L4LB for Kubernetes: Theory and Practice with Cilium+BGP+ECMP

Published at 2020-04-10 | Last Update 2020-08-22

As networking engineers that are responsible for on-premises Kubernetes clusters, you may be throwned a practical business need: **exposing services inside a Kubernetes cluster to the outside** (e.g. to another Kubernetes cluster or lagacy infra). As shown in the picture below:

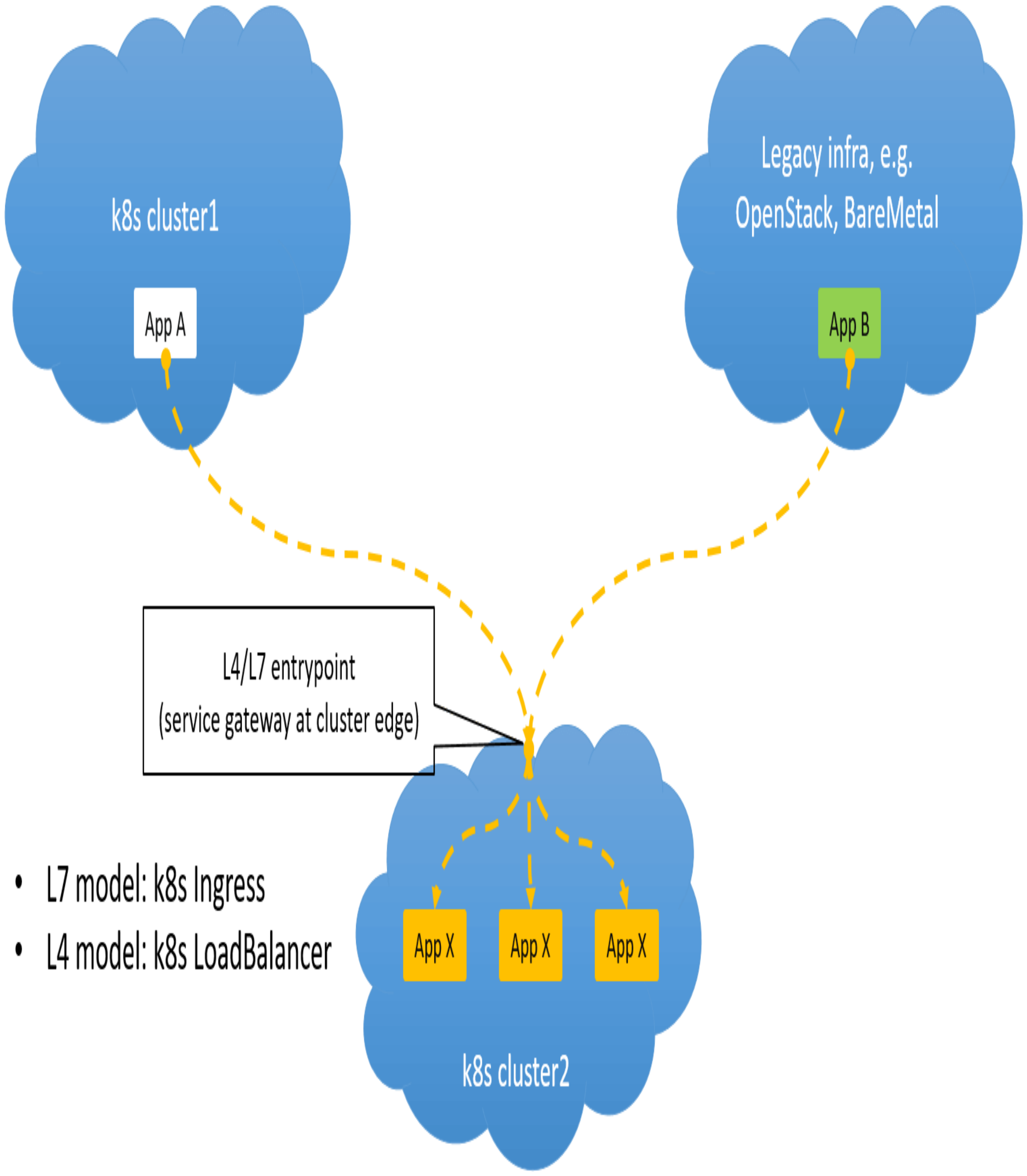

Fig. 0.1 Exposing a Kubernetes service to outside world

Kubernetes provides two **models** for this scenario:

- Ingress: for exposing services via layer 7
- LoadBalancer: for exposing services via layer 4

But, it **leaves the implementations to each vendor**. For example, if you are on AWS, it provides ALB and ELB, each for layer 7 and layer 4.

This post reasons about the design of the latter one, and provides a simple implementation by combining open source softwares.

1. Problem Definition

Suppose you deployed several **DNS servers** (pods) inside a Kubernetes cluster, intended for providing **corp-wide** DNS service.

Now the question is: **how would your legacy applications** (e.g. running in bare metal or VM) or **applications in other Kubernetes clusters access this DNS service**? As depicted in Fig 1.1:

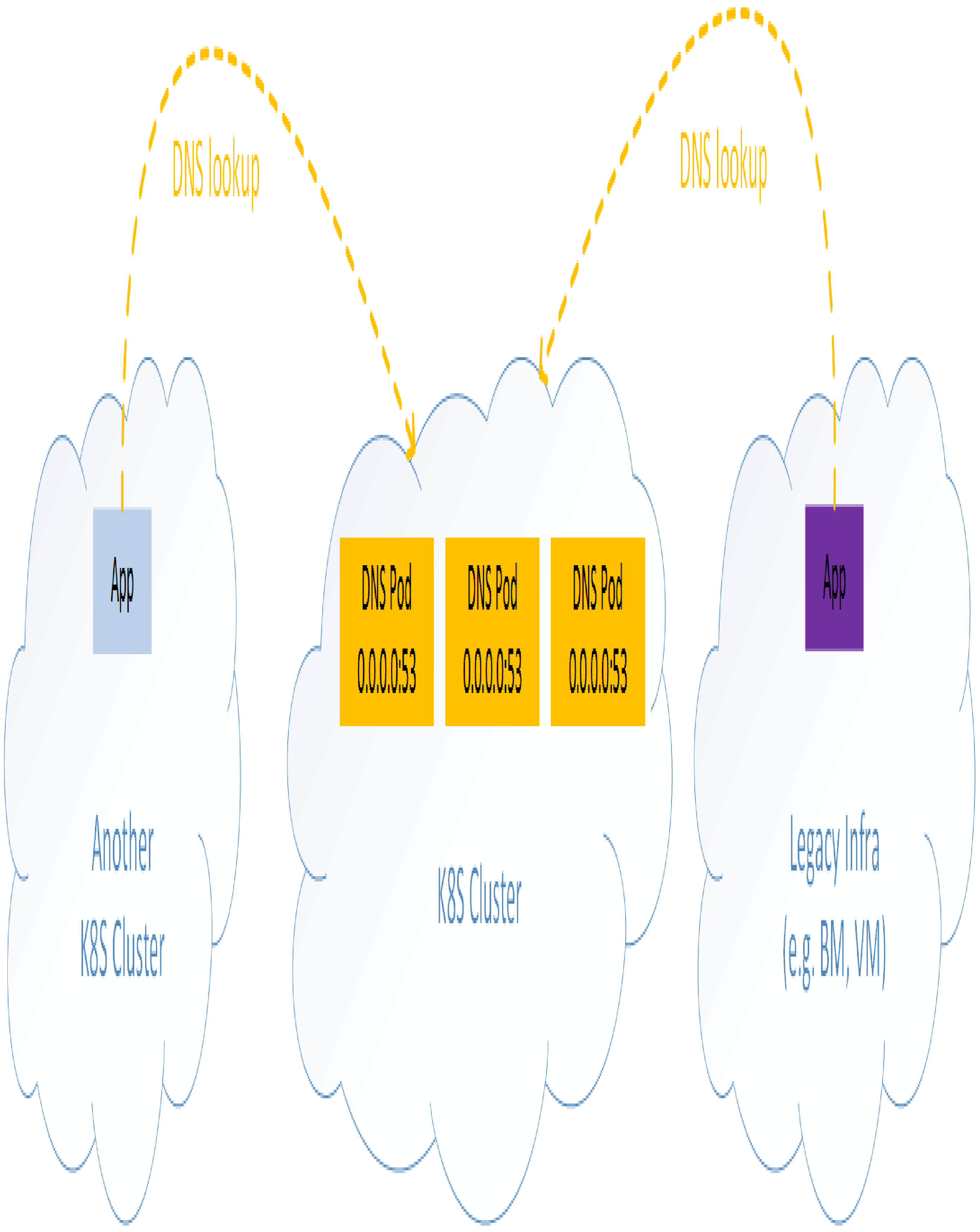

Fig. 1.1 Exposing DNS service in a Kubernetes cluster to outside world

Some tips:

1. DNS provides service with UDP (or TCP) port `53` (thus **layer 4**).
2. DNS pod is **stateless**, here "stateless" means all instances are identical and active, rather than active-backup or master-slave.

2. Requirement Analysis

2.1 L4LB Model

At first glance, it seems that the classic layer 4 load balancer (L4LB) model could be used to address this problem, that is:

1. Assign a VIP to the DNS cluster, all clients access the service by VIP.
2. Forward traffic from VIP to a specific backend with load balancing algorithms.

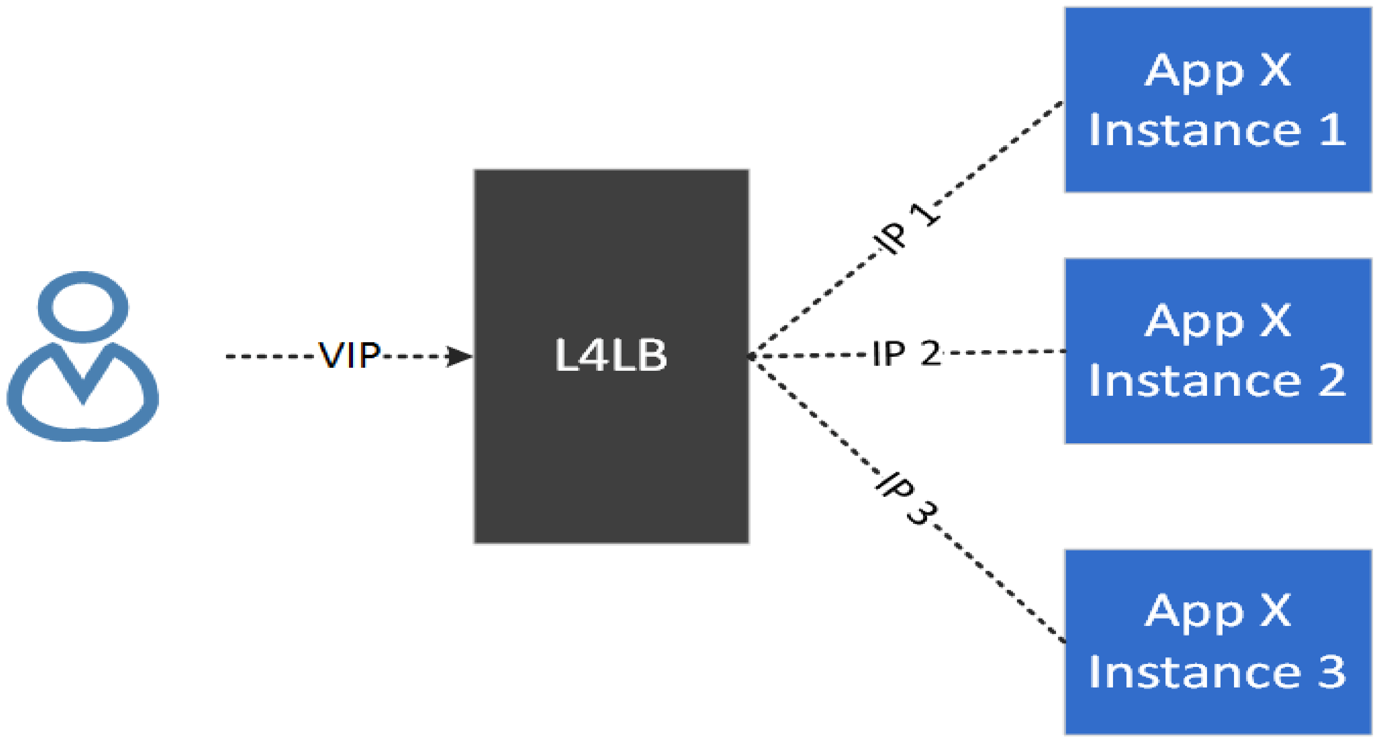

Fig. 2.1 The classic L4LB model

2.2 Special Considerations

Well, this's on the right way, but lacks some important aspects.

First, **our backends (Kubernetes pods) are very dynamic and subject to frequent changes**: kinds of failures (or internal scheduling) may trigger Kubernetes to kill or reschedule a pod. This may happen frequently;

Secondly, the **IP address of a Pod is not immutable**. In comparison, traditional L4LBs favor fixed IP backends, they just pull in or pull out existing backends by health check status.

Thirdly, **backends may be scaled up/down according to capacity plans**, these backend changes should be timely reflected on L4LB, and be transparent to your users/clients without human interventions (some kind of service discovery).

This makes the L4LB a very unique role:

- On the one hand, it must be reachable from the outside world (while most Kubernetes resources couldn't be).
- On the other hand, it falls into a Kubernetes cluster - as it must listen to Kubernetes resource changes in order to update its forwarding rules (from VIP to backends).

2.3 Technical Requirements Summary

Now we summarize the above into following **technical needs**:

1. **Persistent/invariant L4 entrypoint for accessing a Kubernetes service from the outside world**: the entrypoint, e.g. VIP, should be unique and invariant, hiding backend changes from users/clients.
2. **Load balancing**: load balance requests between multiple backends.
3. **Reactive to backend changes**: responsive to backend changes, e.g. instances scale up/down.

For production ready, it also involves the 4th requirement:

\ 4. **High availability**: both software and hardwares (if there are).

In this article, we show such a (simple) design/implementation for on-premises bare metal kubernetes clusters.

3. A L4LB Solution

This post assumes the underlying physical network is a Spine-Leaf architecture (physical network topology only impacts ECMP in this post).

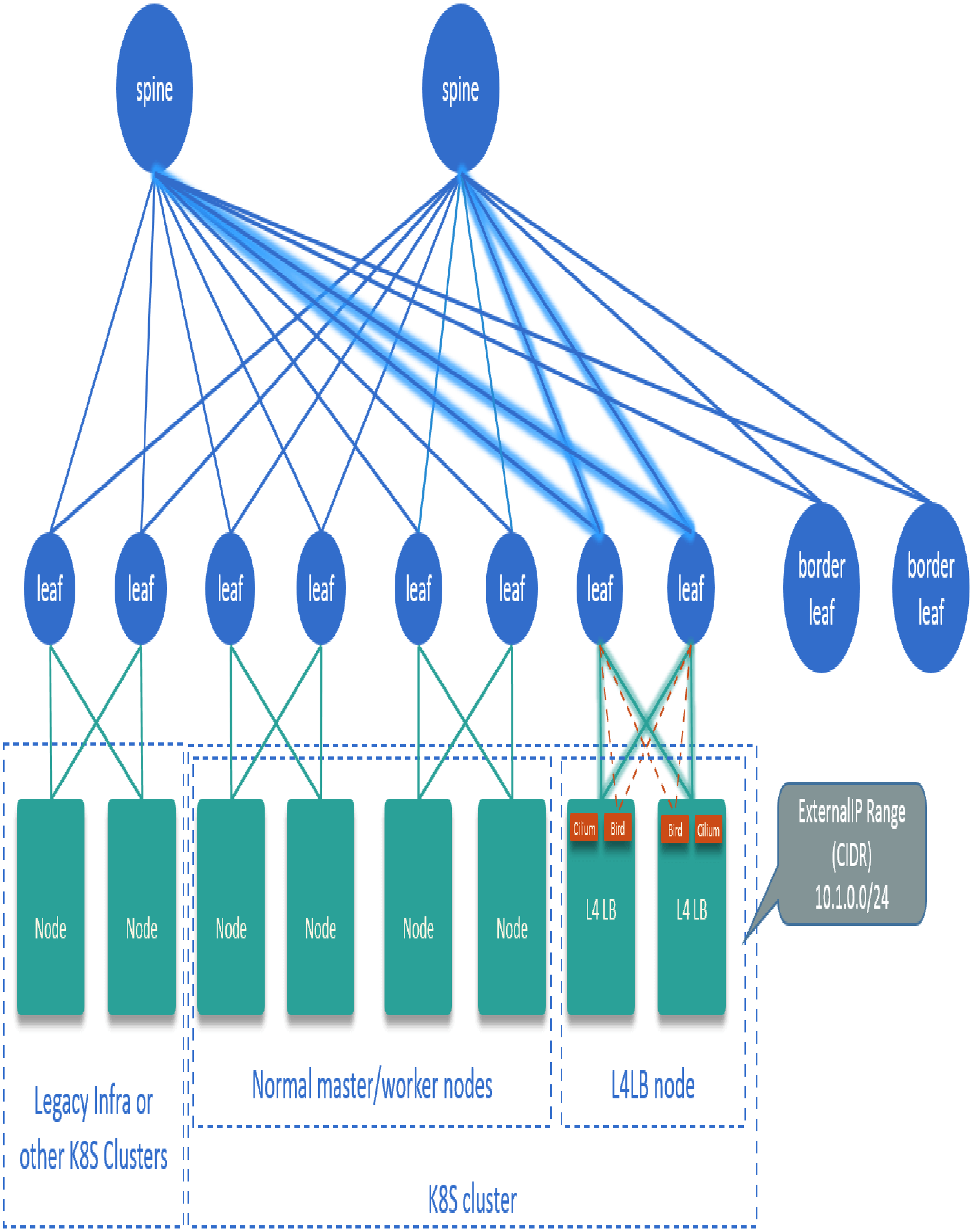

Fig. 3.1 Topology of our L4LB solution

The overall design is depicted in Fig 3.1:

1. Dedicate several nodes as **L4LB nodes**.

2. Running a **BGP agent** on each L4LB node, announcing a specific CIDR to datacenter network, e.g. `10.1.0.0/24` . IPs in this CIDR will be used as VIPs, also called ExternalIPs in K8s.
3. Running a **Cilium agent** on each L4LB node, which listens to Kubernetes resources (especially Services with externalIPs), and generates BPF rules for forwarding packets to backend pods.
4. **Glue VIP CIDR and Cilium agent in the kernel** with a dummy device on each L4LB node.
5. **Enable ECMP** on physical networks.

Traffic path when accessing the example service from the outside world:

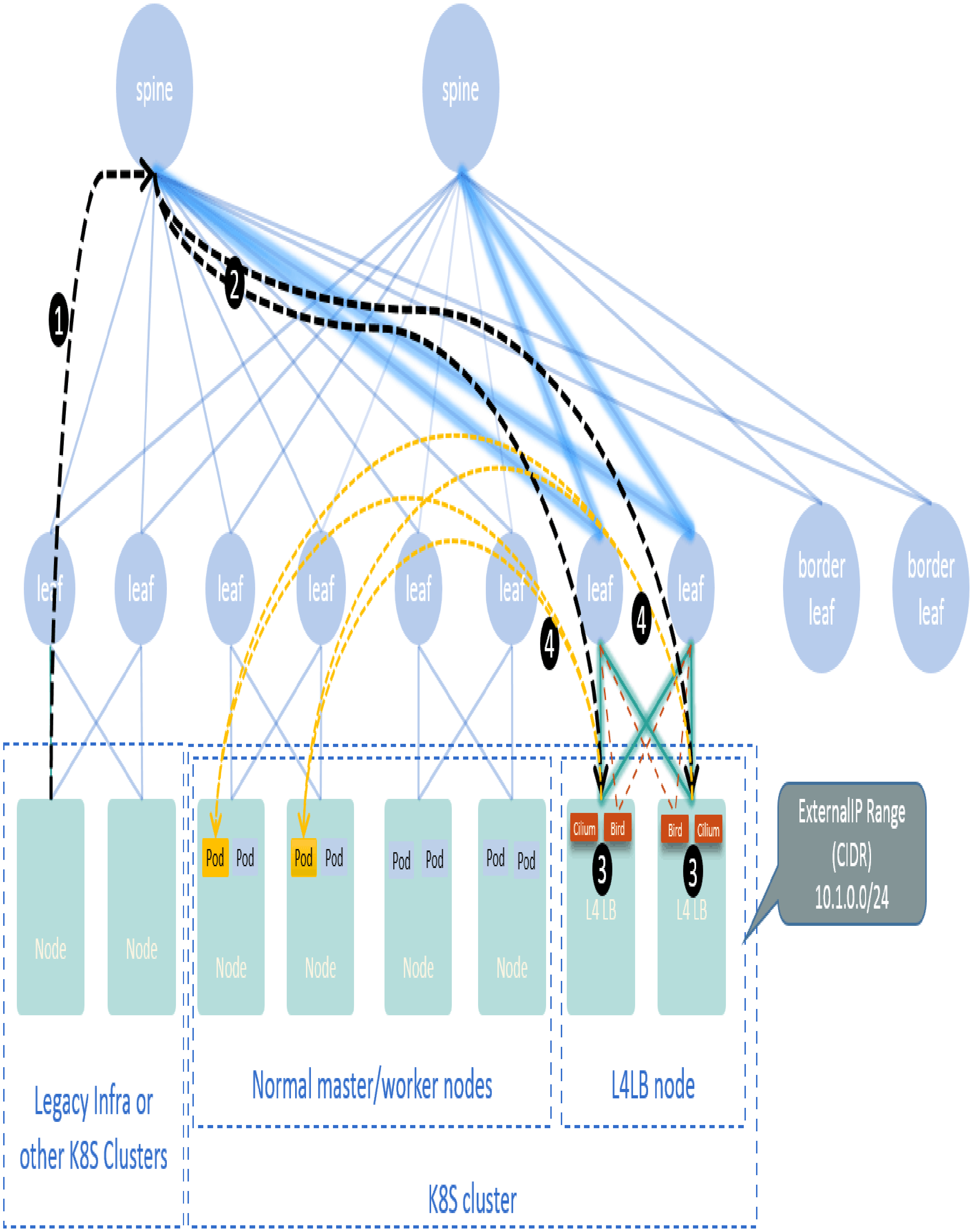

Fig. 3.2 Traffic path when accessing example service from outside of Kubernetes cluster

Where, the four numbered steps in the graph are:

1. Client packets -> physical network (spine)

2. Spine -> L4LB node, via ECMP
3. In-kernel routing (glue layer)
4. L4LB -> Backend pods, with Cilium load balancing

Details explained below. Let's see how it meets the 4 requirements listed in Section 2.3.

3.1. BIRD - BGP agent

L4LB nodes run `bird` as BGP agent.

When bird announces a CIDR (e.g. `10.1.0.0/24`) to the data center network, all subsequent packets with destination IPs within this CIDR (e.g. `10.1.0.2`) will be routed to this node.

If not familiar with BGP announcement, you could simply think of it as broadcasting something like `"CIDR 10.1.0.0/24 is at host XXX"` to the physical network. When routers receive this message, they will install it into their routing tables.

This solves the `1st requirement`: **persistent/invariant (L4) entrypoint**.

3.2. Cilium - Networking agent

`cilium-agent` on L4LB node will listen to Kubernetes apiserver, and generate BPF rules for Kubernetes ExternalIP services to forward traffic from VIPs (which are held by L4LB nodes) to backend pods.

This solves the `2nd and 3rd requirement`: **load balancing** and **timely react to backend changes**.

3.3. ECMP - Physical link HA

With ECMP (Equal Cost Multiple Path) enabled, we could have multiple L4LB nodes announcing the same CIDR, HW routers will load balancing those packets to our L4LB nodes (glowing links in Fig 3.1 & Fig 3.2).

Besides, there is another BGP optimization option called Bidirectional Forwarding Detection (BFD), enable it between routers and `bird` will accelerate routes convergence inside the entire network.

Thus, we solved the `4th requirement`: **high availability**.

3.4 Glue In-kernel Processing

Till now, packets with destination IPs within CIDR (VIPs) will arrive our L4LB nodes, and `cilium-agent` will generate forwarding rules for those VIPs.

But one part is still missing: **there are no rules in the kernel to redirect those packets into `cilium-agent`'s processing scope**. So without additional work, those packets will be dropped inside kernel, instead of be forwarded to backends by Cilium BPF rules, as shown in Fig 3.3:

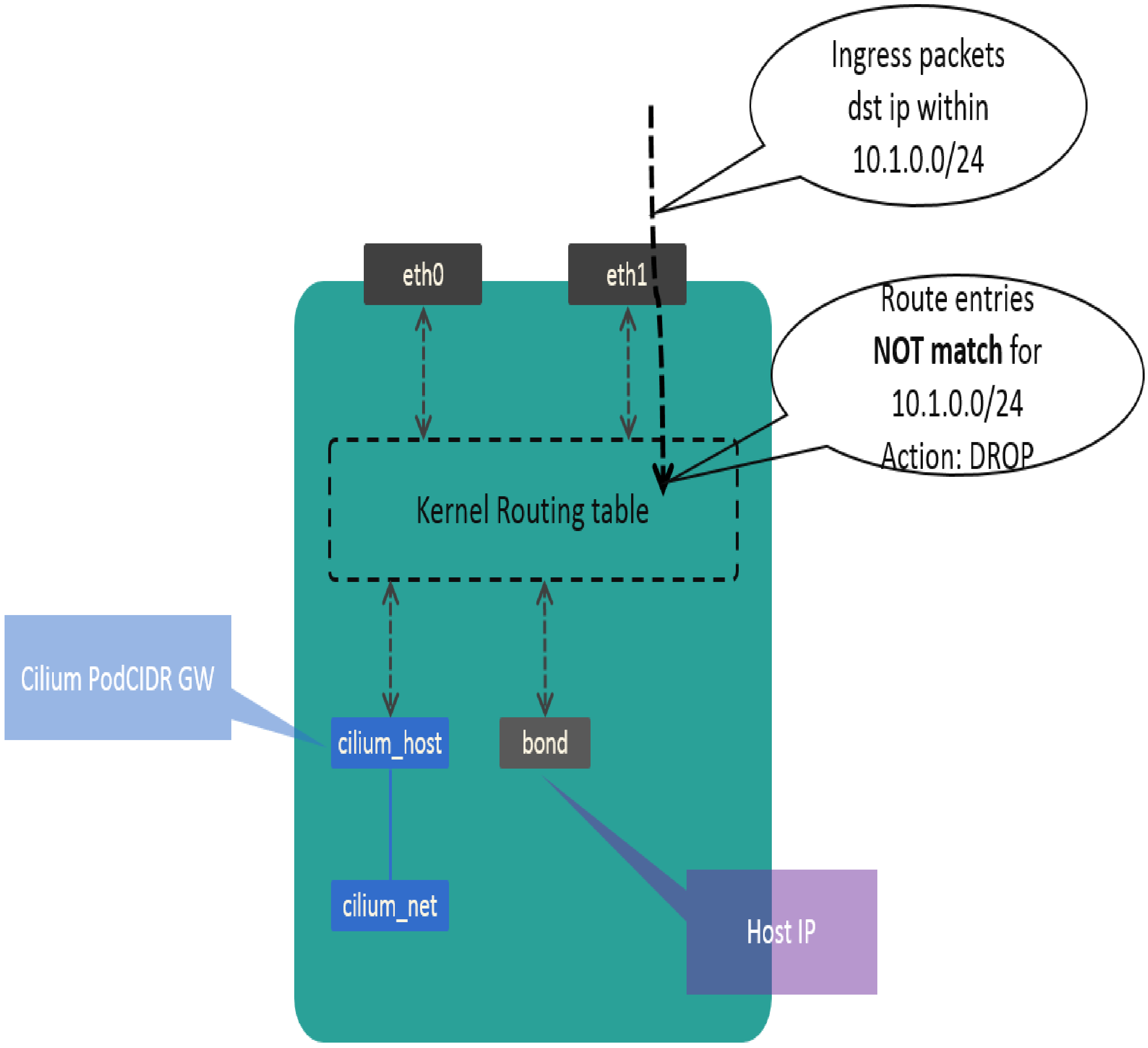
Ingress packets dst ip within 10.1.0.0/24
eth0
eth1
Route entries NOT match for 10.1.0.0/24 Action: DROP
Kernel Routing table
Cilium PodCIDR GW
cilium_host
bond
cilium_net
Host IP

Fig. 3.3 Packets dropped in kernel due to no routing rules

So we need some glue work to fill the gap. Our solution is **create a dummy device, which holds the first IP in our CIDR**.

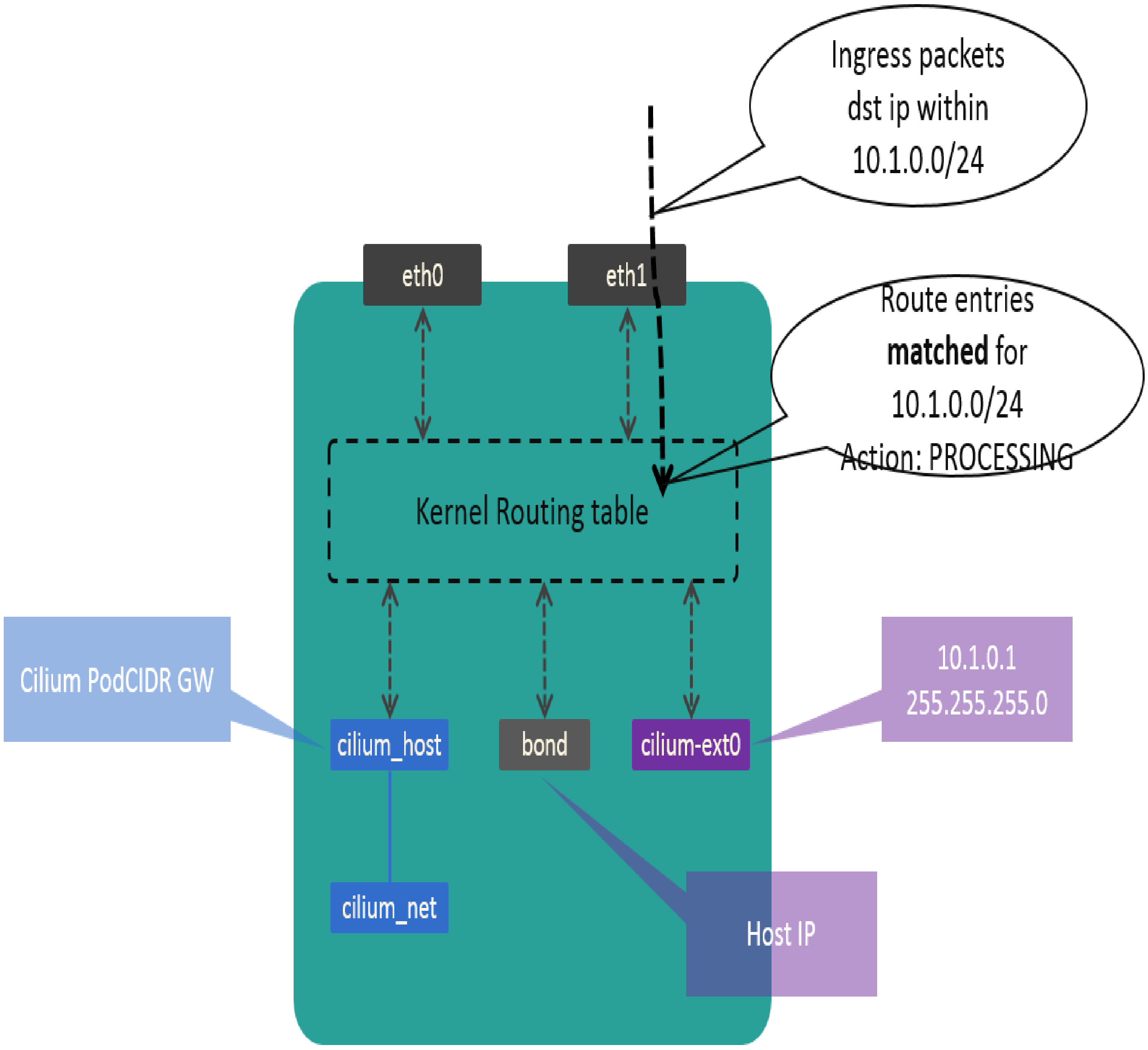

Fig. 3.4 Traffic process in kernel after add dummy device

4 Configurations

System info:

1. Centos: `7.2`
2. Kernel: `4.18+`
3. BIRD: `2.x`
4. Cilium: `1.7+`

Beside, this post assume the Kubernetes cluster uses **direct routing** for Pod networking. (While adjust Cilium configurations will also make this scheme work for non-direct-routing cases, that's beyond the scope of this post).

4.1 Create and configure dummy device

Install:

```
$ modprobe --ignore-install dummy
$ ip link set name cilium-ext0 dev dummy0
```

Configure it to survive host reboot:

```
$ cat ifcfg-cilium-ext0
DEVICE=cilium-ext0
IPADDR=10.1.0.1
NETMASK=255.255.255.0
ONBOOT=yes
TYPE=Ethernet
NM_CONTROLLED=no

$ cat ./modprobe.d/dummy.conf
install dummy /sbin/modprobe --ignore-install dummy; ip l

$ cat ./modules-load.d/dummy.conf
# Load dummy.ko at boot
dummy

$ cp ifcfg-cilium-ext0          /etc/sysconfig/network-sc
$ cp modprobe.d/dummy.conf      /etc/modprobe.d/dummy.con
$ cp modules-load.d/dummy.conf  /etc/modules-load.d/dummy
```

Enable changes:

```
$ systemctl restart network

$ ifconfig cilium-ext0
cilium-ext0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  r
              inet 10.1.0.1  netmask 255.255.255.0  broad
```

```
                ...

$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric I
...
10.1.0.1        0.0.0.0         255.255.255.0   U     0      (
```

4.2 bird

This involves BGP configurations for the `bird` software, as well as HW routers. Configurations may vary a lot according to the BGP schemes you choose, the latter is beyond the scope of this post. Refer to some get started docs, such as the one we wrote: Cilium documentation: Using BIRD to run BGP.

For `bird`, add following configurations to `/etc/bird.conf`,

```
protocol static {
        ipv4;                  # Again, IPv4 channel witi
        ...
        route 10.1.0.0/24 via "cilium-ext0";
}
```

Restart bird and verify changes are applied:

```
$ systemctl restart bird

$ birdc show route
BIRD 2.0.5 ready.
Table master4:
...
10.1.0.0/24          unicast [static1 2020-03-18] * (200)
     dev cilium-ext0
```

4.3 cilium-agent

Normal installation according official documents, as long as the agent could listen to kubernetes apiserver.

Two kinds of load balancing mechanisms:

4.3.1 NAT: to be specific, SNAT

Configurations:

- `kube-proxy-replacement=probe`
- `node-port-mode=snat` (default)

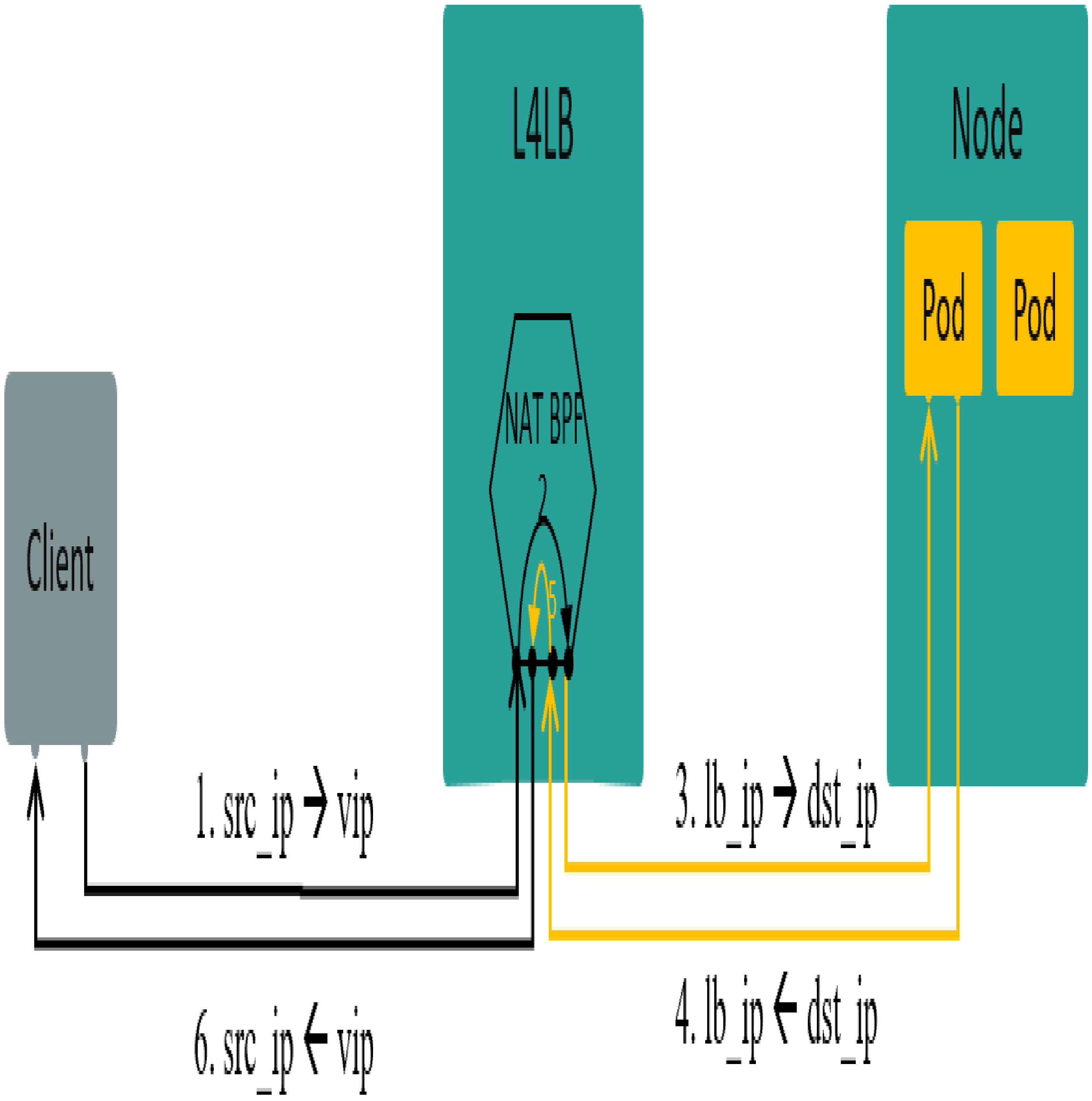

Fig. Traffic path in NAT mode

4.3.2 DSR: direct server return

Configurations:

- `kube-proxy-replacement=probe`
- `node-port-mode=dsr`

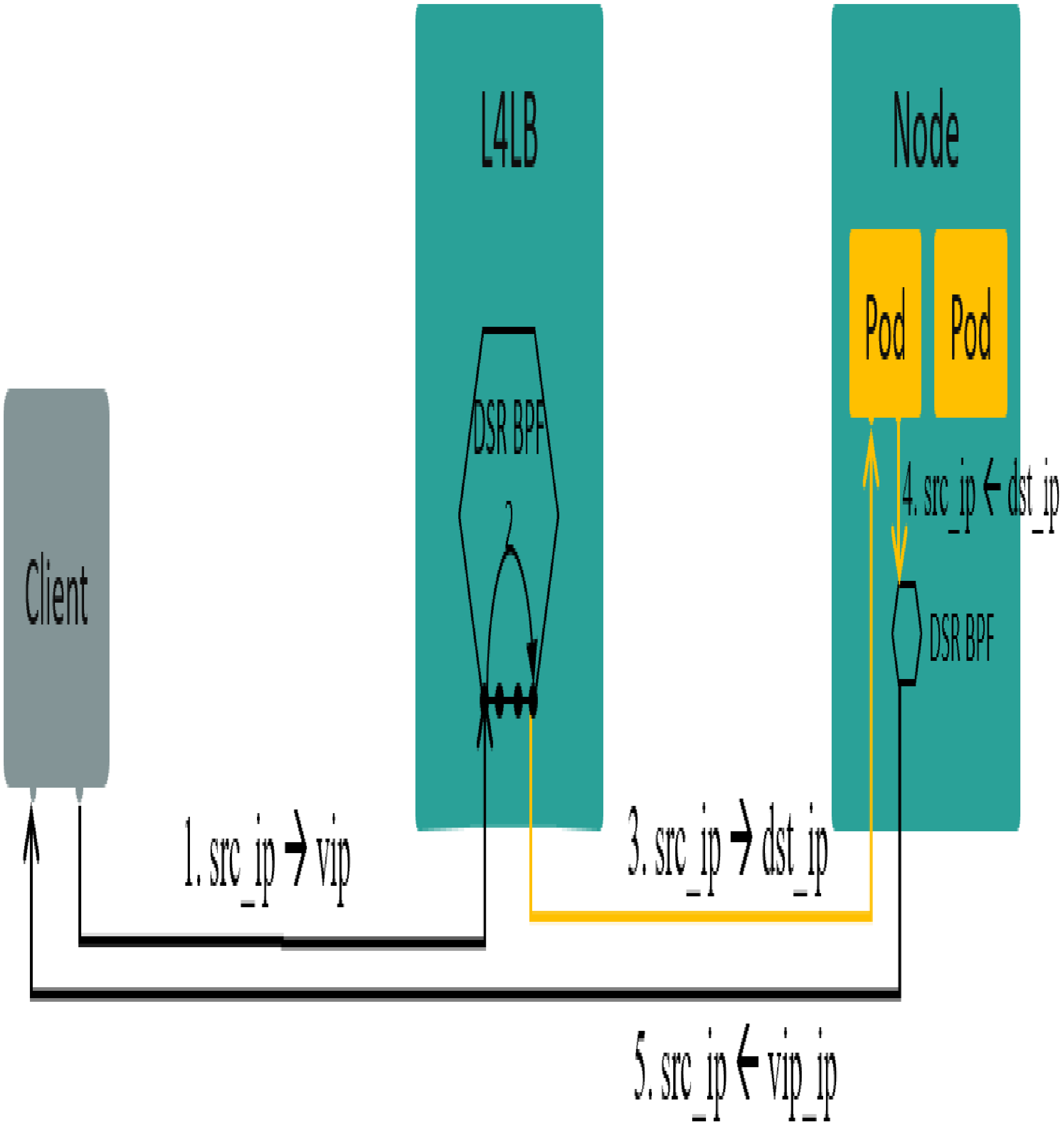

Fig. Traffic path in DSR mode

Note that if using DSR mode, **both L4LB and worker nodes** needs to be configured as **DSR**.

4.4 ECMP

ECMP needs to be configured on physical routers.

BFD should be configured both on physical routers and `bird`. Refer to Cilium documentation: Using BIRD to run BGP.

If everything is ok, you should see something like this on your routers:

```
ROUTER# show ip route 10.1.0.0
...
10.1.0.0/24, ubest/mbest: 2/0
    *via 10.4.1.7, [200/0], 13w6d, bgp-65418, internal, t
    *via 10.4.1.8, [200/0], 12w4d, bgp-65418, internal, t
```

4.5 Verification

For nothing but laziness, I will use my handy nginx service instead of deploying a real DNS service - but the effect is much the same, we will verify our scheme by accessing the nginx from layer 4.

4.5.1 On Master Node

On Kubernetes master, create a service with `externalIPs=10.1.0.2` :

```
$ cat cilium-smoke.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  externalIPs:
  - 10.1.0.2
  ports:
  - port: 80
    name: cilium-smoke
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cilium-smoke
spec:
  serviceName: "nginx"
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: library/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: cilium-smoke

$ kubectl create -f cilium-smoke.yaml
```

Check our service's information:

```
$ kubectl get svc
$ k get svc
NAME    TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
...
nginx   LoadBalancer   10.x.x.x     10.1.0.2      80/TCP
```

As can be seen, it is `LoadBalancer` type, and has an external IP `10.1.0.2` .

4.5.2 On L4LB Node

Execute following command inside cilium-agent pod:

```
$ cilium service list
...

62   10.1.0.2:53    ExternalIPs    1 => 192.168.1.234:53
                                   2 => 192.168.2.172:53
```

> TODO: more BPF info on L4LB node.
>
> `iproute2` on my current L4LB node is too old, which prohibits my further investigation. You could refer to my previous post Cilium Network Topology and Traffic Path on AWS if intersted.

4.5.3 On test node

On a node which is outside of Kubernetes cluster, test our externalIP service with VIP+Port:

```
$ telnet 10.1.0.2 80
Trying 10.1.0.2...
Connected to 10.1.0.2.
Escape character is '^]'.
```

Successful!

5. More Thoughts

5.1 Pros & Cons

5.1.1 Pros

- Simple, straight forward, ease of understanding
- Ease of management of ExternalIPs: manage CIDR instead of distinct IPs, less BGP announcements
- Ease of security rule managements for BGP filtering

5.1.2 Cons

- ECMP hard limits: 16 nodes.
- All traffic goes through L4LB nodes, make them the potential bottleneck (e.g. BW, CPU processing).
- No failover in case of L4LB node downs.

  Currently, if one L4LB node downs, traffic will be rehashed to other L4LB nodes by HW switches via ECMP. However, in the current implementation, all the these traffic will be disruppted as other L4LB nodes could not correctly handle this (no failover). Failover could be achieved with either of:

  i. Session replication: sync connection/session info among all L4LB nodes, so if one node down, the connection could be handled by other nodes (almost) transparently. LVS supports this features as an experiment.
  ii. Consistent hashing: another way to achieve failover, and **recent L4LB solutions favor this fashion**, e.g. Facebook Katran, Google Maglev, Github GLB. **Cilium currently (`v1.8.2`) lacks this feature**.

5.2 Decoupling of BGP agent and networking agent

It's important to understand that, in this L4LB scheme, there are no couplings between BGP agent and host network agent, that is,

- You could transparently replace `bird` with another BGP agent, e.g. `quagga` (but you need to concern whether they support the features you would like, e.g. ECMP, BFD).
- You could also transparently replace `cilium-agent` with other networking agents for Kubernetes worker nodes, e.g. `kube-proxy` .

5.3 The Hidden CLOS Architecture

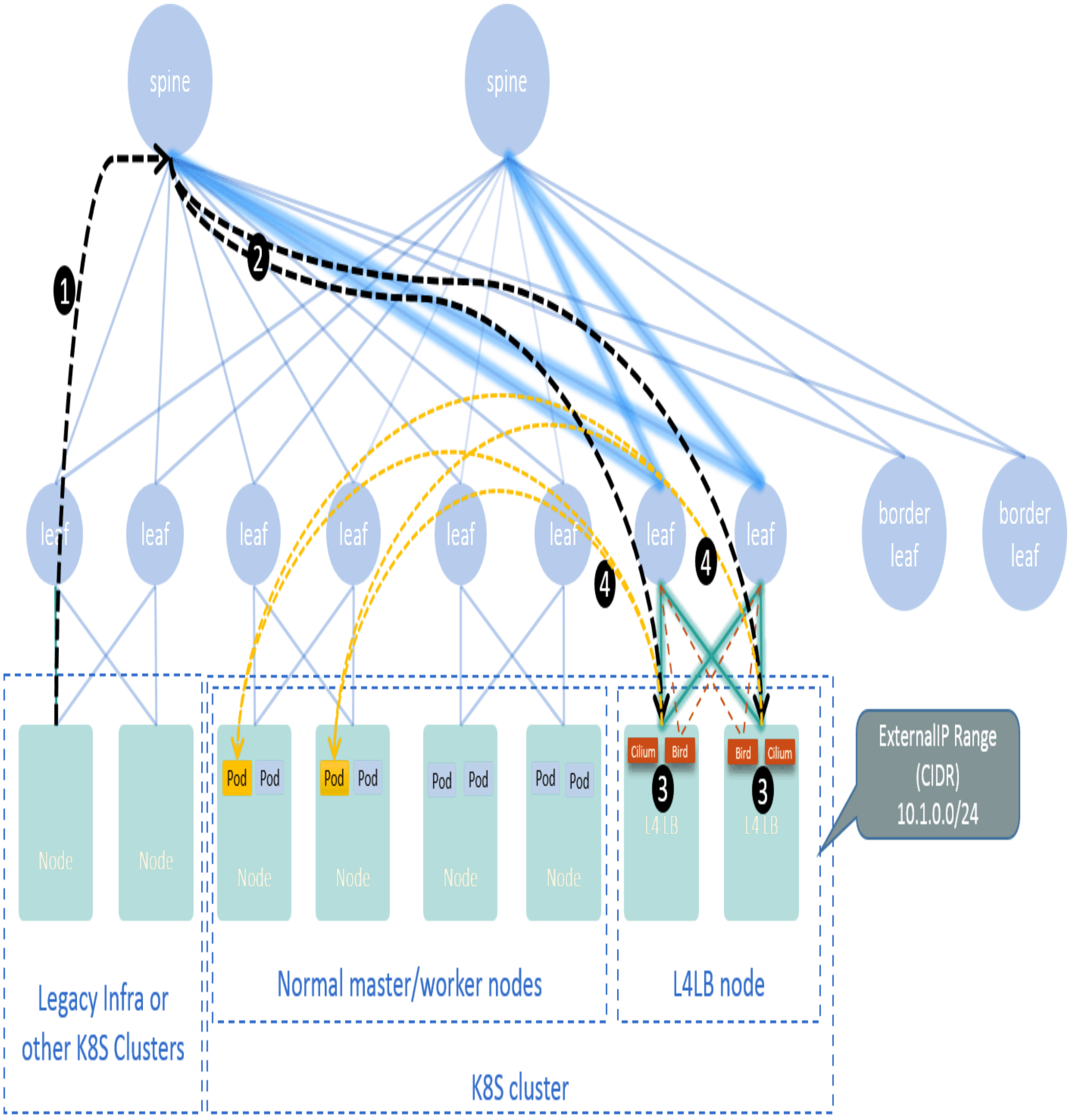

Fig. 3.2 Traffic path when accessing example service from outside of Kubernetes cluster

If think of each TOR pair and corresponding L4LB nodes as an integral entity, Fig 3.2 could be re-depicted as Fig 5.1:

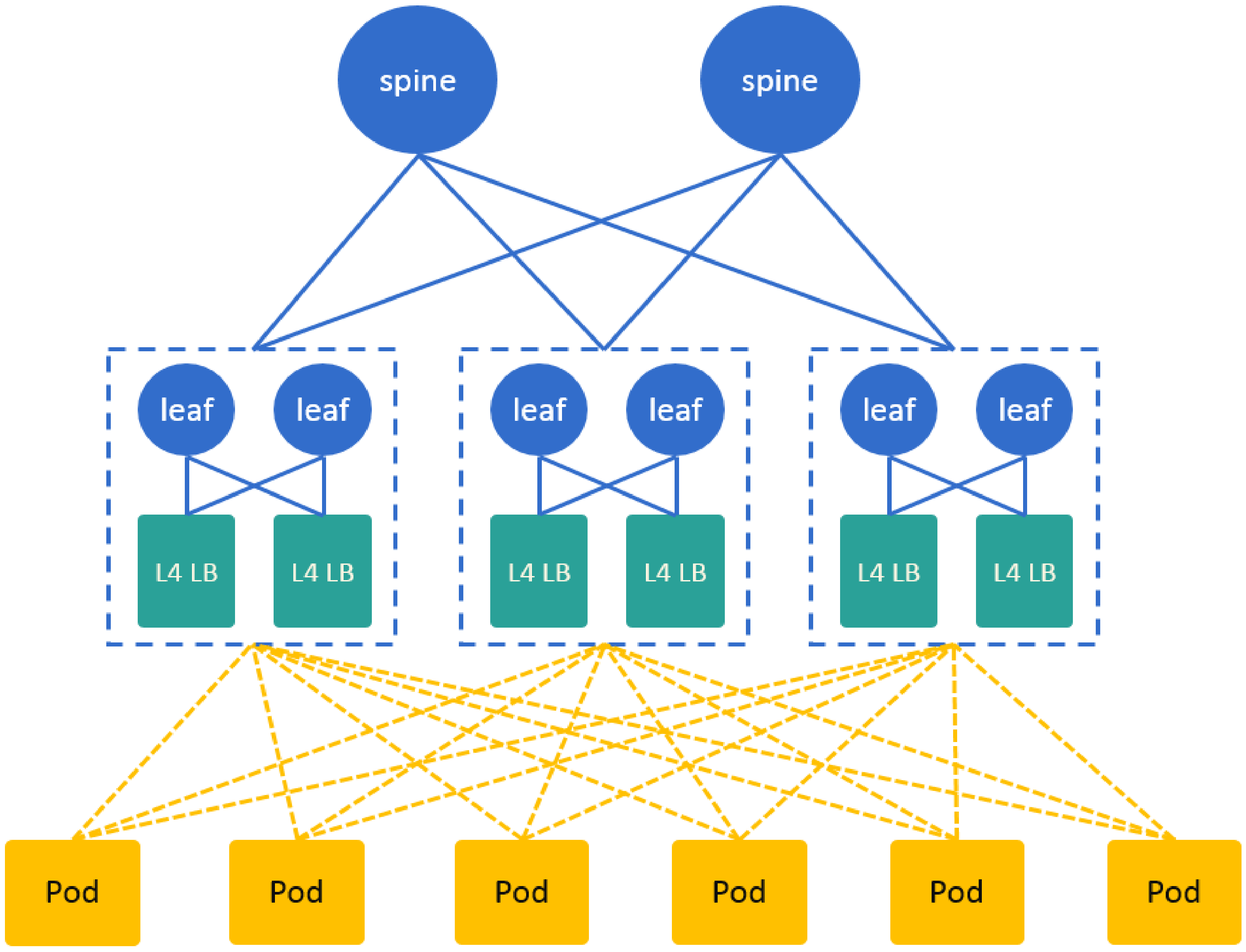

Fig. 5.1 The Hidden CLOS architecture

which is a CLOS network.

Interesting!

5.4 Ingress vs LoadBalancer

Ingress provides persistent L7 entrypoints for accessing services inside Kubernetes cluster from outside world. For example,

- App A: `https://<ingress addr>/app-a/`
- App B: `https://<ingress addr>/app-b/`
- App C: `https://<ingress addr>/app-c/`

Have you wondered how to design an Ingress solution?

If your Ingress is deployed inside Kubernetes cluster, then **itself must have a persistent L4 entrypoint**, namely the `<ingress addr>` in the above example.

Combining the L4LB solution in this post and something like Istio Gateway, you will get a workable Ingress.

6. Similar Works

6.1 MetalLB

MetalLB is similar with this in that it:

1. Announce VIP via BGP
2. Forward/load-balancing to backends with node agent

Difference from the one in this post:

1. BGP agent announces distinct IPs instead of CIDR
2. No dedicated L4LB nodes
3. Forward/load-balancing to backends via kube-proxy (recently supported Cilium)

Problem may faced: more BGP announcements, more routing entries, more difficult for filtering BGP announcements.

6.2 Katran

Katran is a general purpose L4LB from Facebook, also based on BGP + BPF.

6.3 Kube-proxy + BGP

As mentioned in 5.2, replace Cilium with kube-proxy could also achieve this goal. Differences including:

1. Forwarding rules on L4LB node will be based on iptables or LVS instead of BPF
2. Performance degrades (maybe)

7 Summary

This post analyzed the technical requirements of L4LB for Kubernetes clusters, and realized a simple one based on Cilium+BGP+ECMP.