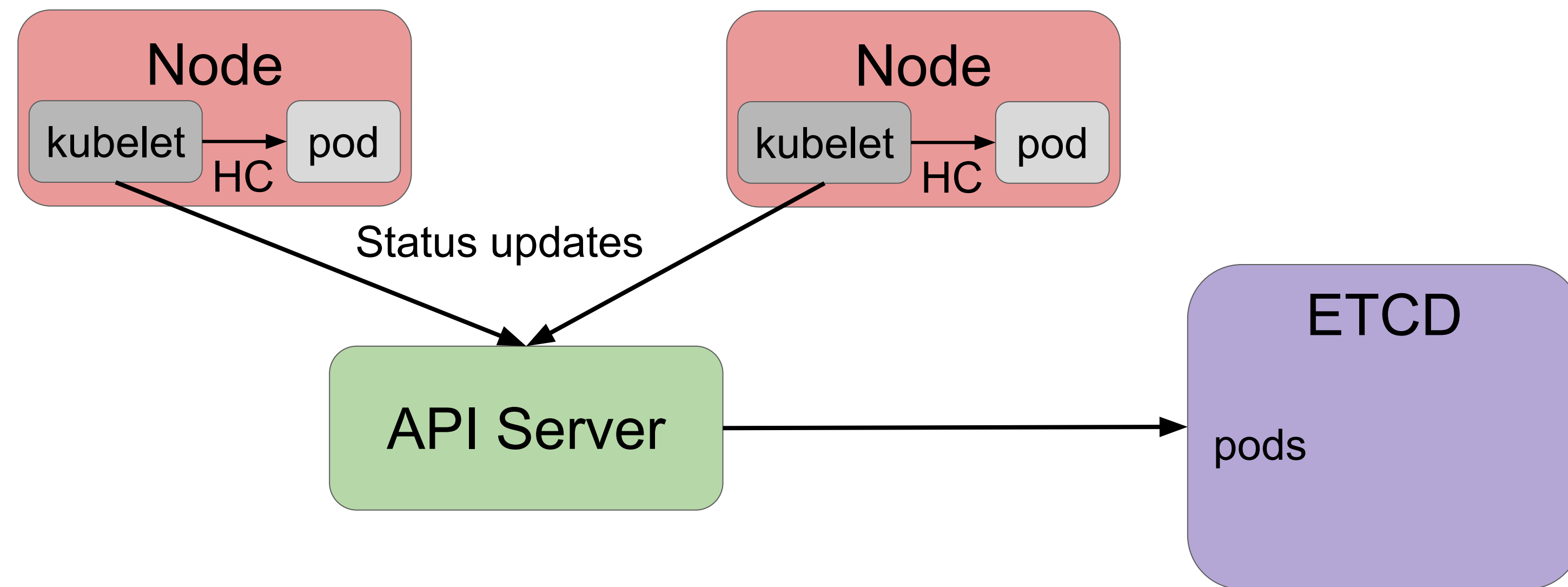
# while true ; do curl 10.200.246.139 ; sleep 1 ; done
Container: 10.150.7.10      | Source: 10.150.6.17      | Version: v2
Container: 10.150.6.16      | Source: 10.150.6.17      | Version: v2
Container: 10.150.4.10      | Source: 10.150.6.17      | Version: v2
Container: 10.150.7.10      | Source: 10.150.6.17      | Version: v2
Container: 10.150.6.16      | Source: 10.150.6.17      | Version: v2
Container: 10.150.4.10      | Source: 10.150.6.17      | Version: v2
Container: 10.150.7.10      | Source: 10.150.6.17      | Version: v2
Container: 10.150.6.16      | Source: 10.150.6.17      | Version: v2
Container: 10.150.4.10      | Source: 10.150.6.17      | Version: v2
```

Endpoint only consist of pods that are Ready

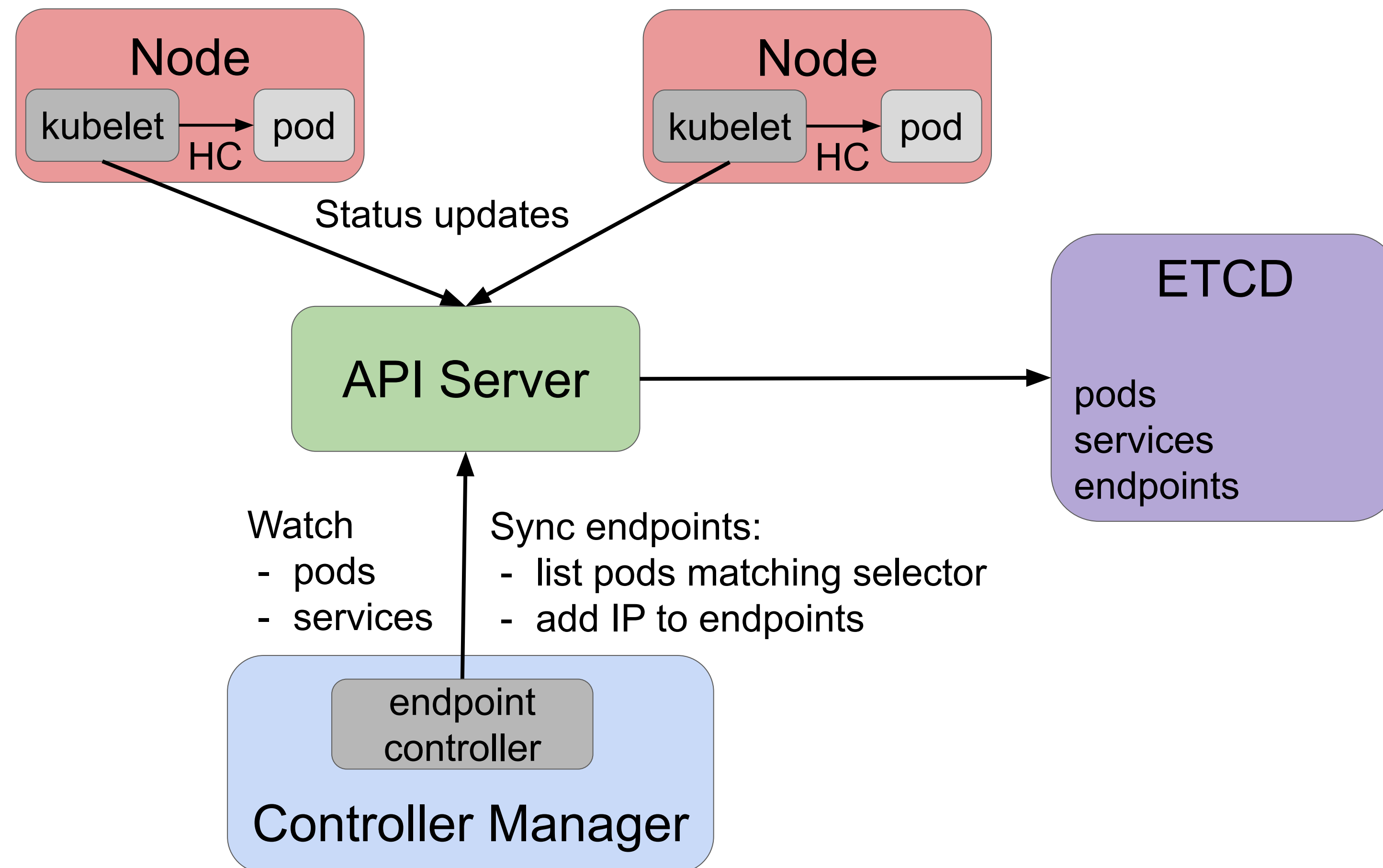


How does it work?

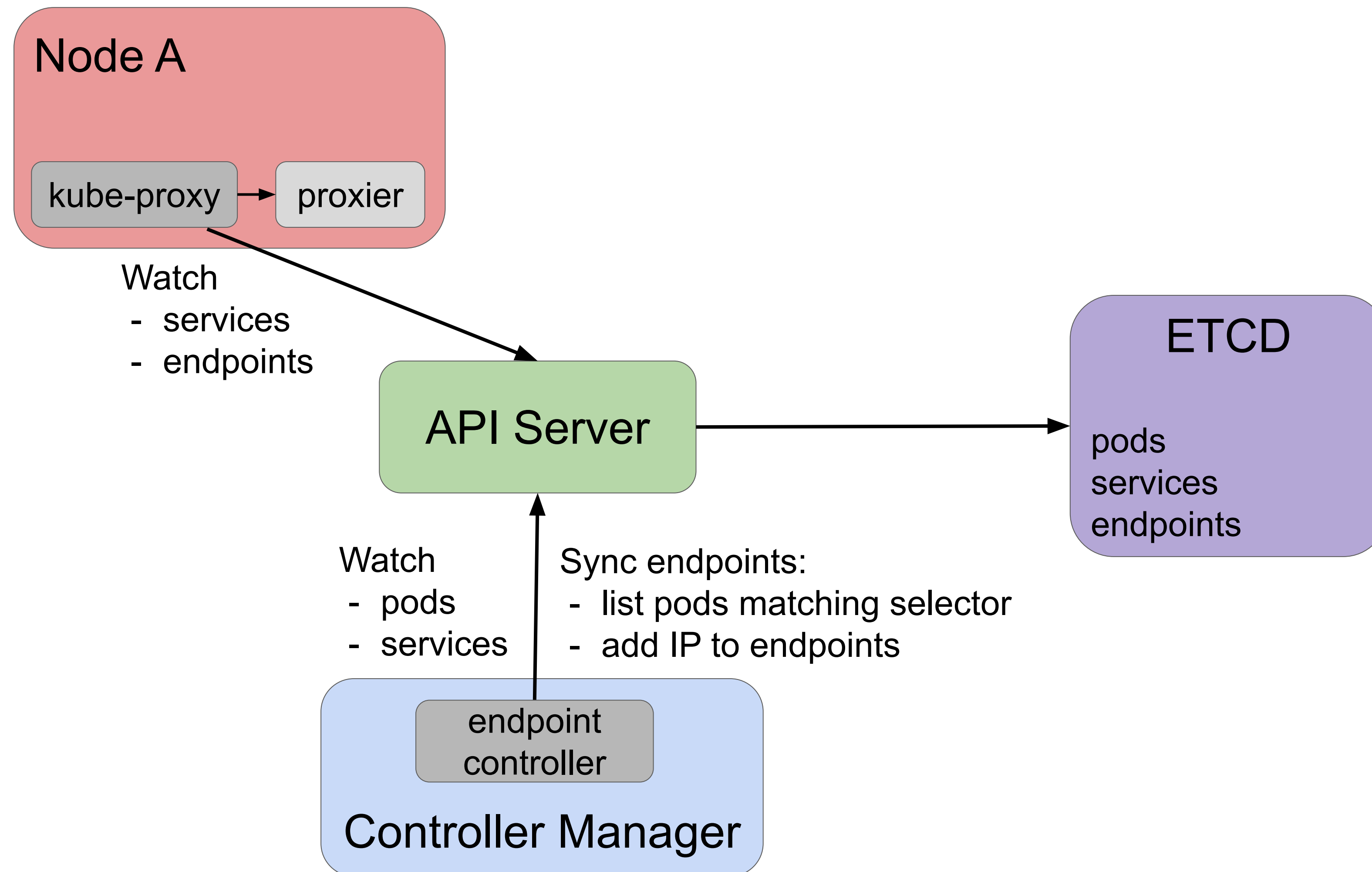
Pod statuses



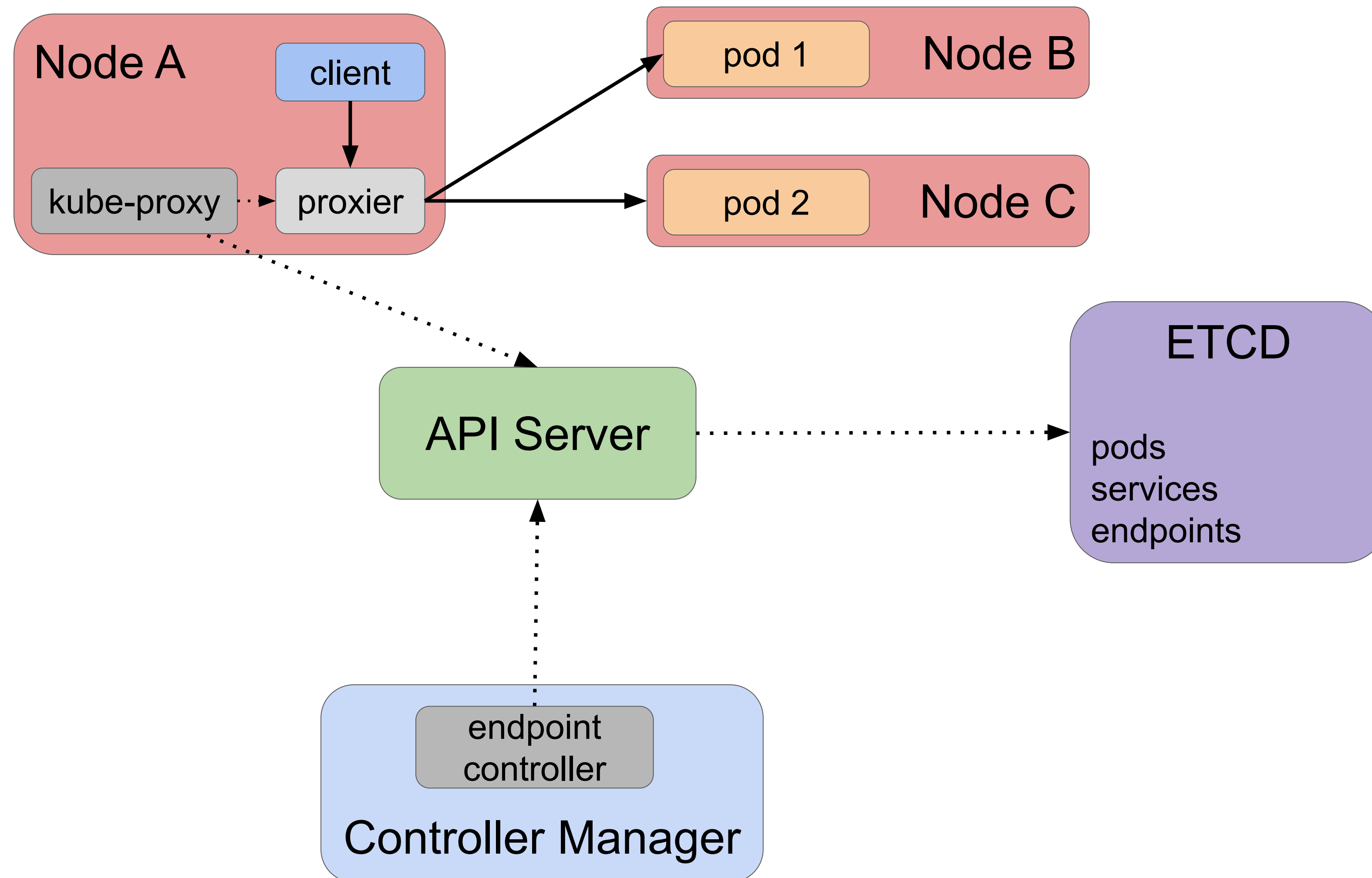
Endpoints



Accessing services



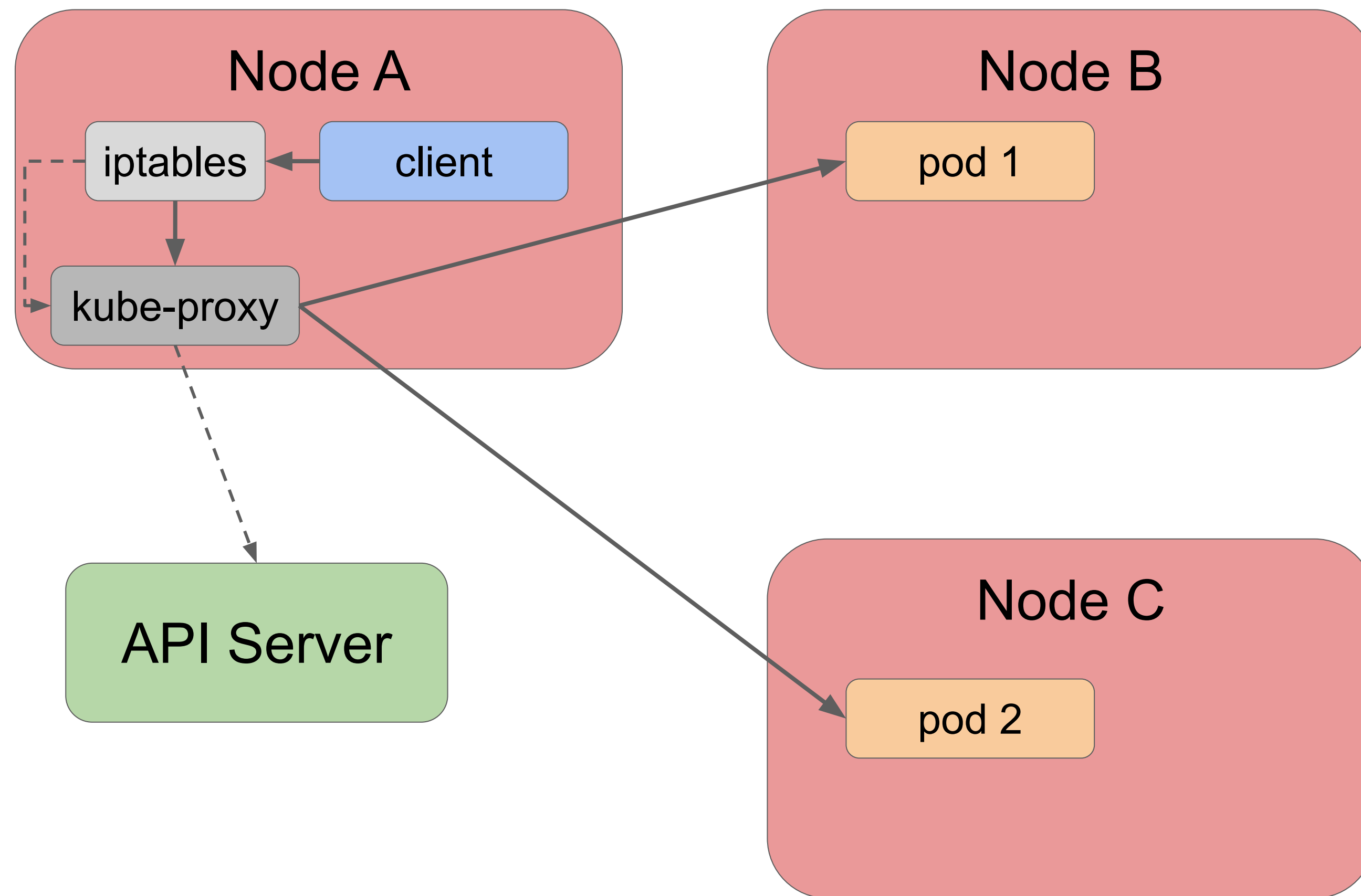
Accessing services



Implementations

userspace

proxy-mode=userspace



kube-proxy binds 1 port per service

Client traffic redirected to kube-proxy

kube-proxy connects to pods

proxy-mode=userspace

PREROUTING

any / any =>

KUBE-PORTALS-CONTAINER

All traffic is processed by kube chains

proxy-mode=userspace

PREROUTING

any / any =>

KUBE-PORTALS-CONTAINER

All traffic is processed by kube chains

KUBE-PORTALS-CONTAINER

any / VIP:PORT => NodeIP:PortX

Portal chain

One rule per service

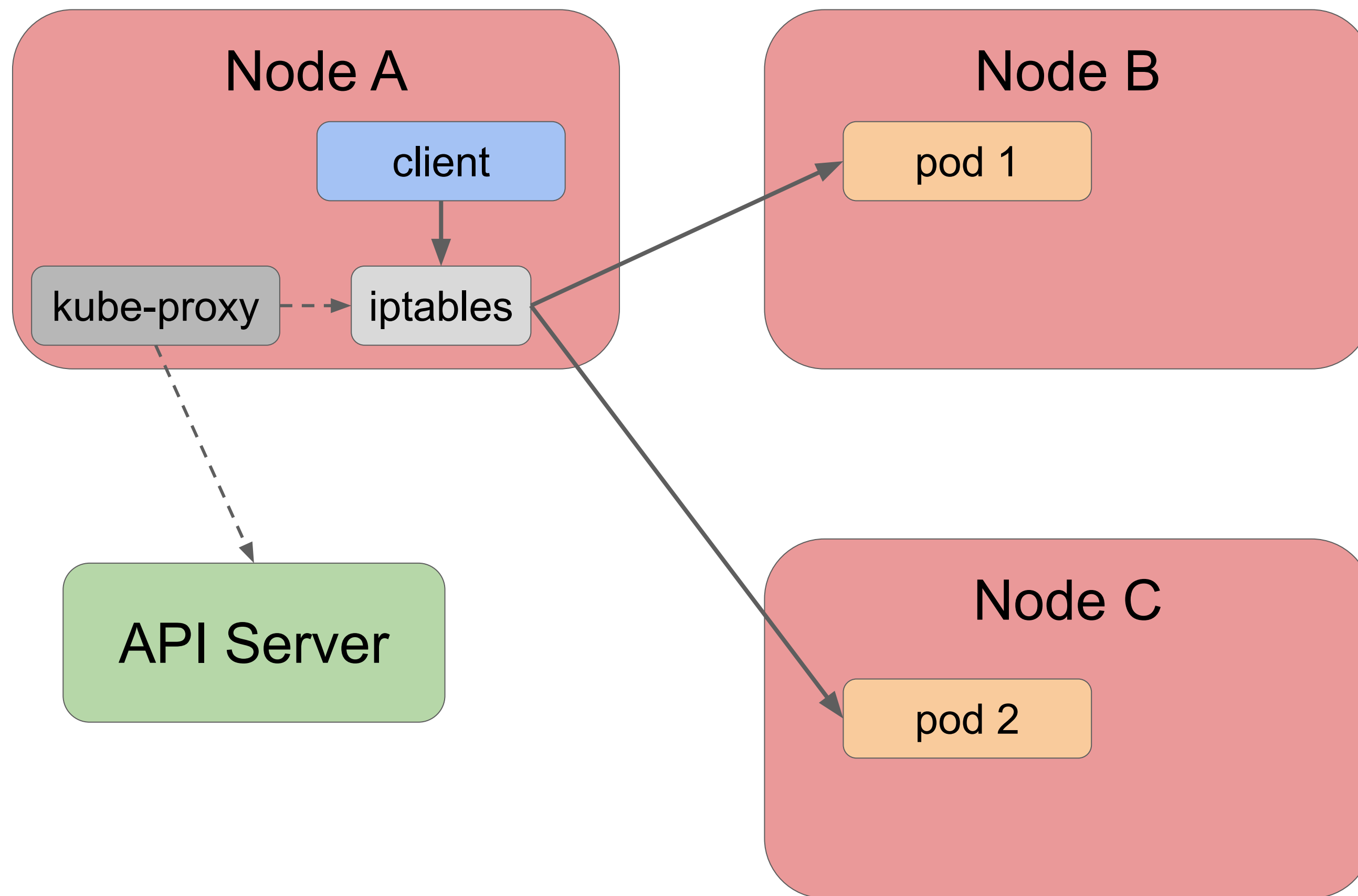
One port allocated for each service (and bound by kube-proxy)

Limitations

- Performance (userland proxying)
- Source IP cannot be kept
- Not recommended anymore
- Since Kubernetes 1.2, default is iptables

iptables

proxy-mode=iptables



Outgoing traffic

1. Client to Service IP
2. DNAT: Client to Pod1 IP

Reverse path

1. Pod1 IP to Client
2. Reverse NAT: Service IP to client

proxy-mode=iptables

PREROUTING / OUTPUT

any / any => KUBE-SERVICES

All traffic is processed by kube chains

proxy-mode=iptables

PREROUTING / OUTPUT

any / any => KUBE-SERVICES

KUBE-SERVICES

any / VIP:PORT => KUBE-SVC-XXX

Global Service chain

Identify service and jump to appropriate service chain

proxy-mode=iptables

PREROUTING / OUTPUT

any / any => KUBE-SERVICES

KUBE-SERVICES

any / VIP:PORT => KUBE-SVC-XXX

KUBE-SVC-XXX

any / any proba 33% => KUBE-SEP-AAA

any / any proba 50% => KUBE-SEP-BBB

any / any => KUBE-SEP-CCC

Service chain (one per service)

Use **statistic** iptables module (*probability of rule being applied*)

Rules are evaluated **sequentially** (hence the 33%, 50%, 100%)

proxy-mode=iptables

PREROUTING / OUTPUT

any / any => KUBE-SERVICES

KUBE-SERVICES

any / VIP:PORT => KUBE-SVC-XXX

KUBE-SVC-XXX

any / any proba 33% => KUBE-SEP-AAA

any / any proba 50% => KUBE-SEP-BBB

any / any => KUBE-SEP-CCC

KUBE-SEP-AAA

endpoint IP / any => KUBE-MARK-MASQ

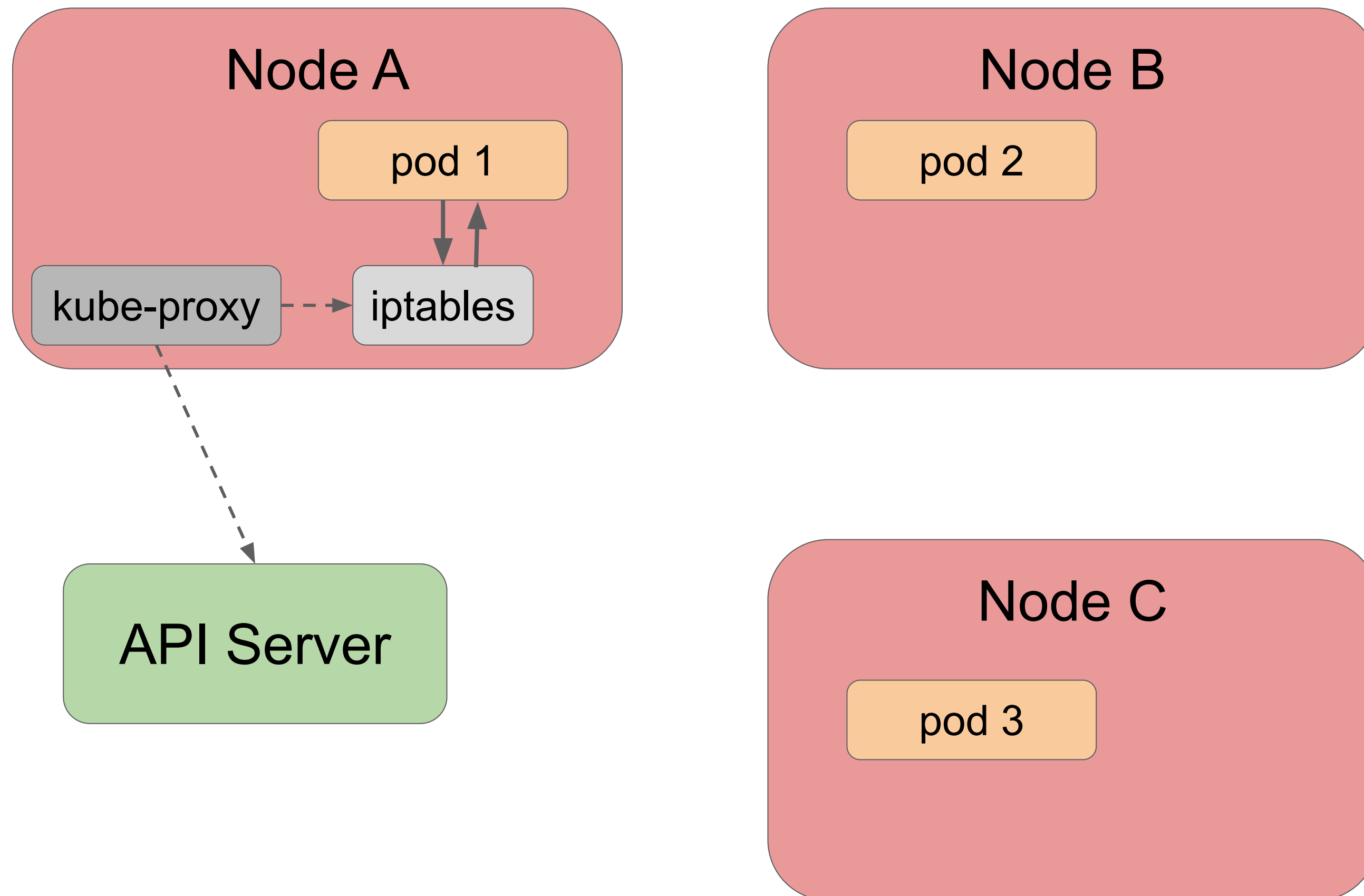
any / any => DNAT endpoint IP:Port

Endpoint Chain

Mark hairpin traffic (client = target) for SNAT

DNAT to the endpoint

Hairpin traffic?



Client can also be a destination

After DNAT:

Src IP= Pod1, Dst IP= Pod1

No reverse NAT possible

=> SNAT on host for this traffic

1. Pod1 IP => SVC IP
2. SNAT: HostIP => SVC IP
3. DNAT: HostIP => Pod1 IP

Reverse path

1. Pod1 IP => Host IP
2. Reverse NAT: SVC IP => Pod1IP

Persistency

```
spec:  
  type: ClusterIP  
  sessionAffinity: ClientIP  
  sessionAffinityConfig:  
    clientIP:  
      timeoutSeconds: 600
```

KUBE-SEP-AAA

endpoint IP / any => KUBE-MARK-MASQ

any / any => DNAT endpoint IP:Port

recent : set rsource KUBE-SEP-AAA

Use “recent” module
Add Source IP to set named KUBE-SEP-AAA

Persistence

KUBE-SVC-XXX

any / any recent: **rcheck KUBE-SEP-AAA => KUBE-SEP-AAA**
any / any recent: rcheck KUBE-SEP-BBB => KUBE-SEP-BBB
any / any recent: rcheck KUBE-SEP-CCC => KUBE-SEP-CCC

Load-balancing rules

Use **recent** module

If Source IP is in set named KUBE-SEP-AAA,
jump to KUBE-SEP-AAA

KUBE-SEP-AAA

endpoint IP / any => KUBE-MARK-MASQ

any / any => DNAT endpoint IP:Port

recent : set rsource KUBE-SEP-AAA

Use “recent” module

Add Source IP to set named KUBE-SEP-AAA

Limitations

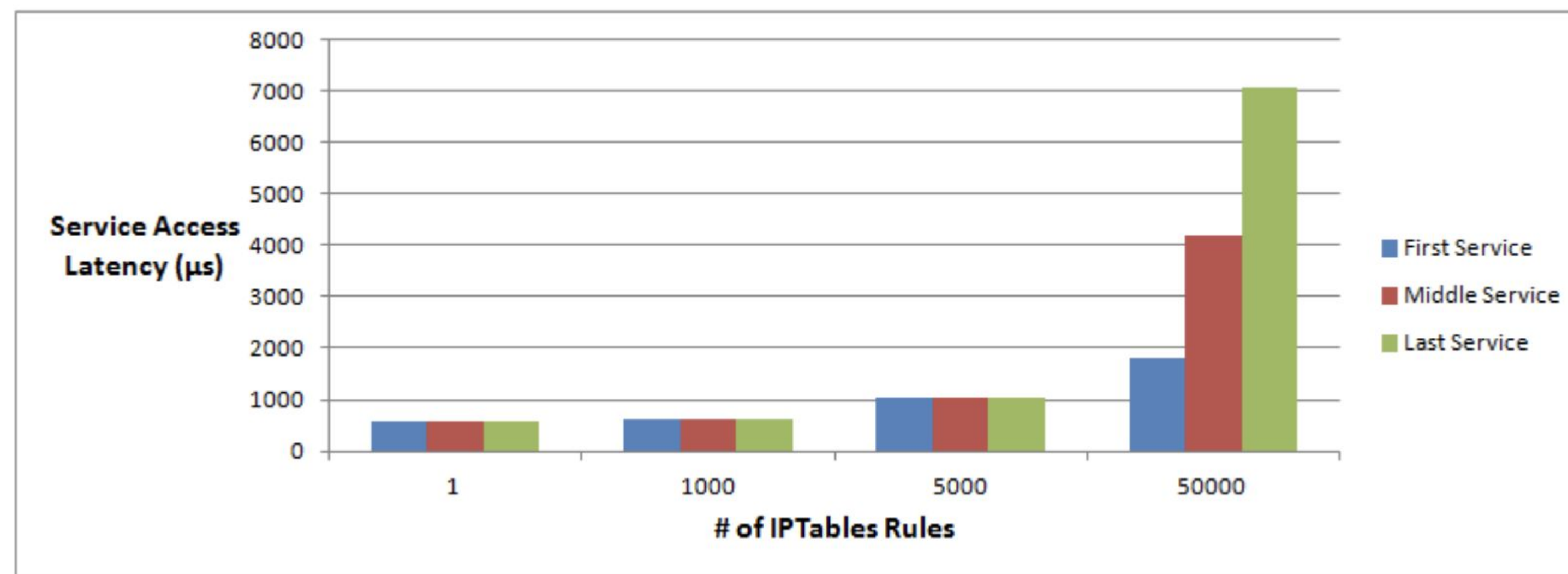
- iptables was not designed for load-balancing
- hard to debug (very quickly, 10000s of rules)
- performance impact
 - control plane: syncing rules
 - data plane: going through rules

IPTables Service Routing Performance

Where is latency generated?

- Long list of rules in a chain
- Enumerate through the list to find a service and pod

In this test, there is one entry per service in KUBE-SERVICES chain.



	1 Service (µs)	1000 Services (µs)	10000 Services (µs)	50000 Services (µs)
First Service	575	614	1023	1821
Middle Service	575	602	1048	4174
Last Service	575	631	1050	7077

“Scale Kubernetes to Support 50,000 Services” Haibin Xie & Quinton Hoole
Kubecon Berlin 2017

Latency to Add IPTables Rules

- Where is the latency generated?
 - not incremental
 - copy all rules
 - make changes
 - save all rules back
 - IPTables locked during rule update
- Time spent to add one rule when there are 5k services (40k rules): 11 minutes
- 20k services (160k rules): 5 hours

“Scale Kubernetes to Support 50,000 Services” Haibin Xie & Quinton Hoole
Kubecon Berlin 2017

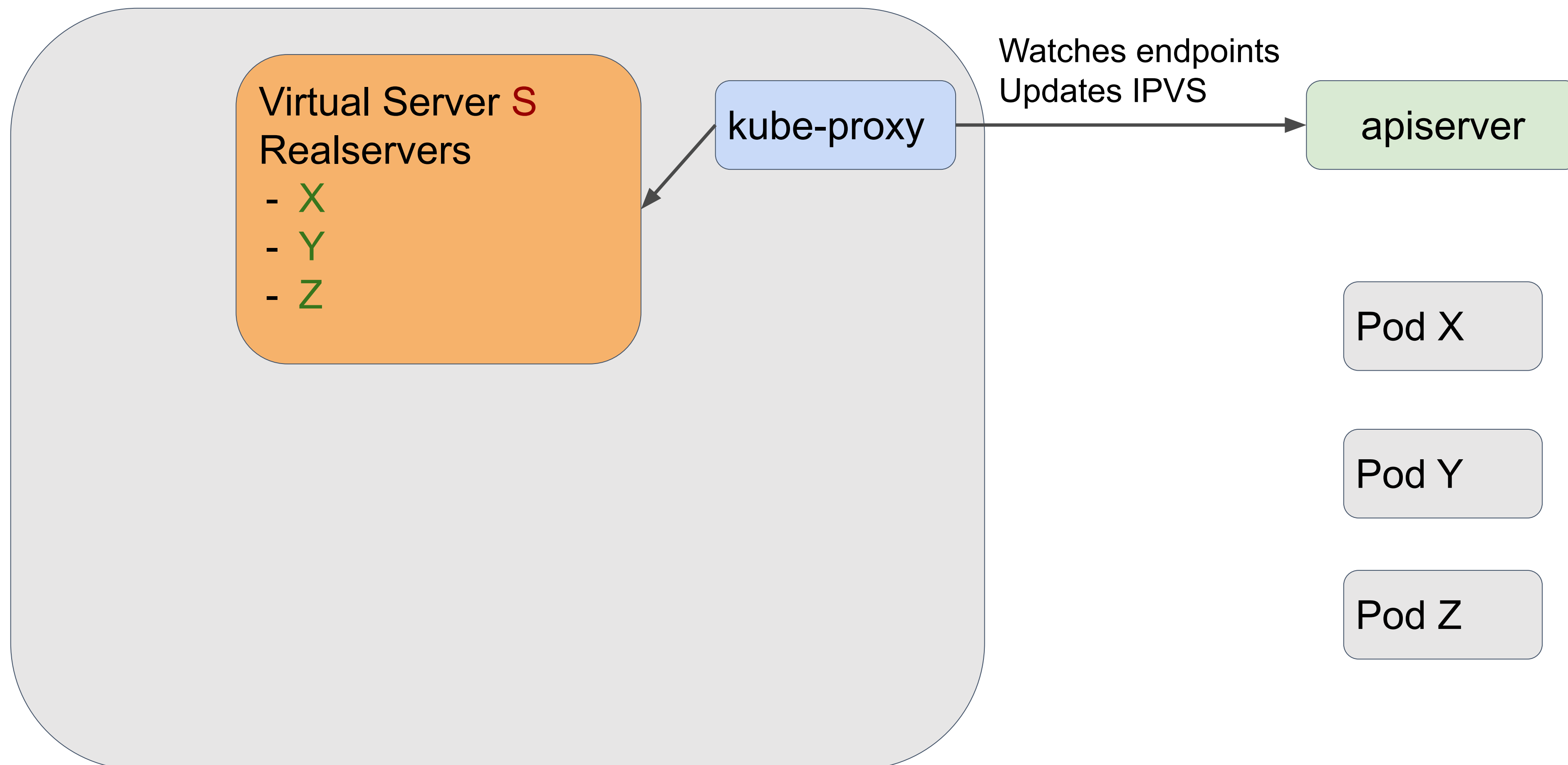
ipvs

proxy-mode=ipvs

- L4 load-balancer build in the Linux Kernel
- Many load-balancing algorithms
- Native persistency
- Fast atomic update (add/remove endpoint)
- Still relies on iptables for some use cases (SNAT)
- GA since 1.11

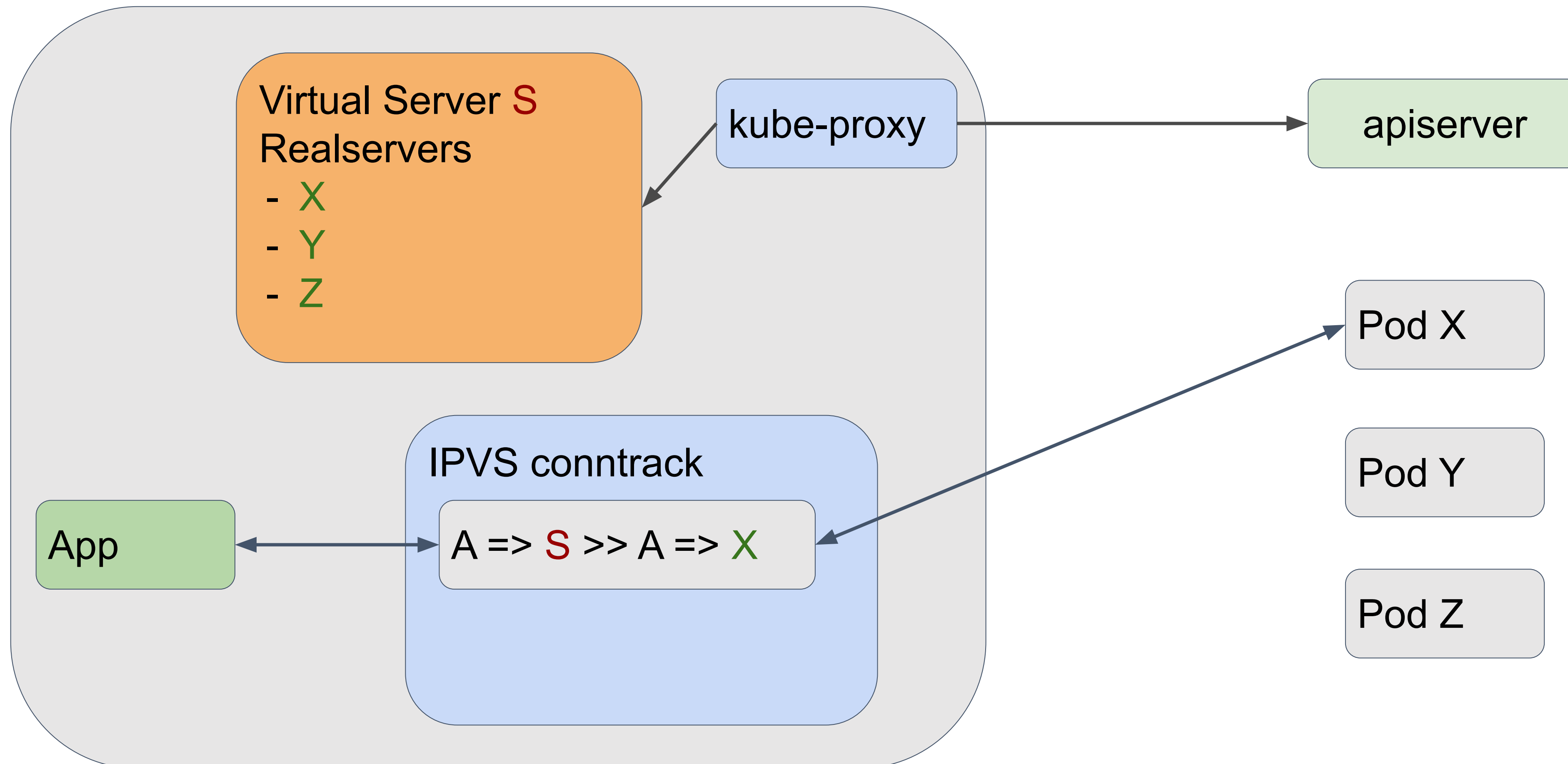
proxy-mode=ipvs

Service ClusterIP:Port **S**
Backed by pod:port **X, Y, Z**



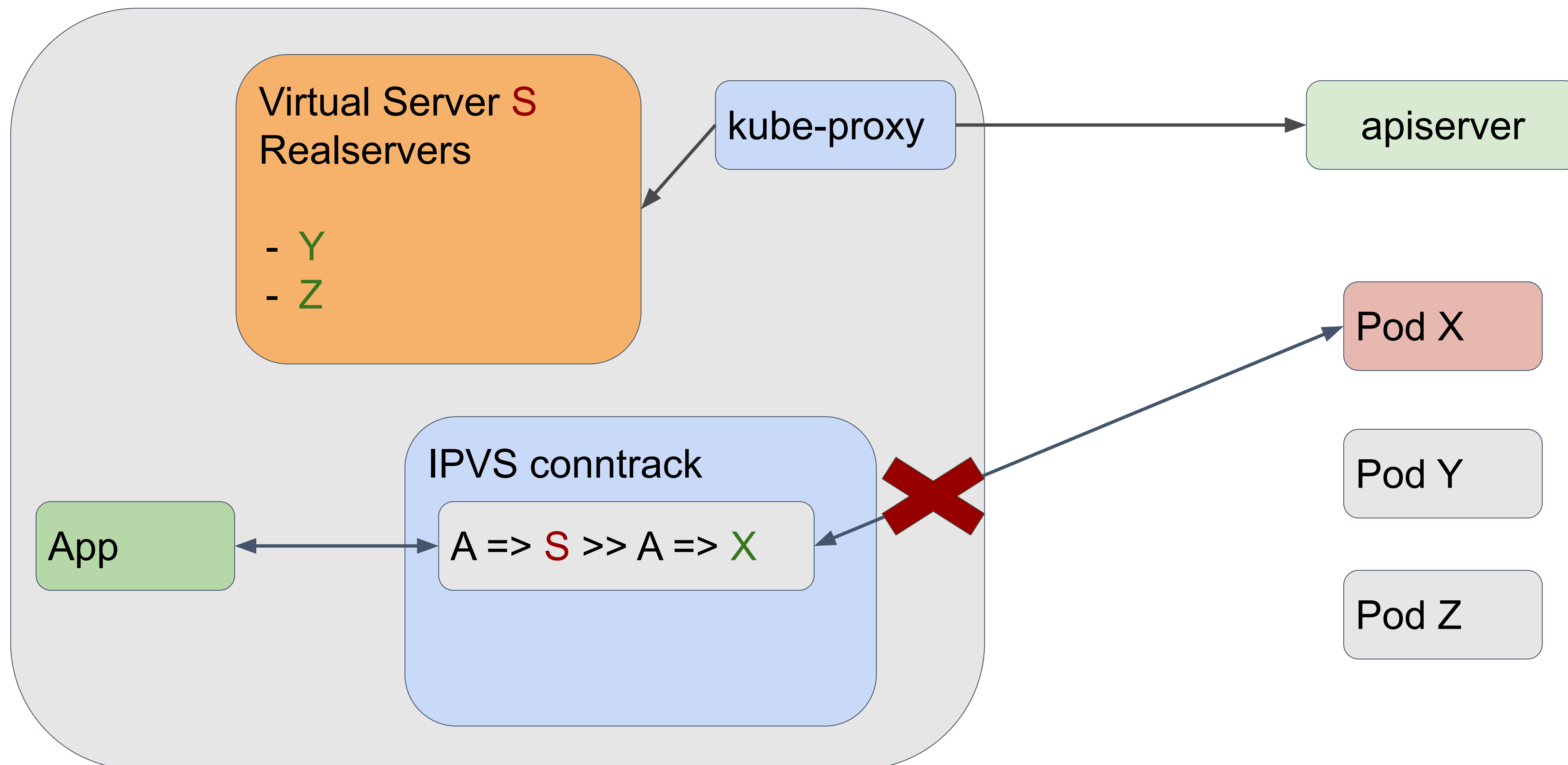
New connection

App establishes connection to **S**
IPVS associates Realserver **X**



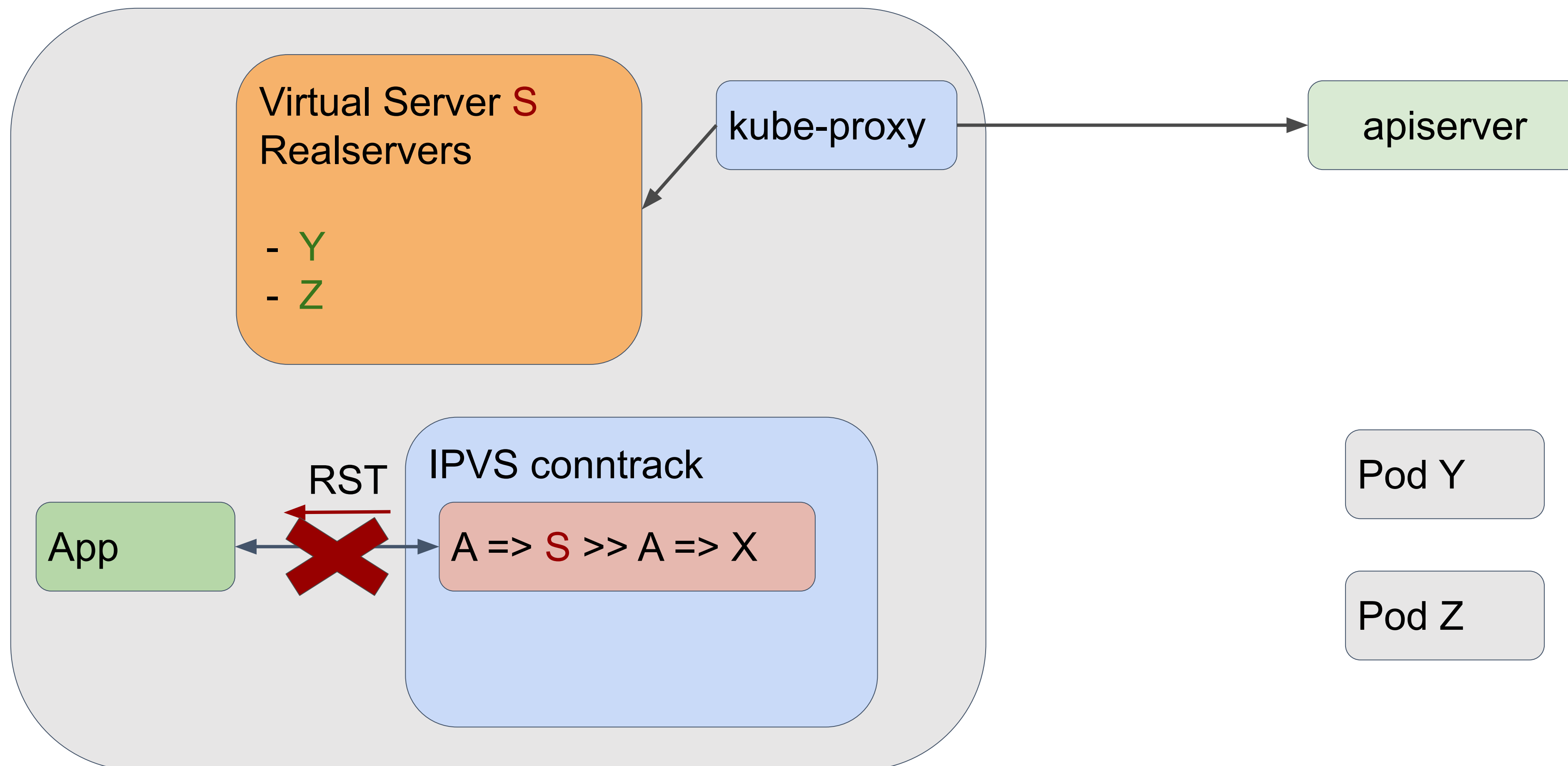
Pod X deleted

Apiserver removes X from S endpoints
Kube-proxy removes X from realservers
Kernel drops traffic (no realserver)



Clean-up

`net/ipv4/vs/expire_nodest_conn`
Delete conntrack entry on next packet
Forcefully terminate (RST)

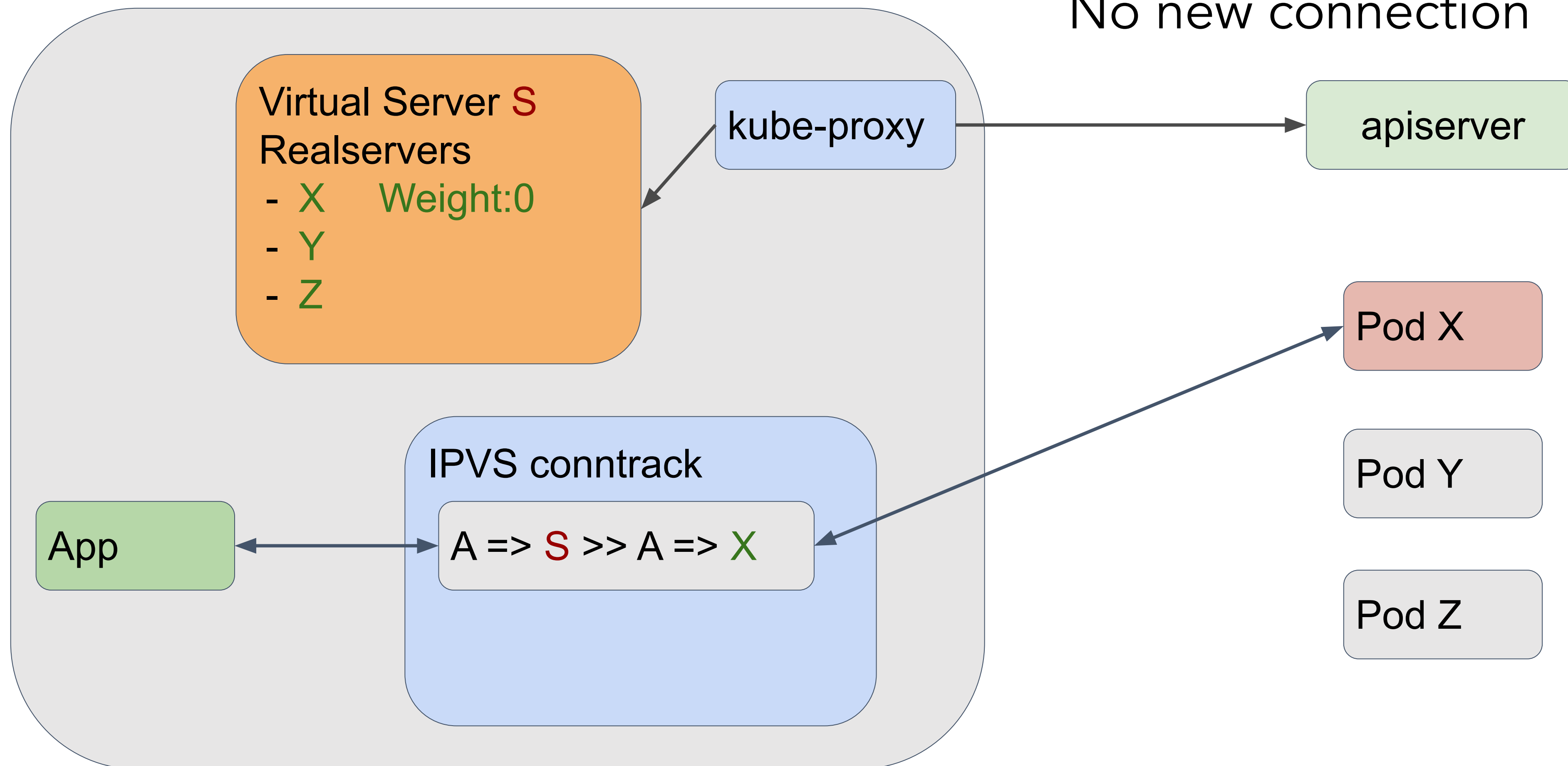


Limit?

- No graceful termination
- As soon as a pod is Terminating connections are destroyed
- Addressing this took time

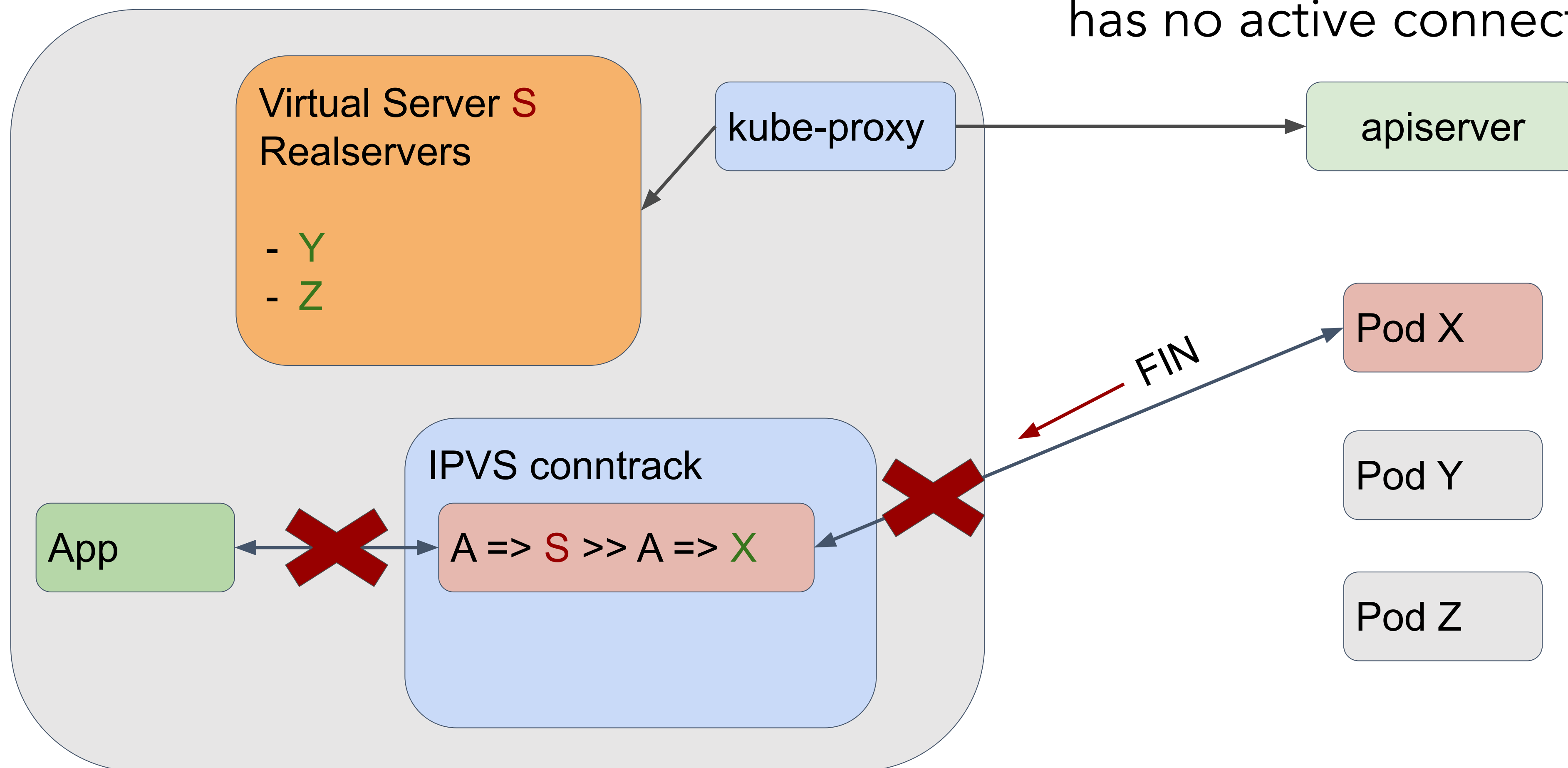
Graceful termination (1.12)

Apiserver removes X from S endpoints
Kube-proxy sets Weight to 0
No new connection

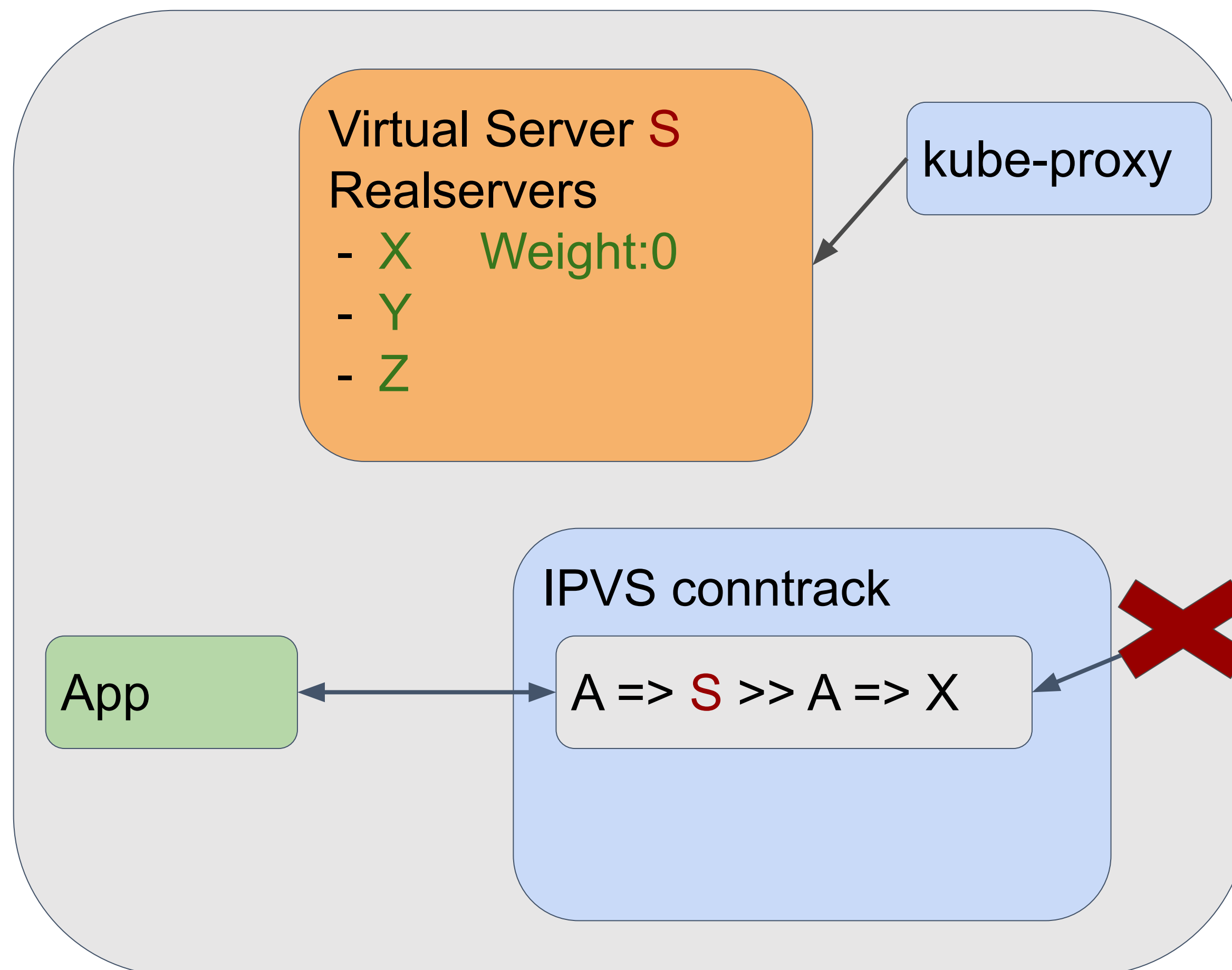


Garbage collection

Pod exits and sends FIN
Kube-proxy removes realserver when it
has no active connection



What if X doesn't send FIN?



Conntrack entries expires after 900s
If A sends traffic, it never expires
Traffic blackholed until App detects issue
~15mn
> Mitigation: lower tcp_retries2

Pod Y

Pod Z

IPVS connection tracking

- Default timeouts
 - tcp / tcpfin : 900s / 120s
 - udp: 300s
- Under high load, ports can be reused fast and keep entry active
 - Especially with `conn_reuse_mode=0`
- Very bad for DNS => No graceful termination for UDP

IPVS connection tracking

- Very recent feature: allow for configurable timeouts
 - Ideally we'd want:
 - Terminating pod: set weight to 0
 - Deleted pod: remove backend
- > This would require an API change for Endpoints

IPVS status

- Works pretty well at large scale
- Not 100% feature parity (a few edge cases)
- Tuning the IPVS parameters took time
- Improvements under discussion

The background is a solid purple color. It features several thin, light purple lines: a diagonal line running from the top-left towards the bottom-right, and a vertical line on the right side. A white square is positioned in the lower-right area, with its top-left corner at the intersection of the two light purple lines.

Common challenges

Control plane scalability

- On each update to a service (new pod, readiness change)
 - The full endpoint object is recomputed
 - The endpoint is sent to all kube-proxy
- Example of a single endpoint change
 - Service with 2000 endpoints, ~100B / endpoint, 5000 nodes
 - Each node gets $2000 \times 100\text{B} = 200\text{kB}$
 - Apiservers send $5000 \times 200\text{kB} = 1\text{GB}$
- Rolling update?
 - Total traffic => $2000 \times 1\text{GB} = 2\text{TB}$

=> Endpoint Slices

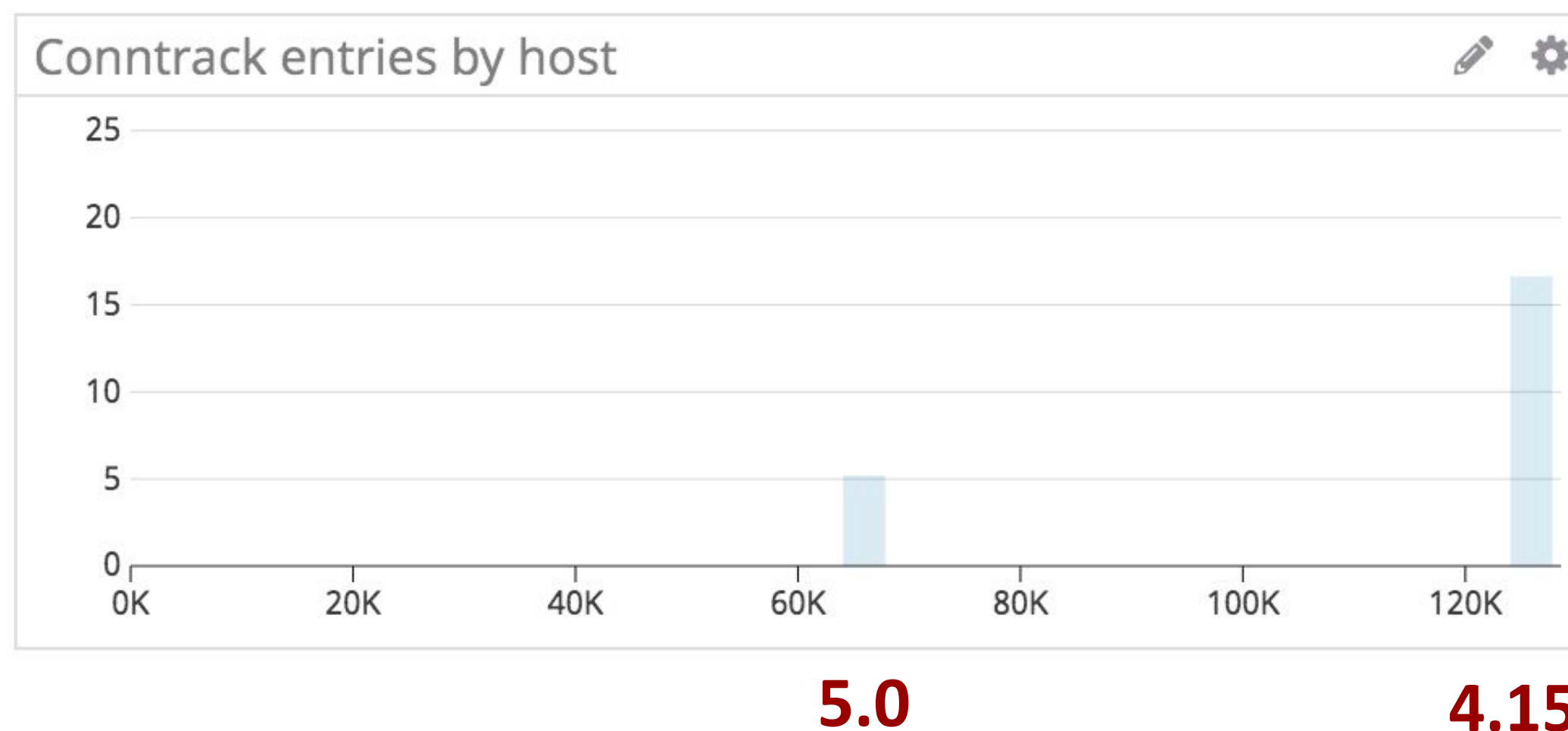
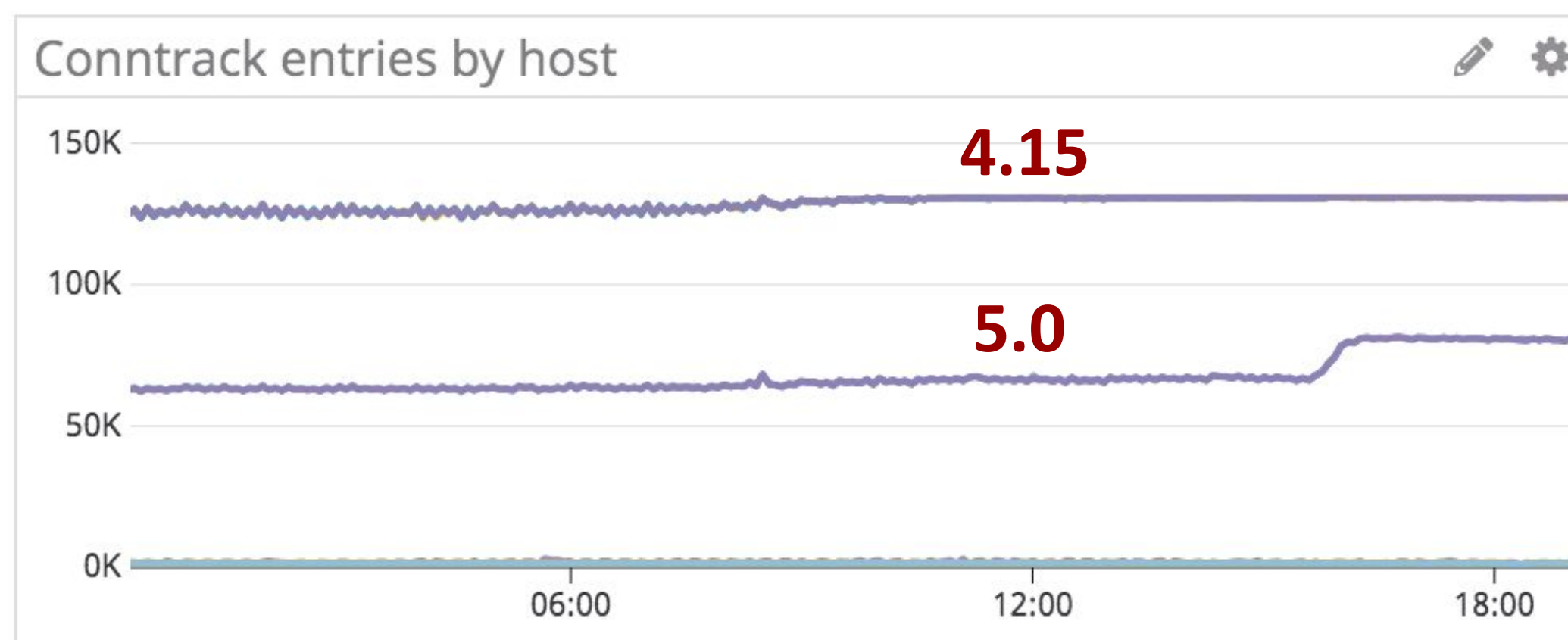
- Service potentially mapped to multiple EndpointSlices
- Maximum 100 endpoints in each slice
- Much more efficient for services with many endpoints
- Beta in 1.17

Conntrack size

- kube-proxy relies on the conntrack
 - Not great for services receiving a lot of connections
 - Pretty bad for the cluster DNS
- DNS strategy
 - Use node-local cache and force TCP for upstream
 - Much more manageable

Conntrack and UDP

- Very good surprise with Kernel 5.0+



netfilter: conntrack: udp: only extend timeout to stream mode after 2s

Currently DNS resolvers that send both A and AAAA queries from same source port can trigger stream mode prematurely, which results in non-early-evictable conntrack entry for three minutes, even though DNS requests are done in a few milliseconds.

Add a two second grace period where we continue to use the ordinary 30-second default timeout. Its enough for DNS request/response traffic, even if two request/reply packets are involved.

ASSURED is still set, else conntrack (and thus a possible NAT mapping ...) gets zapped too in case conntrack table runs full.

Signed-off-by: Florian Westphal <fw@strlen.de>

Signed-off-by: Pablo Neira Ayuso <pablo@netfilter.org>

netfilter: conntrack: udp: set stream timeout to 2 minutes

We have no explicit signal when a UDP stream has terminated, peers just stop sending.

For suspected stream connections a timeout of two minutes is sane to keep NAT mapping alive a while longer.

It matches tcp conntracks 'timewait' default timeout value.

Signed-off-by: Florian Westphal <fw@strlen.de>

Signed-off-by: Pablo Neira Ayuso <pablo@netfilter.org>

🔗 master (#78) 📦 v5.5 ... v5.0-rc1

The background is a solid purple color. It features a decorative graphic consisting of a thin, light purple line that starts from the top-left corner, extends diagonally down towards the bottom-right, and then turns vertically down to the bottom edge. Another thin, light purple line runs horizontally from the left edge to the point where the first line turns vertically.

Recent features

New features

- Dual-stack support
 - Alpha in 1.16
- Topology aware routing
 - Use topology keys to route to services
 - Alpha in 1.17

The background is a solid purple color. A white diagonal line runs from the top-left corner towards the bottom-right, ending at a white right-angled triangle. The triangle's hypotenuse is the diagonal line, and its right angle is at the bottom-right corner. The word "Conclusion" is written in white, sans-serif font, positioned to the left of the triangle.

Conclusion

Conclusion

- kube-proxy works at significant scale
- A lot of ongoing efforts to improve kube-proxy
- iptables and IPVS are not perfect matches for the service abstraction
 - Not designed for client-side load-balancing
- Very promising eBPF alternative: Cilium
 - No need for iptables / IPVS
 - Works without conntrack
 - See Daniel Borkmann's talk

Thank you

