KubeCon | CloudNativeCon

North America 2023

Modern Load Balancing
Improving Application's Resource Availability and Performance

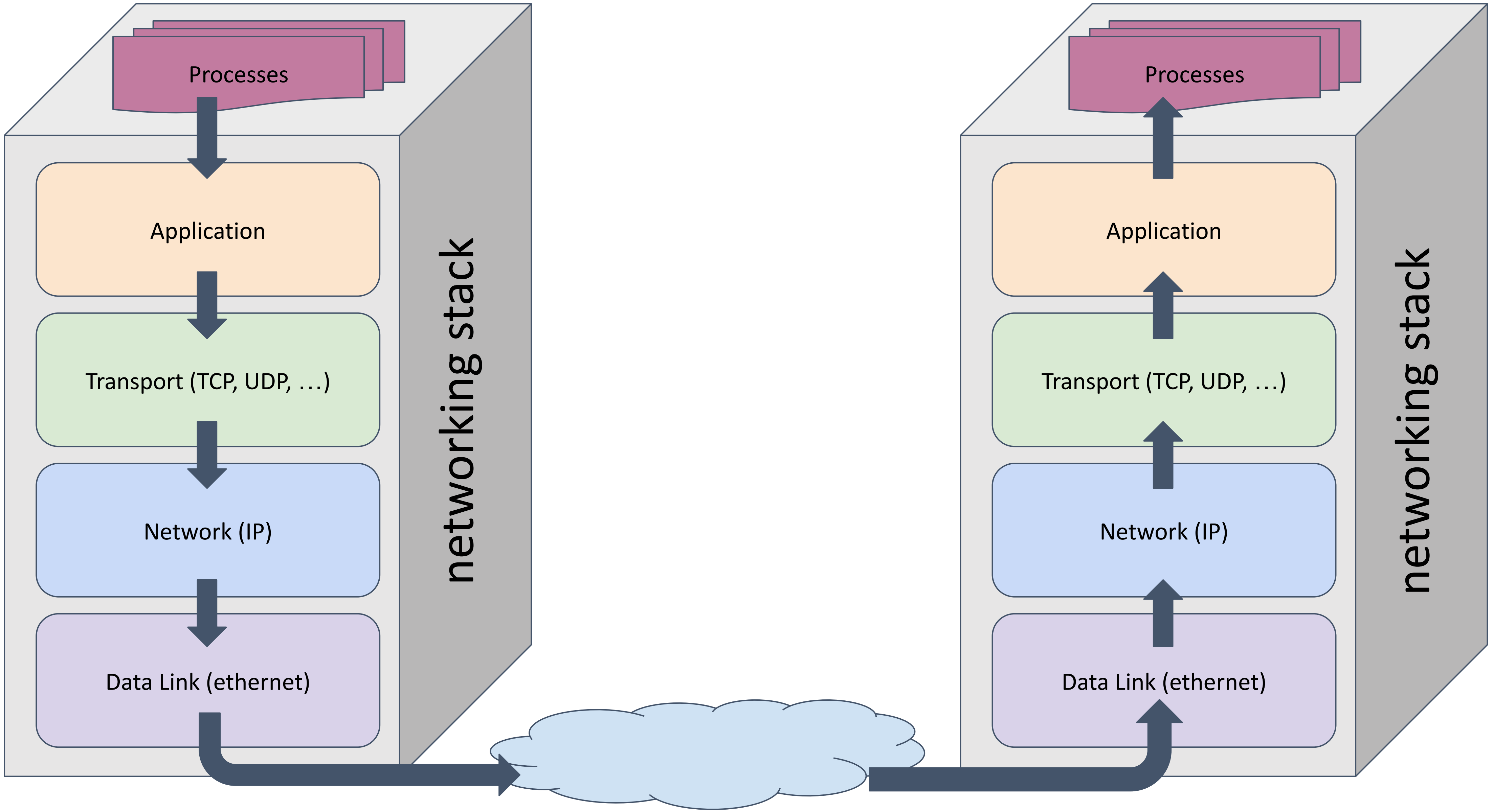
Antonio Ojea, Google
Gerrit DeWitt, Google

KubeCon | CloudNativeCon
North America 2023

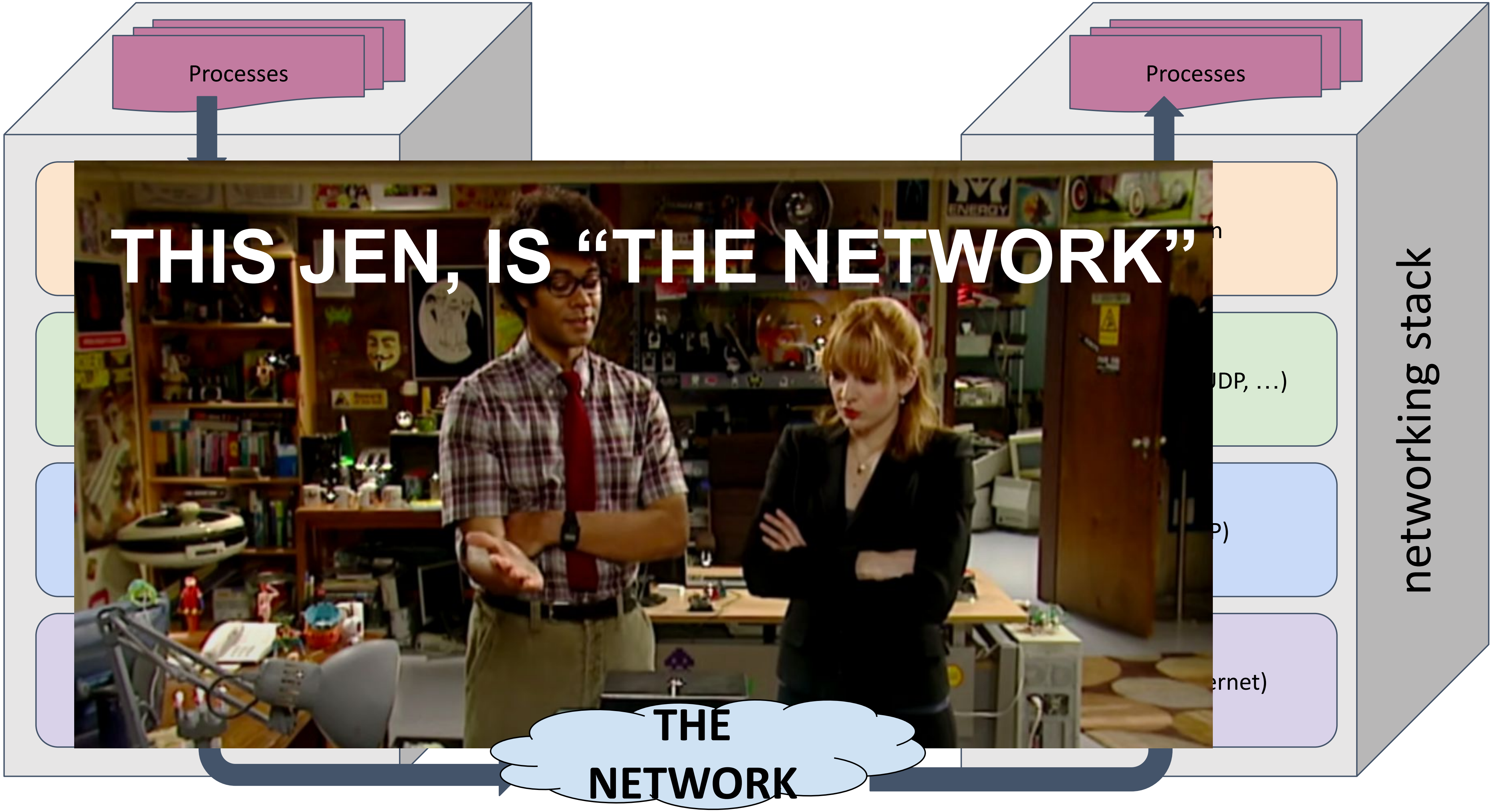
Why do I need a Load Balancer?

- High availability
- Performance
- Service discovery

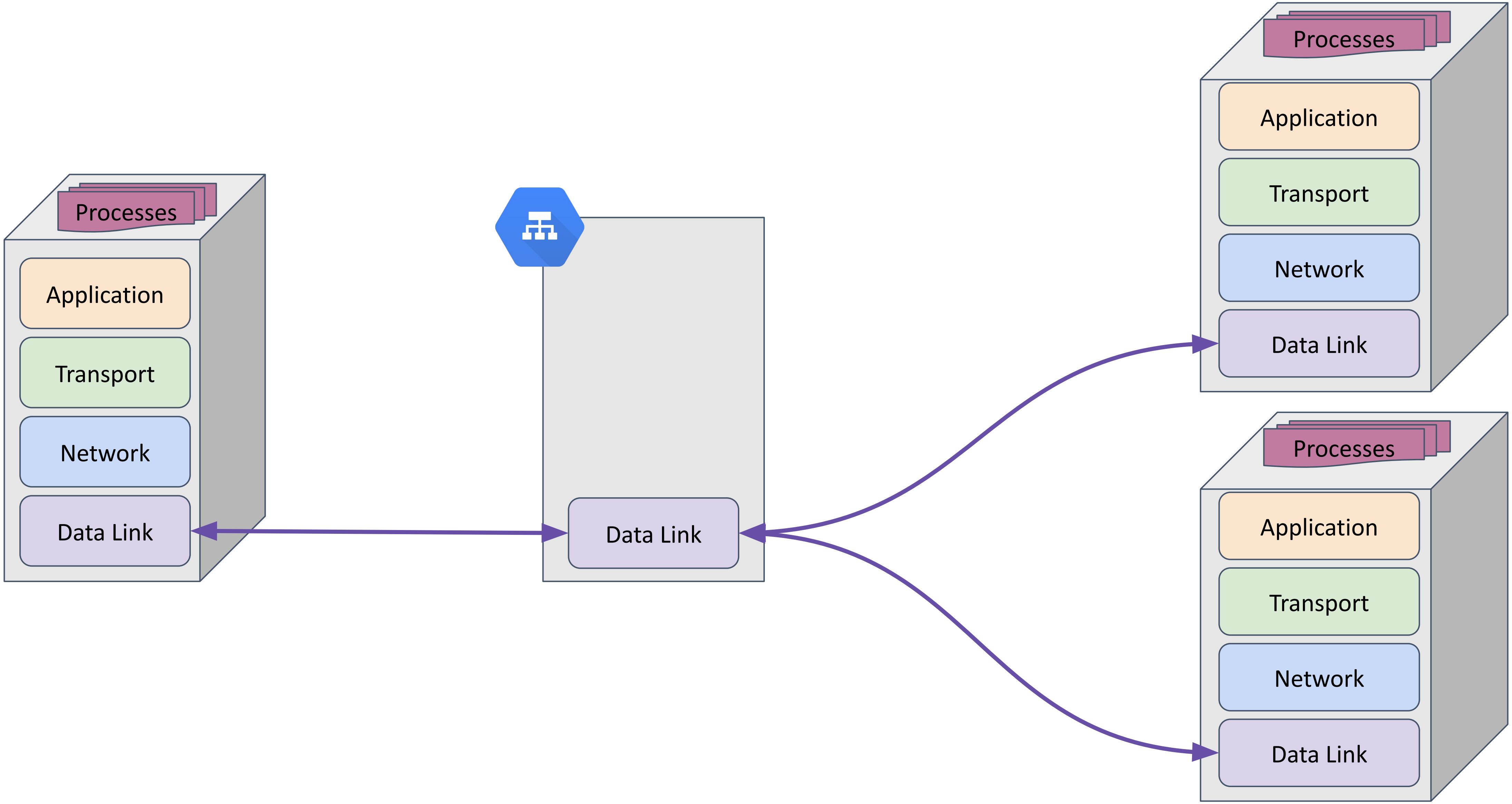
The networking stack



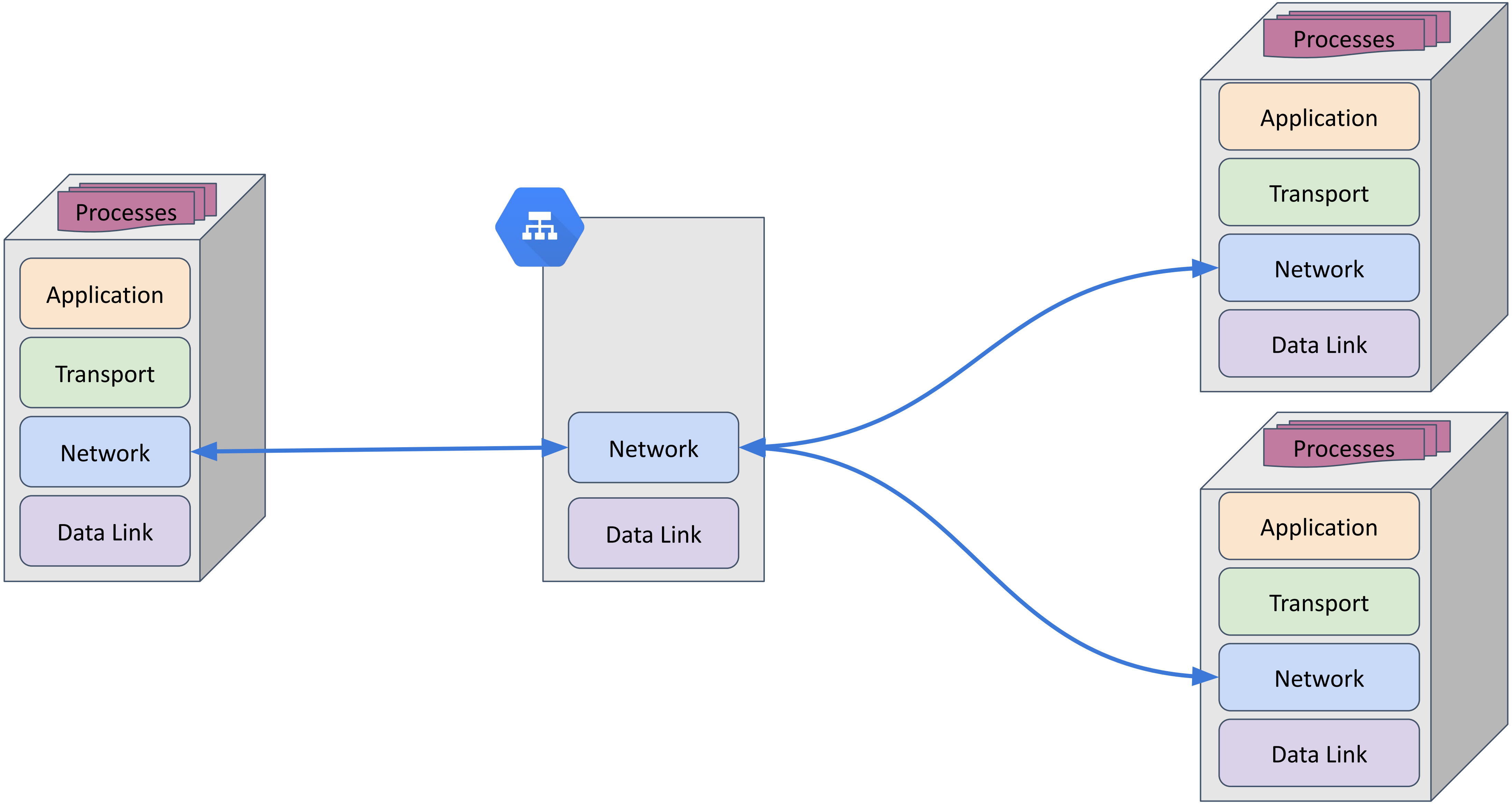
The networking stack



How Load Balancers works: L2

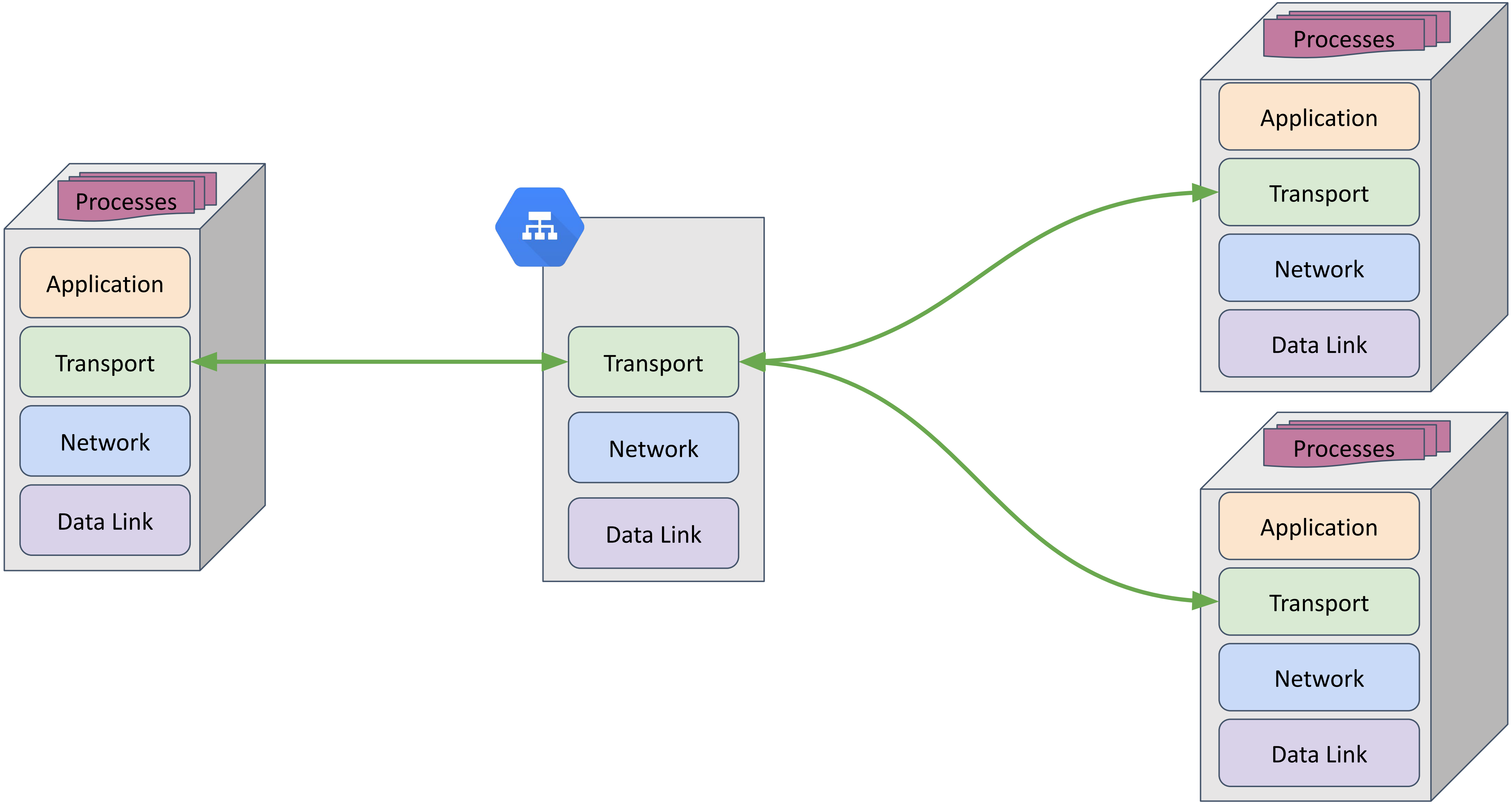


How Load Balancers works: L3



How Load Balancers works: L4

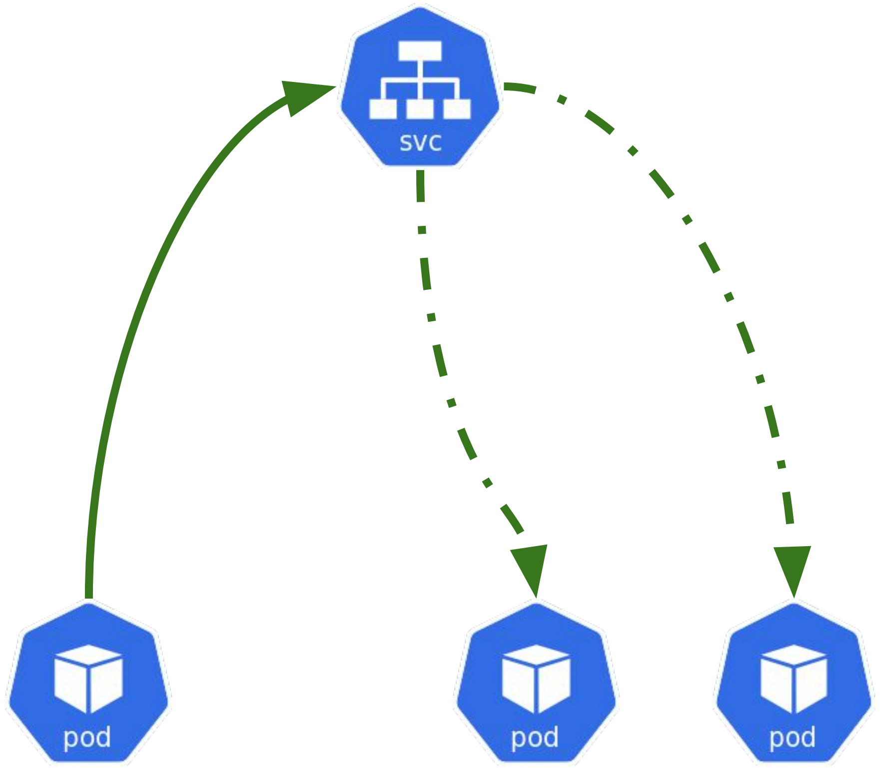
KubeCon | CloudNativeCon
North America 2023



Kubernetes Services: ClusterIP


```yaml
apiVersion: v1
kind: Service
metadata:
  name: service
spec:
  clusterIP: 10.96.0.1
  clusterIPs:
  - 10.96.0.1
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: 6443
  type: ClusterIP
```

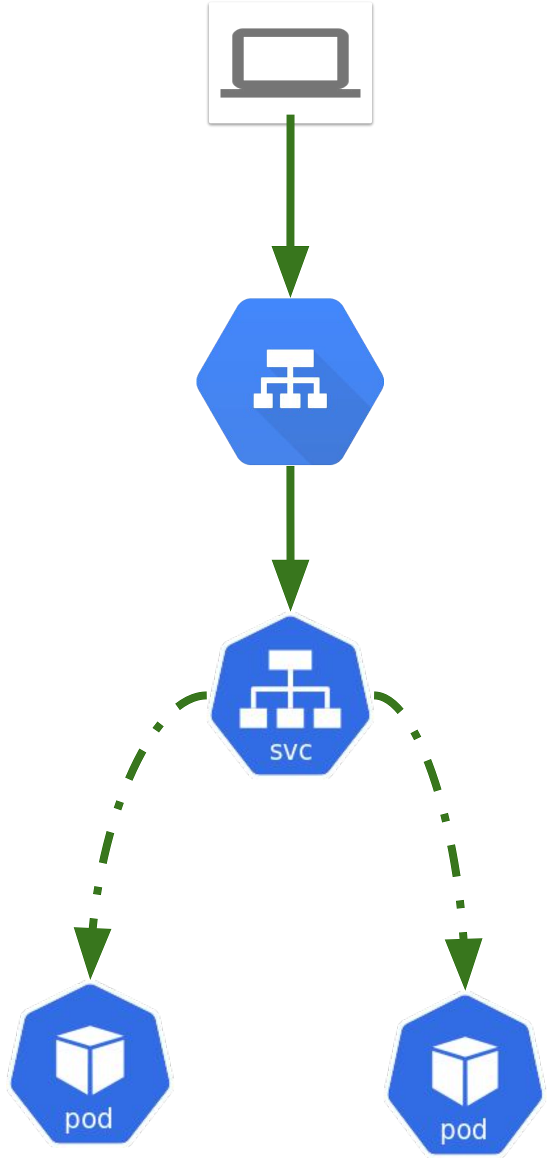
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server-deployment
  labels:
    app: MyApp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: MyApp
  template:
    metadata:
      labels:
        app: MyApp
    spec:
      terminationGracePeriodSeconds: 30
      containers:
      - name: agnhost
        image: k8s.gcr.io/e2e-test-images/agnhost:2.39
        args:
          - netexec
          - --http-port=80
          - --delay-shutdown=30
```



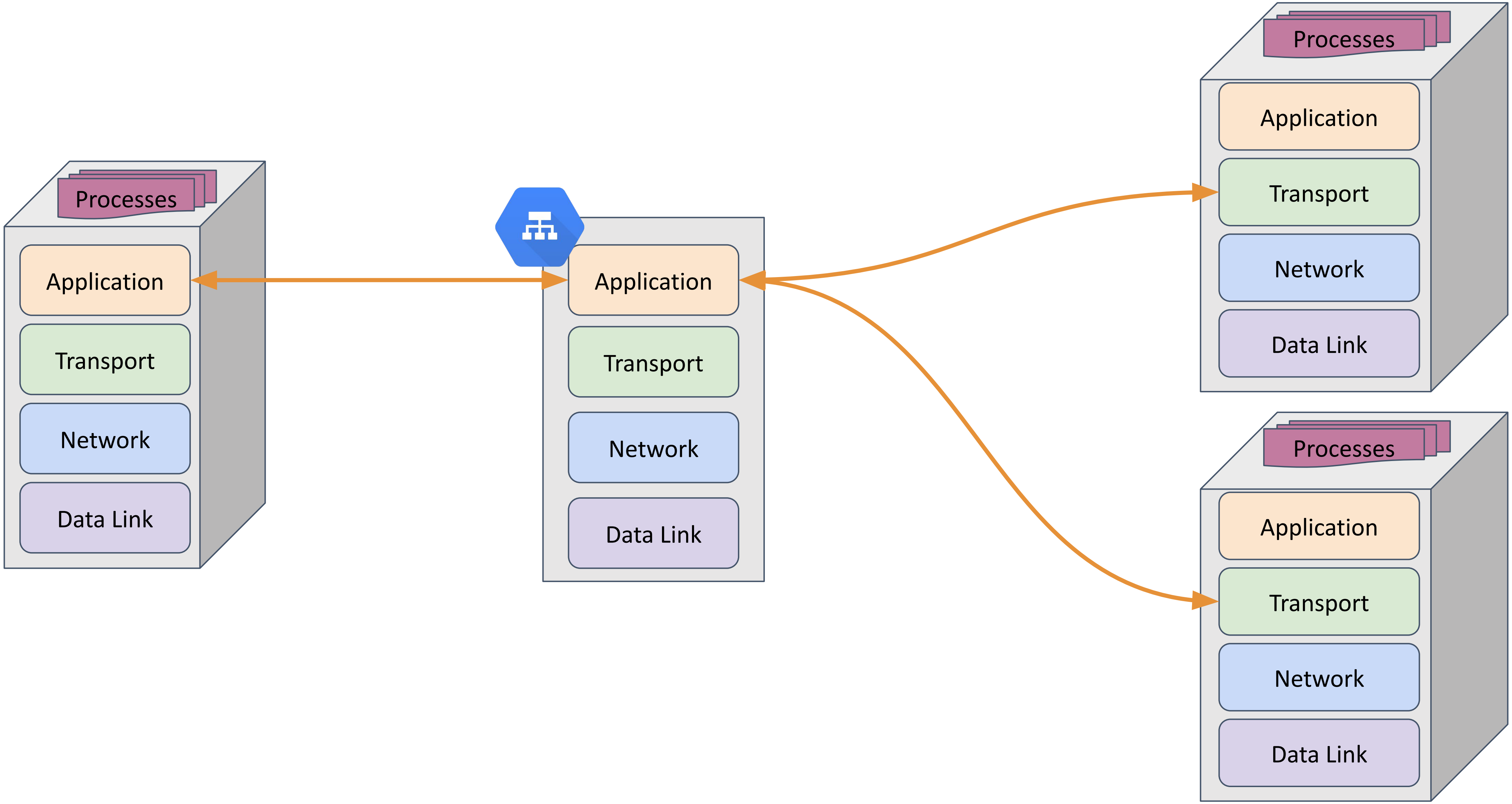
Kubernetes Services: LoadBalancer


```yaml
apiVersion: v1
kind: Service
metadata:
  name: service
spec:
  clusterIP: 10.96.0.1
  clusterIPs:
  - 10.96.0.1
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: 6443
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
  -   ip: 202.34.23.12
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server-deployment
  labels:
    app: MyApp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: MyApp
  template:
    metadata:
      labels:
        app: MyApp
    spec:
      terminationGracePeriodSeconds: 30
      containers:
      - name: agnhost
        image: k8s.gcr.io/e2e-test-images/agnhost:2.39
        args:
          - netexec
          - --http-port=80
          - --delay-shutdown=30
```

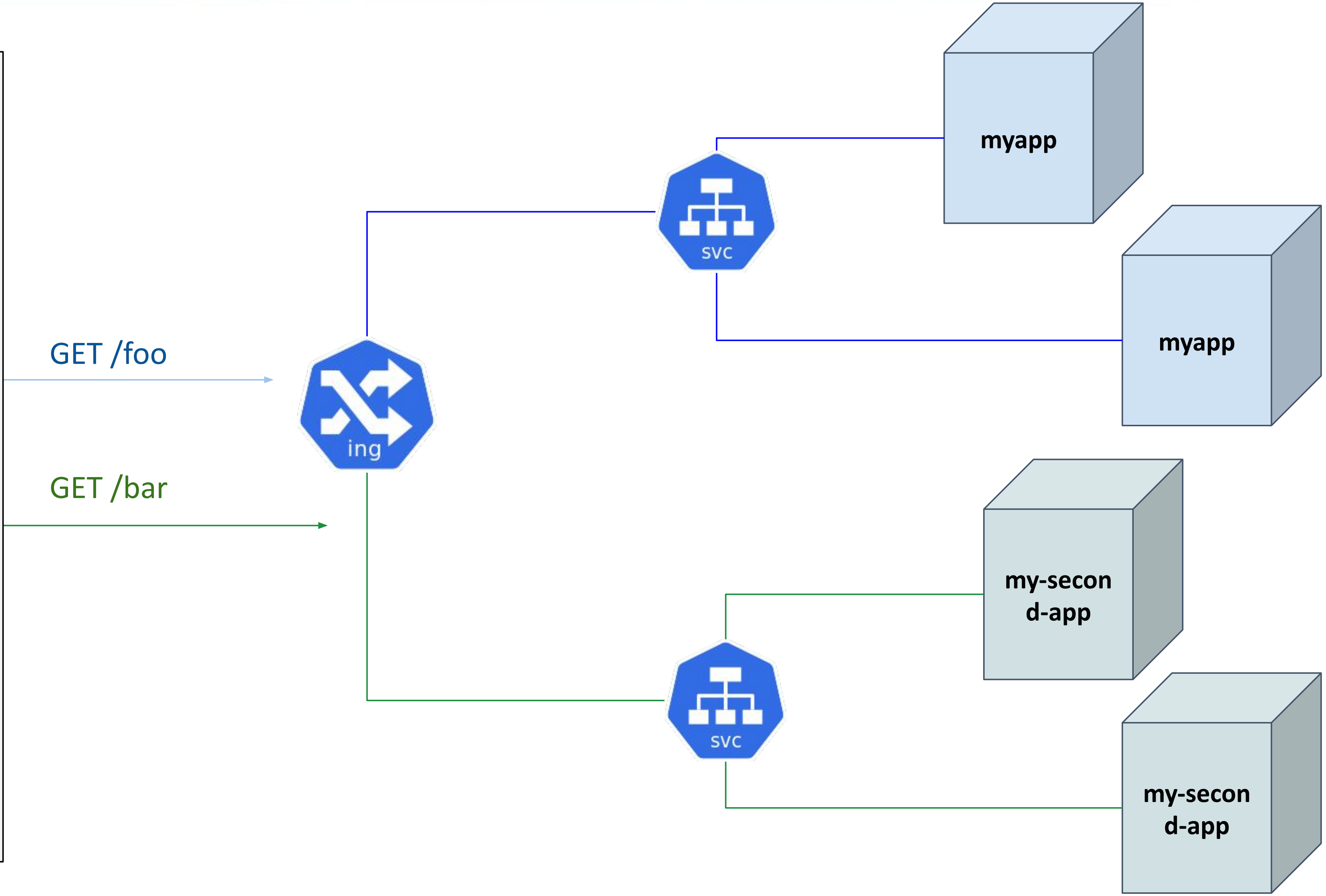


How Load Balancers works: L7

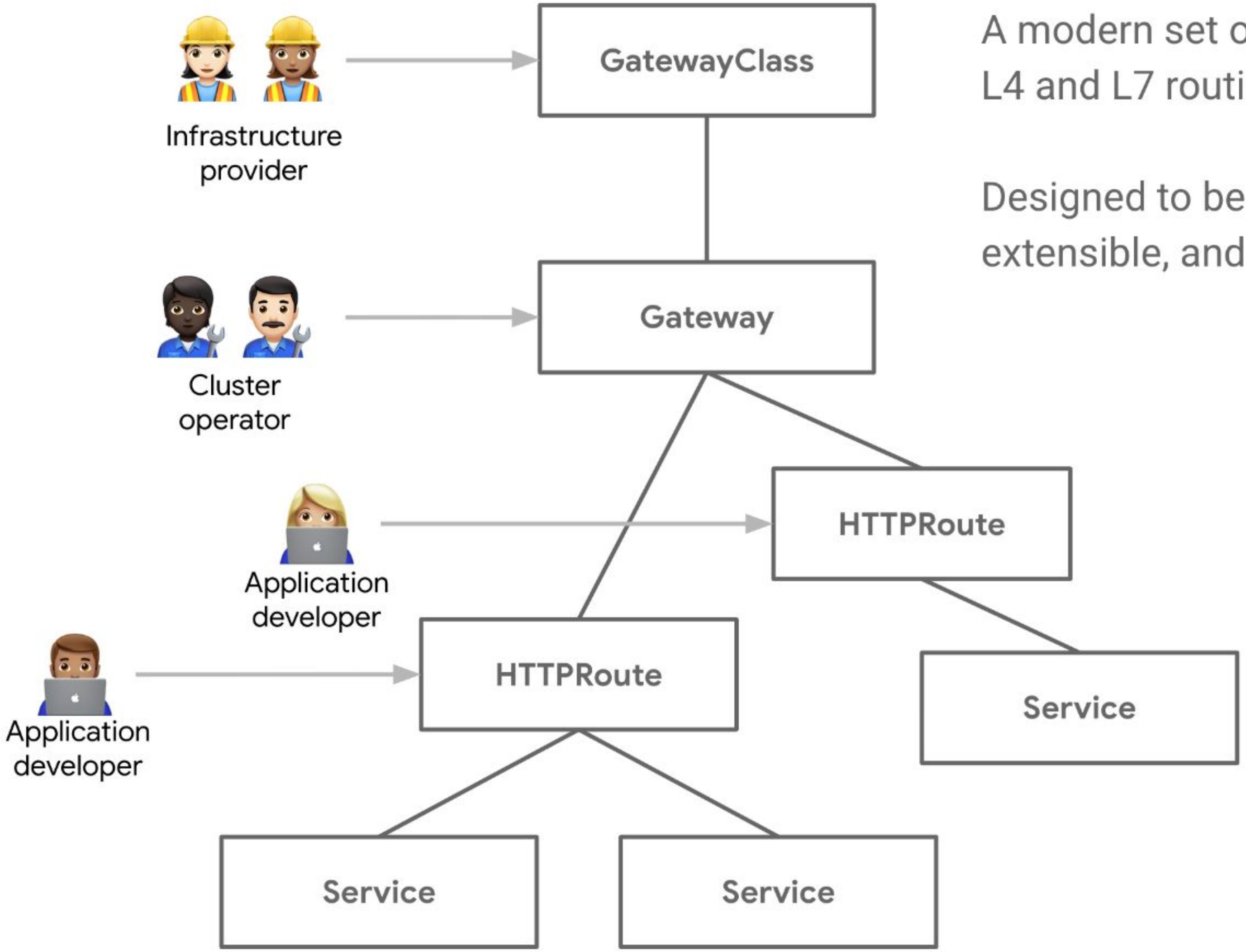


Kubernetes Ingress == L7 Load Balancer


```yaml
apiVersion: v1
kind: Ingress
metadata:
  name: minimal-ingress
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: myapp
            port:
              number: 80
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: my-second-app
            port:
              number: 80
```



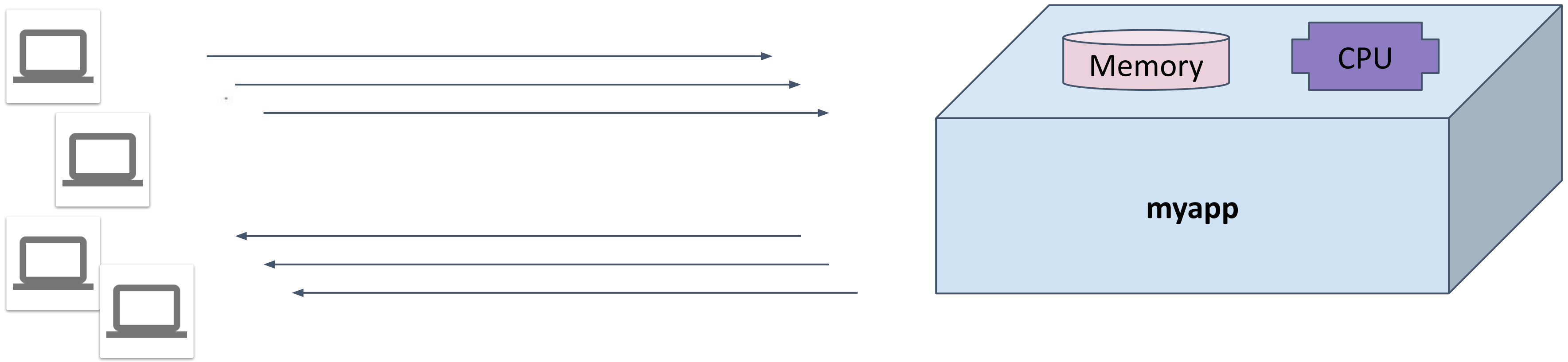
Gateway API aims to define the space



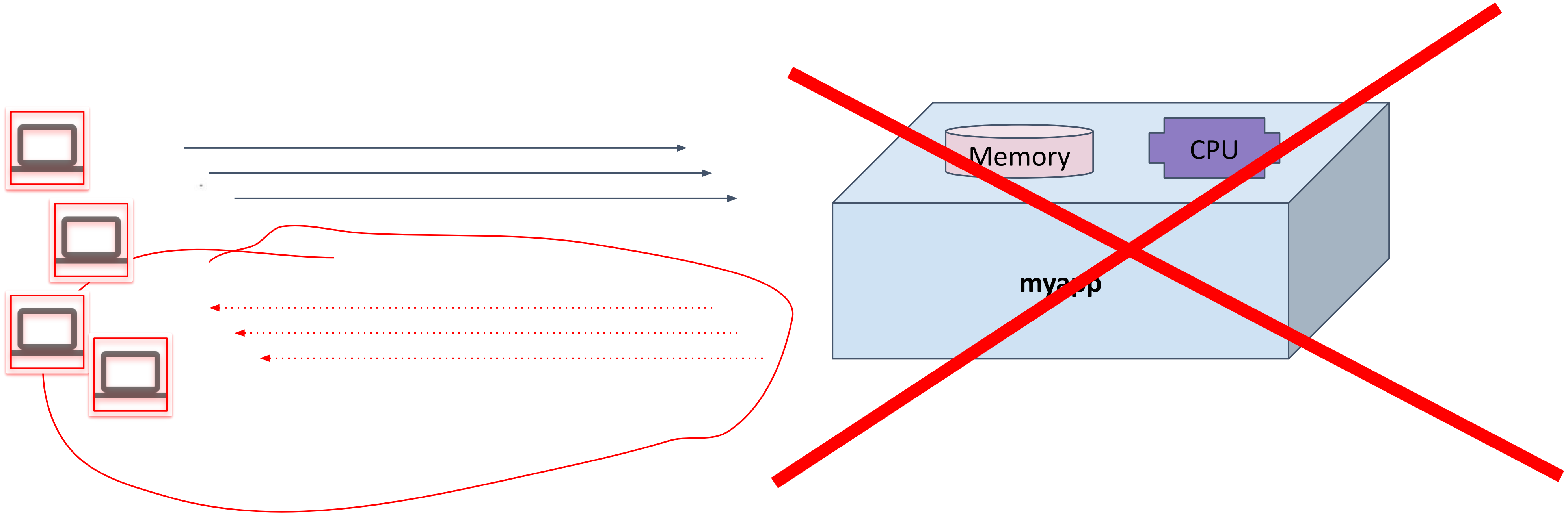
A modern set of APIs for deploying L4 and L7 routing in Kubernetes

Designed to be generic, expressive, extensible, and role-oriented

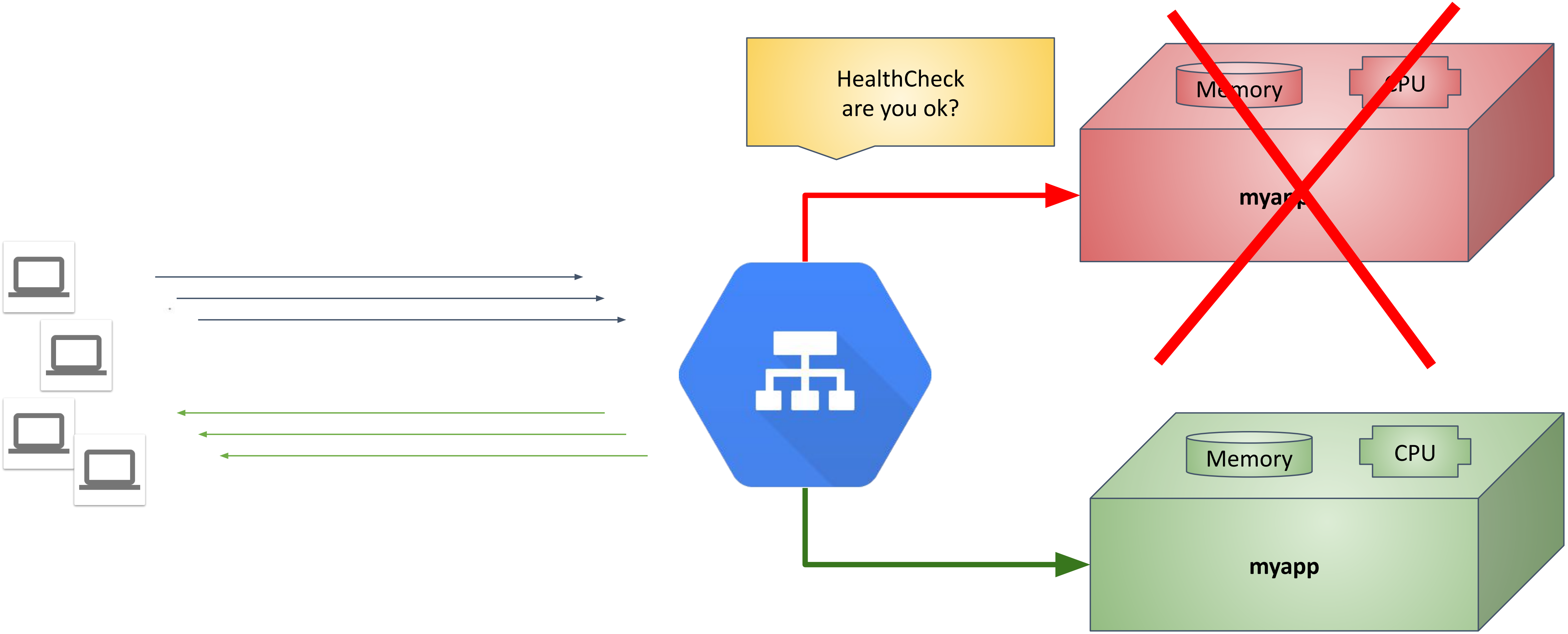
Solving the High Availability Problem



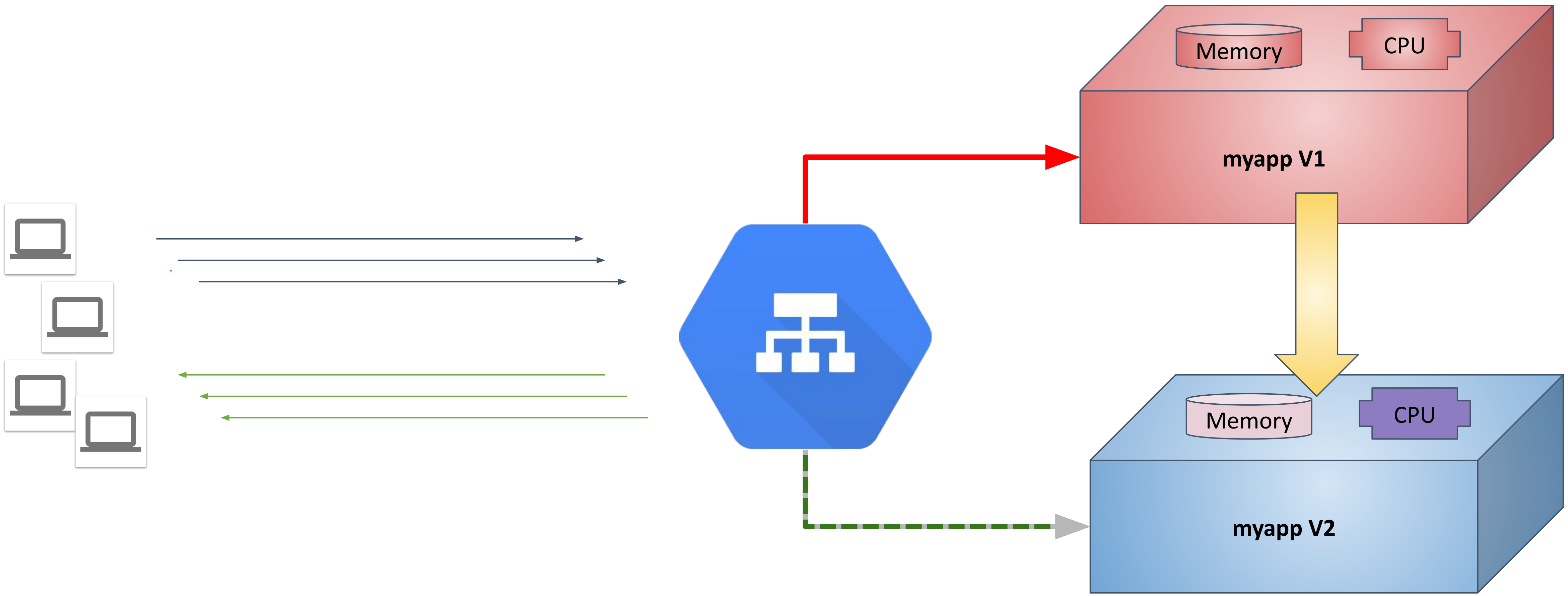
Solving the High Availability Problem



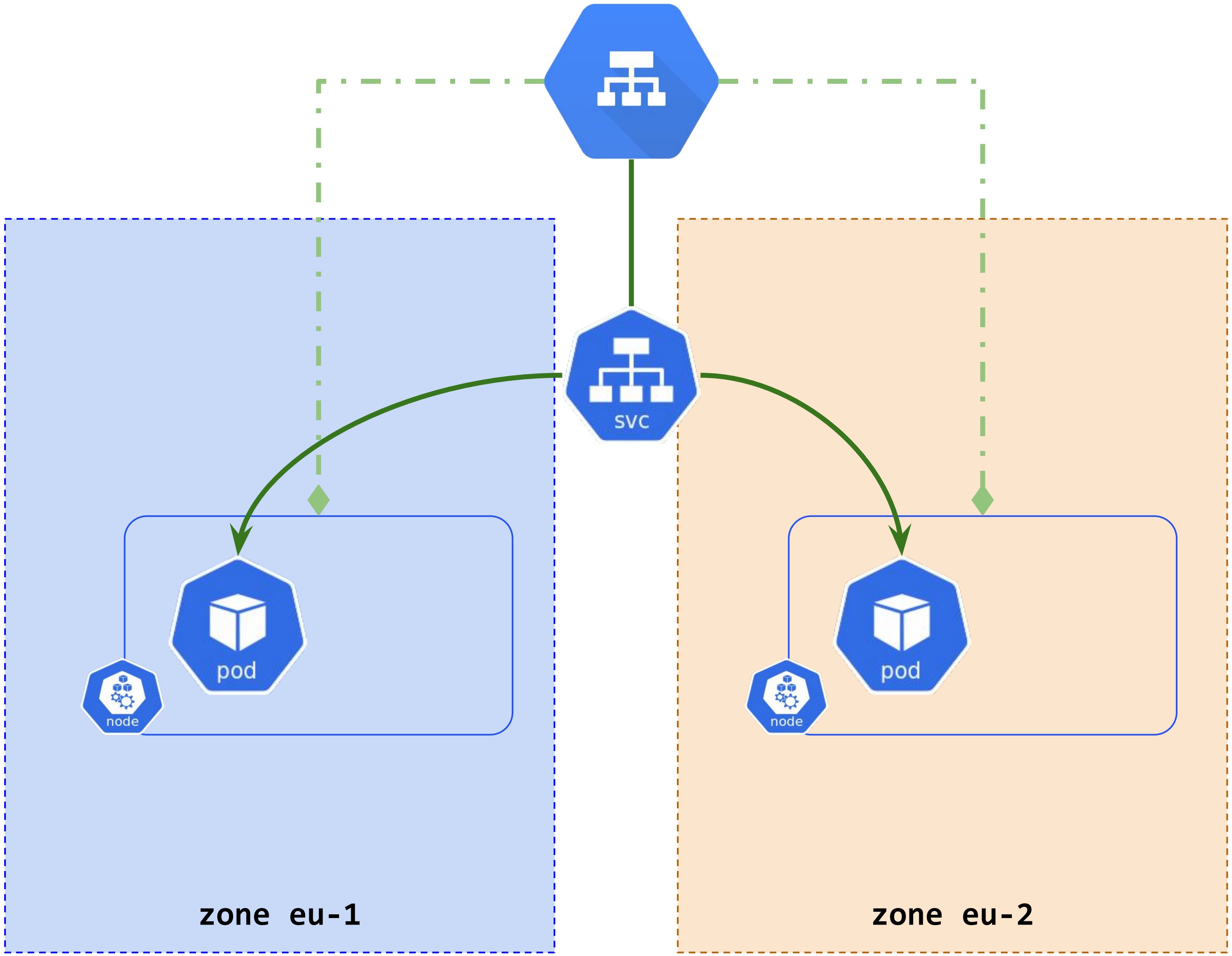
Solving the High Availability Problem



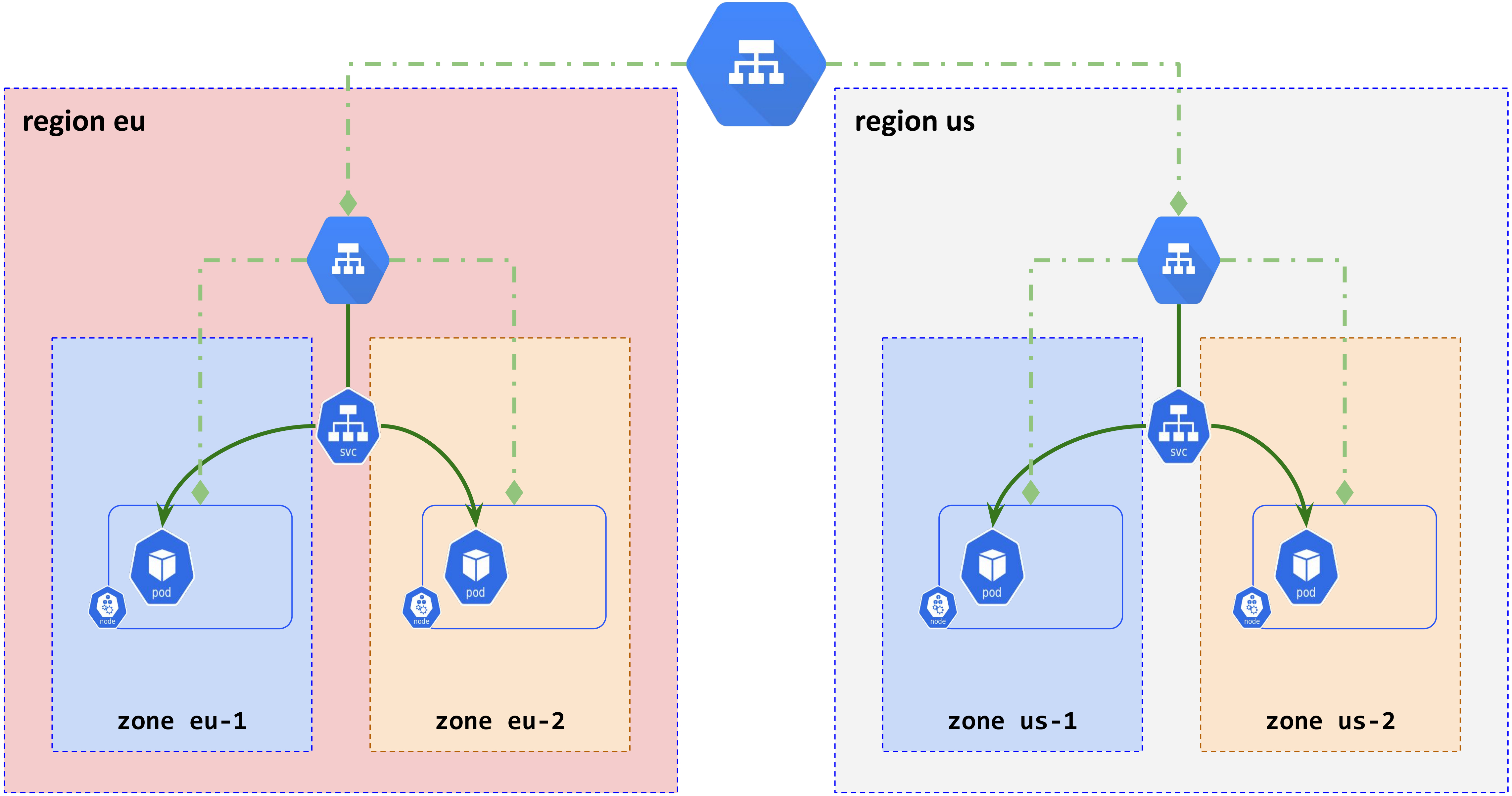
Rolling updates



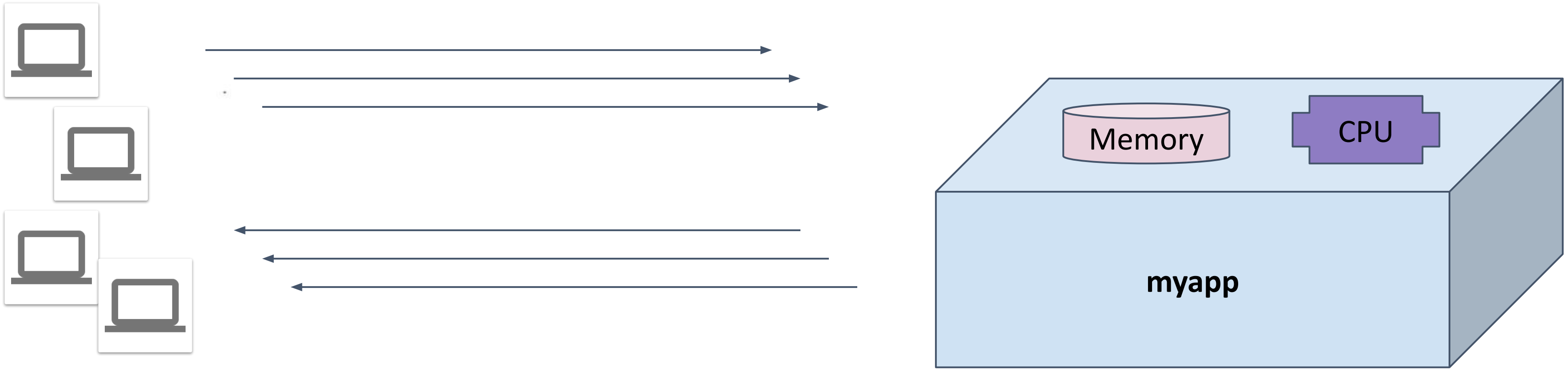
Regional High Availability



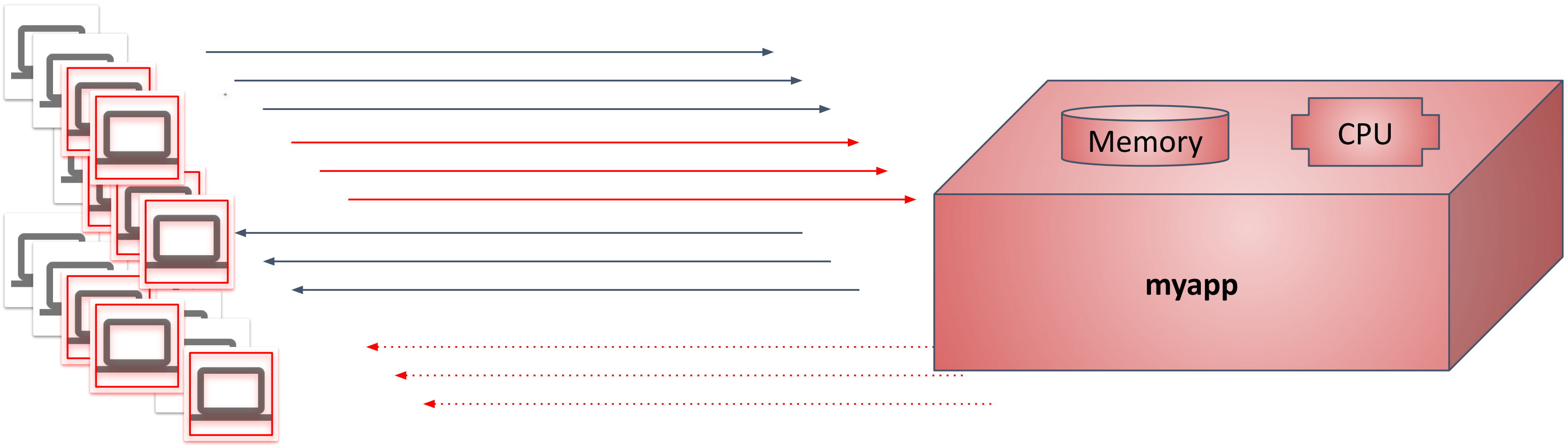
Global High Availability



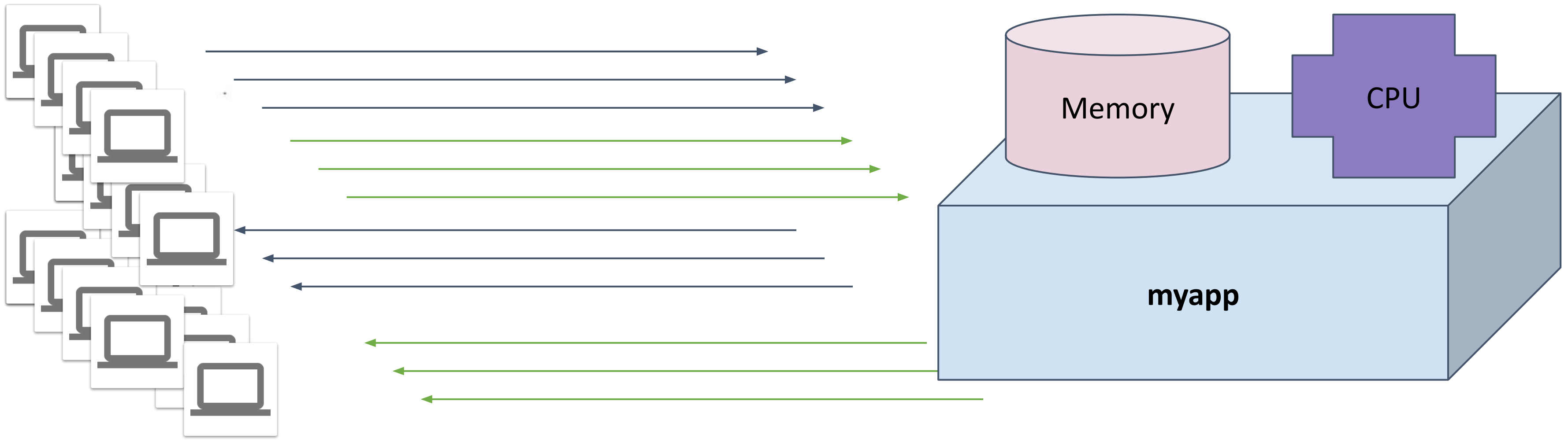
Solving the Performance Problem



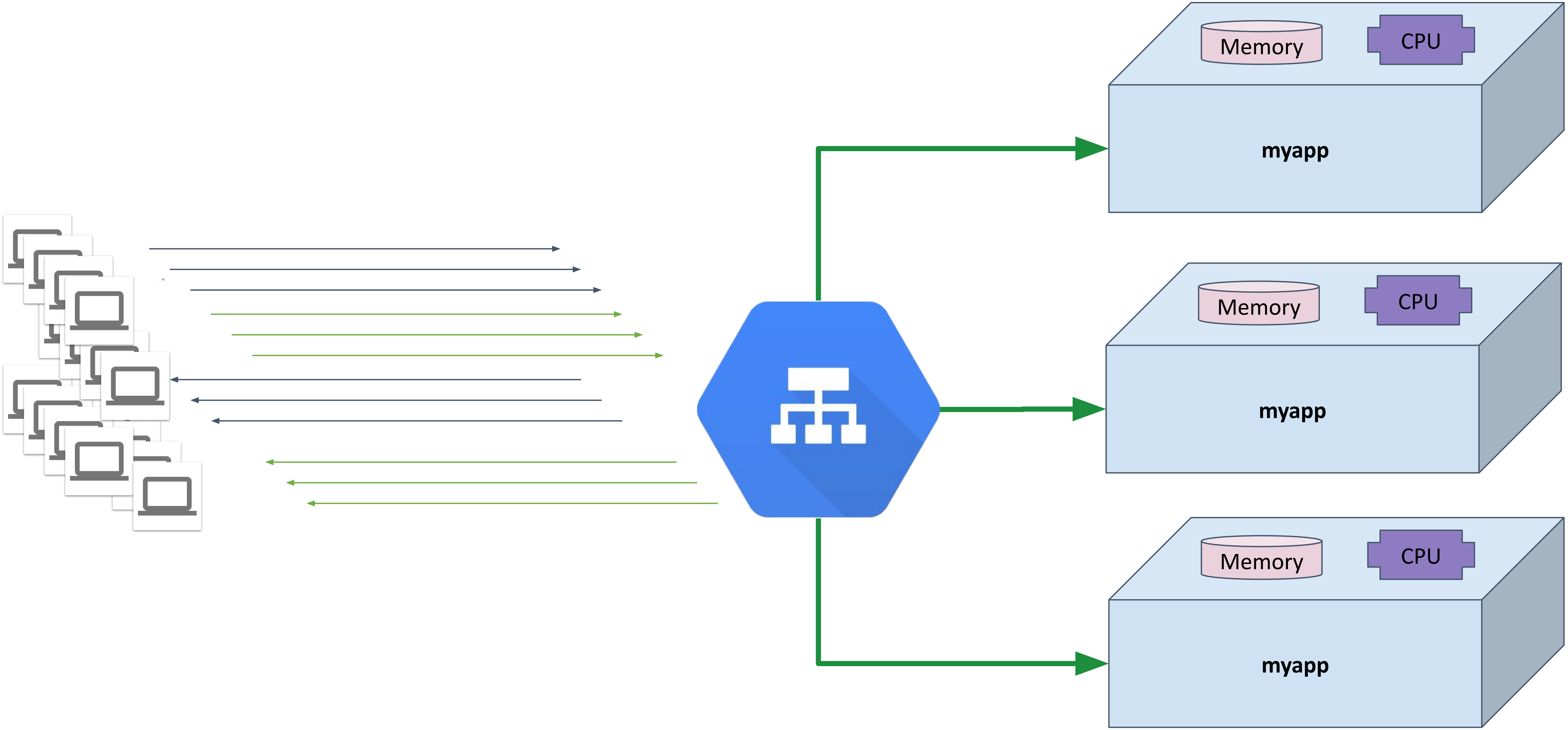
Solving the Performance Problem



Scaling UP



Scaling OUT

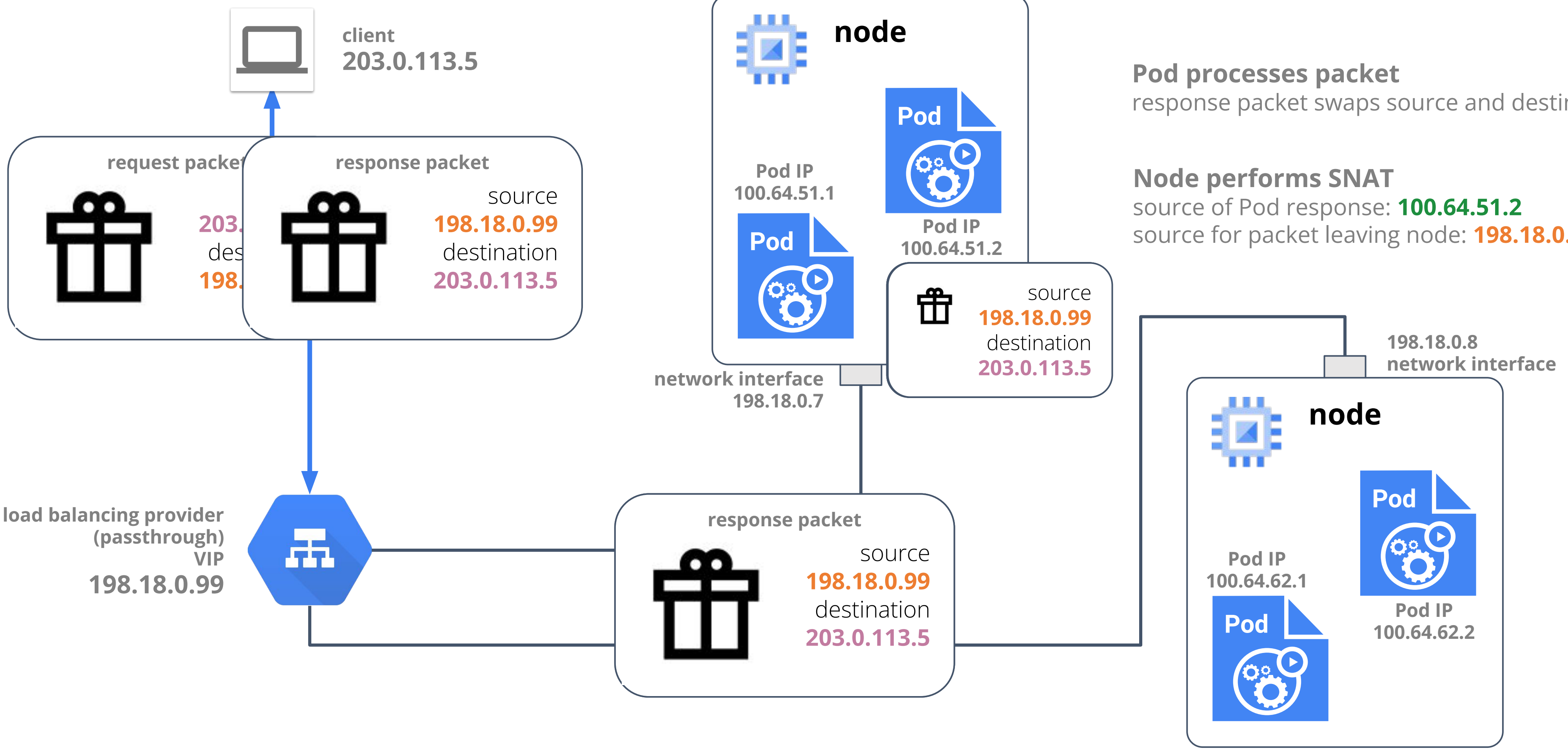


Passthrough vs. Proxy load balancing

Passthrough
any IP protocol – classified as OSI L3/L4
routes packets, does not terminate connections
request packets arrive on the network interface of a node having a destination IP addresses that matches the load balancer's forwarding rule (VIP)
node performs DNAT on the request packet, routing it to a serving Pod
Pod replies
node performs SNAT on the response packet and emits the response packet via its network interface Source matches the load balancer VIP *Direct Server Return (DSR)*
Perfect for Services of type LoadBalancer

Proxy
TCP based, two TCP connections
TCP connection between client and proxy software TCP connection between proxy and Pod
TCP or OSI L4+ HTTP(S), HTTP/2, Redis, etc.
in an ideal implementation Proxy sends request packets to a Pod IP address *Container Native*
Pod replies
in an ideal implementation Pod IP addresses are routable in the network
Proxy receives Pod's response packet, proxy copies response data into its response packet to the client
some LoadBalancer Service implementations use proxies, but Proxies are perfect for Gateway and Ingress

Life of a packet

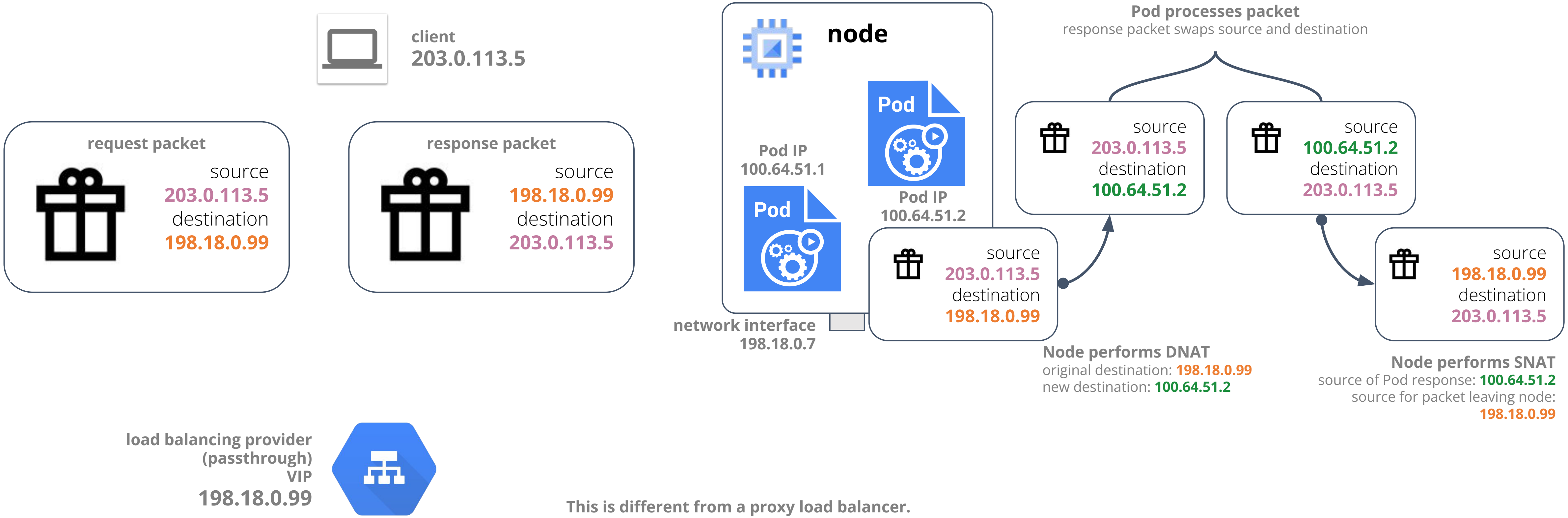


Node performs DNAT
original destination: 198.18.0.99
new destination: 100.64.51.2

Pod processes packet
response packet swaps source and destination

Node performs SNAT
source of Pod response: 100.64.51.2
source for packet leaving node: 198.18.0.99

Life of a packet



This is different from a proxy load balancer.
The load balancer doesn't deliver packets with destinations matching a node IP address (nodePort).
The load balancer routes packets to the network interface of the node VM.
The destination for the packet that arrives on the node matches the load balancer VIP.

Load Balancer Inclusive

externalTrafficPolicy

Two possible values, Cluster or Local

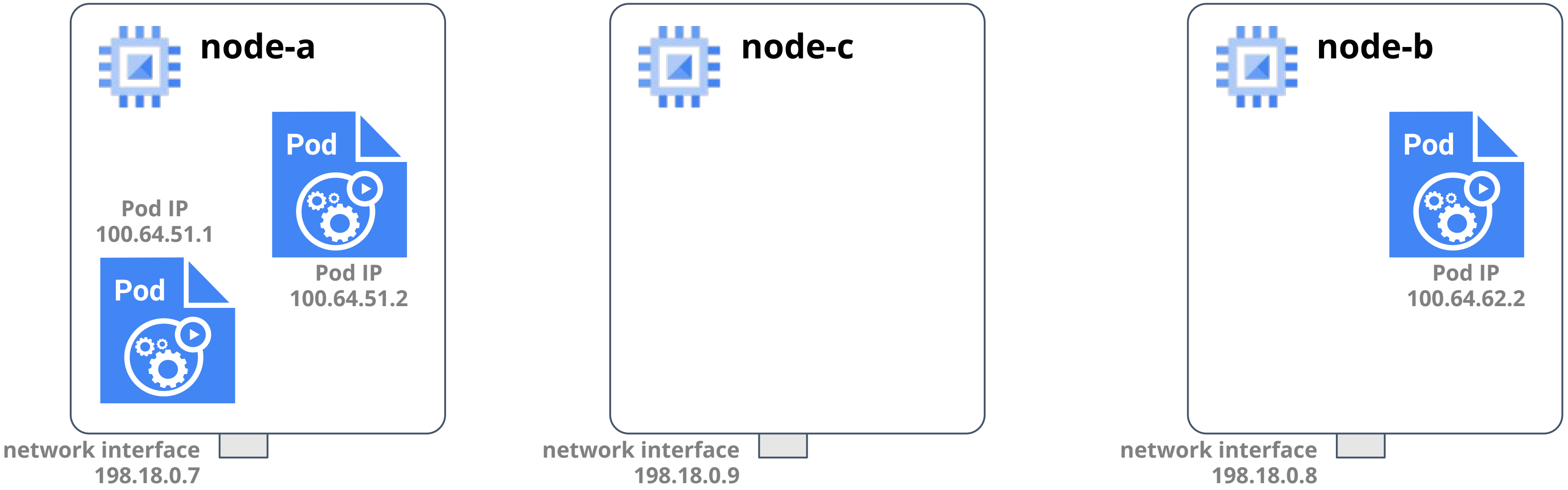
Helps the load balancer decide which nodes receive load balanced packets

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lb-service
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Node grouping, health checking

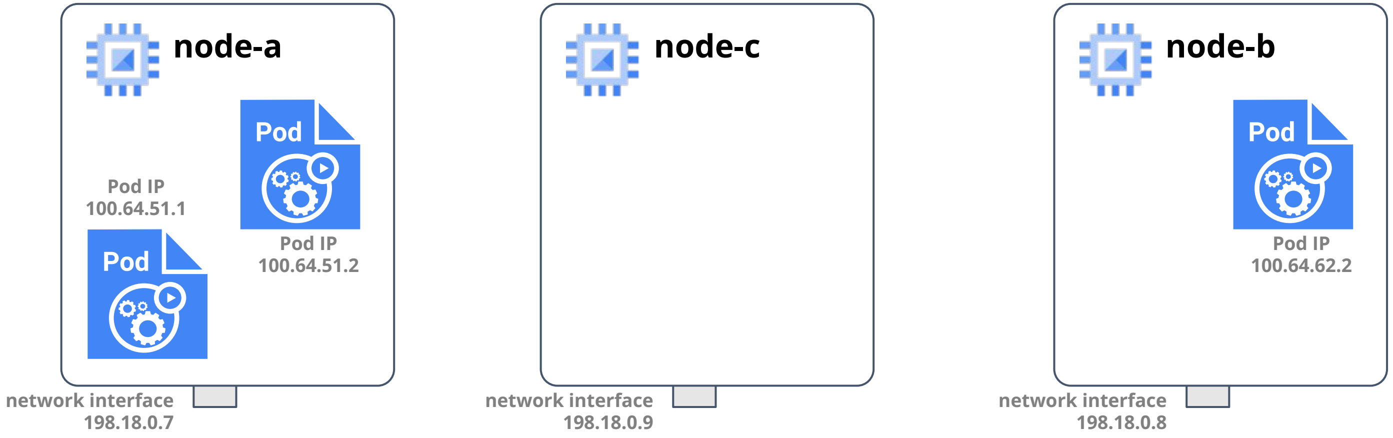
Method 1: **Group nodes using instance groups**

Make instance groups, collectively holding all nodes ...whether or not a node has a serving Pod for the Service	Decide which nodes receive packets externalTrafficPolicy + load balancer health checks



	node-a	node-c	node-b
externalTrafficPolicy: **Cluster**	Healthy	Healthy	Healthy
externalTrafficPolicy: **Local**	Healthy	Not healthy	Healthy

Node grouping, health checking



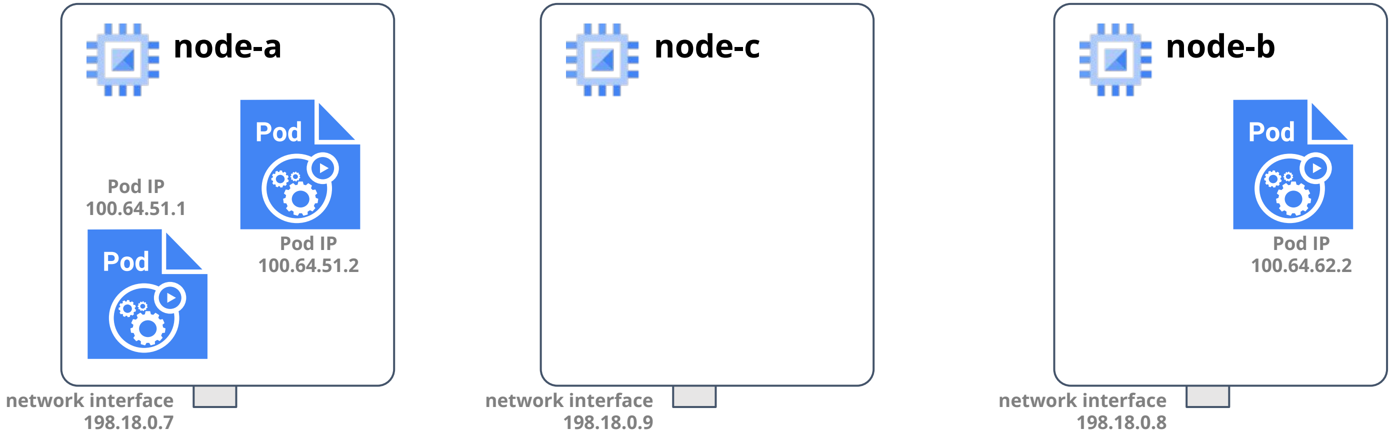
	node-a	node-c	node-b
externalTrafficPolicy: **Cluster**	Healthy	Healthy	Healthy
externalTrafficPolicy: **Local**	Healthy	Not healthy	Healthy

Load balancer health checks

Packets sent from probe systems that operate as part of the load balancer provider
Not a readiness check, not a liveness check

The kube-proxy or its “equivalent” (e.g. cilium-agent) receives the load balancer health check packets and responds to them

Node grouping, health checking



	node-a	node-c	node-b
externalTrafficPolicy: **Cluster**	Healthy	Healthy	Healthy
externalTrafficPolicy: **Local**	Healthy	Not healthy	Healthy

Conditions for being load balancer healthy

For externalTrafficPolicy: Cluster, every node is always healthy; very simple.

For externalTrafficPolicy: Local, a node is healthy if it has *at least* one serving Pod that meets both criteria:

YES passing readiness probe

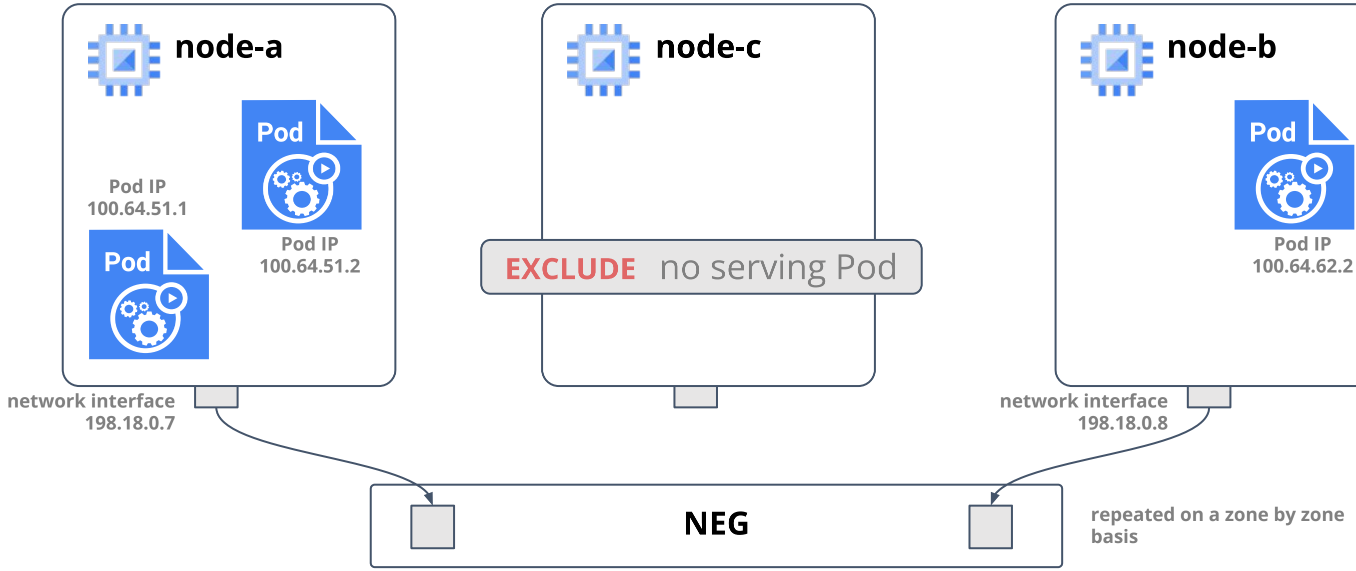
YES is *not* Terminating

Node grouping, health checking

Method 2: **Group nodes using network endpoint groups (NEGs)**

where each endpoint identifies the network interface of the *node*

externalTrafficPolicy: **Local**



Within the NEGs, only include the nodes with at least one non terminating serving Pod.
These nodes are also expected to pass the load balancer health checks.

A node is healthy if it has *at least* one serving Pod that is:

YES passing readiness probe

YES is *not* Terminating

Node grouping, health checking

Method 2: **Group nodes using network endpoint groups (NEGs)**

where each endpoint identifies the network interface of the *node*

externalTrafficPolicy: **Cluster**

See:
https://cloud.google.com/kubernetes-engine/docs/concepts/service-load-balancer#neg_backends

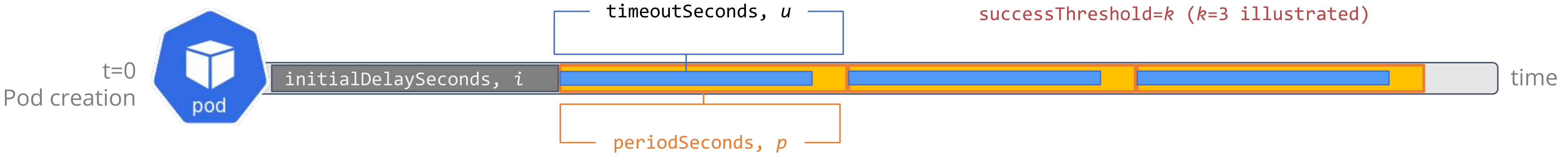
NEG membership consists of a *subset of nodes*
whether or not the nodes have at least one non-terminating serving Pod

When using eTP Local

When using externalTrafficPolicy: **Local**

Define a meaningful readiness probe for the main container of the serving Pods.

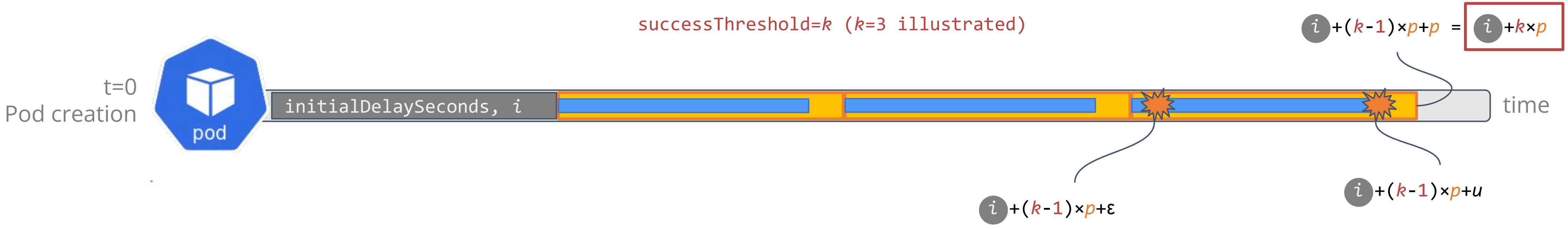
The load balancer health check will pass after the readiness probe passes.
The load balancer health check will fail after the readiness probe fails.



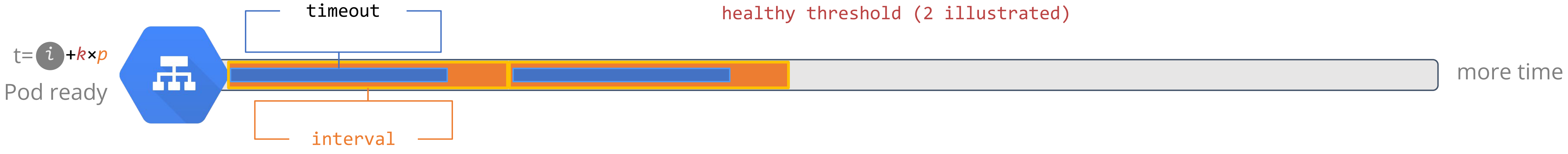
For sanity:
$u < p$

When using eTP Local

The load balancer health check will pass after the readiness probe passes.
When will the readiness probe pass?



When will the load balancer health check pass?



$\boxed{i+k\times p+(\text{LB healthy threshold})\times(\text{LB interval})}$

When using eTP Local

What are the active backends for the load balancer?

The nodes that pass the load balancer health check.
This is all that matters to the load balancer!

The node grouping method and externalTrafficPolicy indirectly determine the active backends.

Back to life of a packet

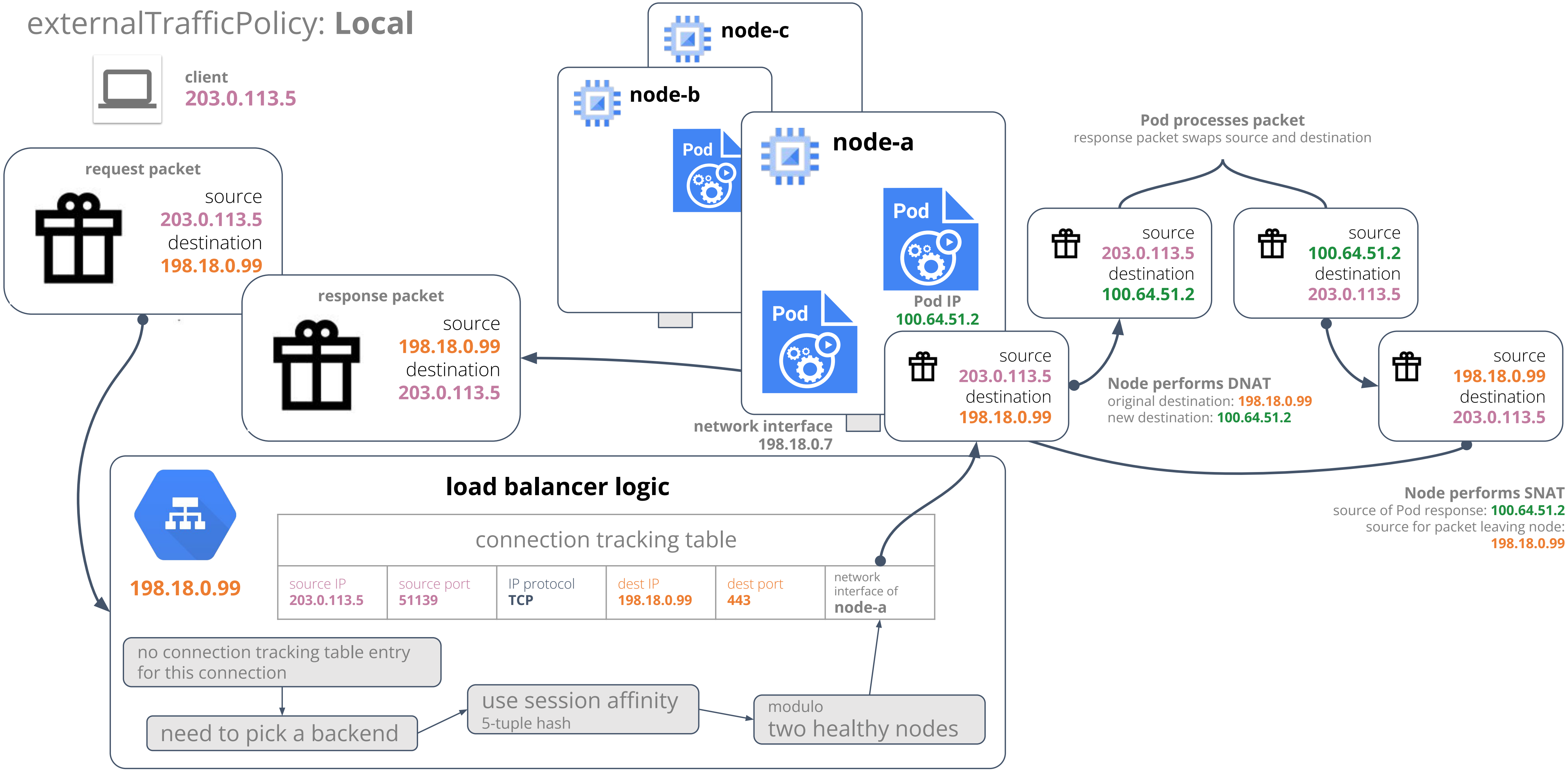
externalTrafficPolicy: **Local**
three nodes in the cluster
two nodes have serving Pods

none of the serving Pods are terminating
all of the serving Pods pass readiness probes

two nodes with serving Pods pass load balancer health checks
two nodes with serving Pods are the load balancer's active backends

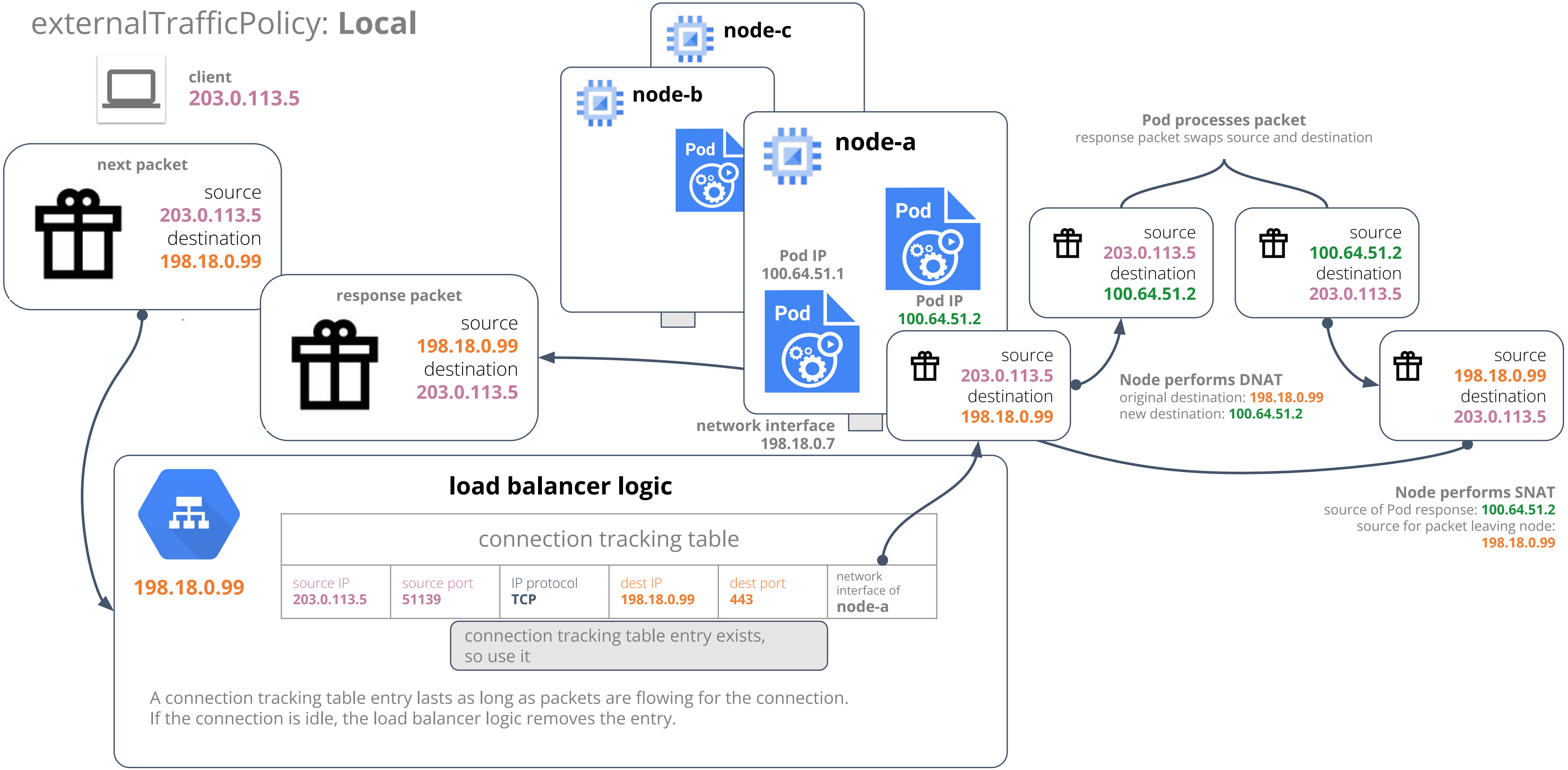
Back to life of a packet

externalTrafficPolicy: **Local**



Back to life of a packet

externalTrafficPolicy: **Local**



Back to life of a packet

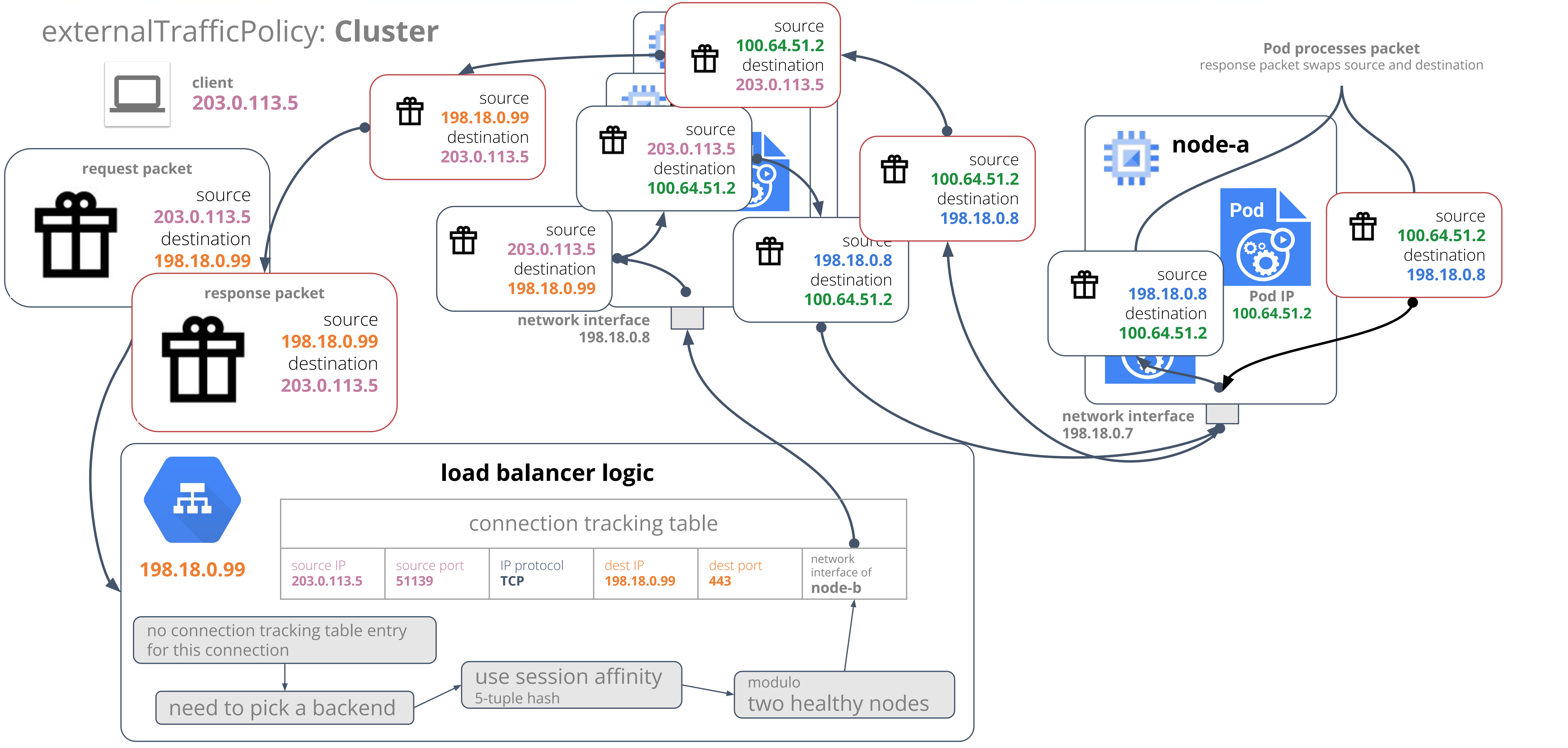
externalTrafficPolicy: **Cluster**

three nodes in the cluster
two nodes have serving Pods

none of the serving Pods are terminating
all of the serving Pods pass readiness probes

two nodes with serving Pods pass load balancer health checks
two nodes with serving Pods are the load balancer's active backends

Back to life of a packet



Idle connections

When the connection tracking table entry is removed due to an idle connection, the next packet is processed as if it were the first packet in a *new* connection.

But:
The next packet isn't part of a new connection.

When are idle connections problematic?

Idle connections

Idle connections are not problematic as long as the number of active backends is constant.

New TCP connection
Load balancer picks a backend (using session affinity hash)
Connection tracking table entry is created
Connection tracking table entry used to route packets to the node's NIC
Connection becomes idle for longer than the connection tracking table can tolerate
Connection tracking table entry evicted
(Client and serving Pod on node think the connection is still active.)
Next packet sent
Load balancer picks a backend (using session affinity hash) *again*
Picks the same backend
Identical connection tracking table entry re-created
Connection tracking table entry used to route packets to the *same* node's NIC

Idle connections

Idle connections are problematic when the number of active backends changes *and* a connection tracking table entry is removed.

New TCP connection
Load balancer picks a backend (using session affinity hash)
Connection tracking table entry is created
Connection tracking table entry used to route packets to the node's NIC
Connection becomes idle for longer than the connection tracking table can tolerate
Connection tracking table entry evicted
(Client and serving Pod on node think the connection is still active.)
Next packet sent
Load balancer picks a backend (using session affinity hash) *again*
Picks a different backend (node)!
New connection tracking table entry created
First packet (without a SYN flag) is delivered to the NIC of the new backend
Kernel of node issues a TCP RST (reset) – correct behavior
Nothing appears in application logs – also correct behavior

Idle connections

Idle connections are problematic when the number of active backends changes
and there is no connection tracking table entry present (any longer).

total nodes **VARIABLE** nodes with serving Pods **CONSTANT** **GOAL** keep active backend count stable	total nodes **CONSTANT** nodes with serving Pods **VARIABLE** **GOAL** keep active backend count stable	total nodes **VARIABLE** nodes with serving Pods **VARIABLE** **GOAL** don't let the connection go idle
externalTrafficPolicy: **Local**	externalTrafficPolicy: **Cluster**	externalTrafficPolicy **Local** or **Cluster** *and* use TCP keepalives

externalTrafficPolicy determines which nodes are active backends…
Maintain the same set of active backends so that
the backend selection method picks the same node.
Create a new connection tracking table entry just like the first.

when in doubt…

TCP keepalives

KubeCon | CloudNativeCon North America 2023

A TCP keepalive is a special packet which is designed to keep connection tracking table entries live in an intermediate system (like a load balancer).

https://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/

(A TCP keepalive is not the same thing as what some proxy software calls an “HTTP keepalive.” The term “HTTP keepalive” is better translated to “TCP idle.”)

Application code	Open sockets with the `SO_KEEPALIVE` option.	

Keepalive options	Kernel defaults
	`tcp_keepalive_time`
	`tcp_keepalive_intvl`
		Socket option
		TCP_KEEPIDLE
		TCP_KEEPINTVL

`tcp_keepalive_time (TCP_KEEPIDLE)`: Send the first keepalive packet after this many seconds from last data packet.
`tcp_keepalive_intvl (TCP_KEEPINTVL)`: Send subsequent keepalive packets this often.
The connection will stay open unless a certain number of keepalive packets are not answered (defined by tcp_keepalive_probes or TCP_KEEPCNT).

TCP keepalives

Kernel defaults

Ref: https://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/

```
/sbin/sysctl -w net.ipv4.tcp_keepalive_time=30
# or write to /proc/sys/net/ipv4/tcp_keepalive_time
```

Set TCP_KEEPIDLE to 30s.

```
/sbin/sysctl -w net.ipv4.tcp_keepalive_intvl=55
# or write to /proc/sys/net/ipv4/tcp_keepalive_intvl
```

Set TCP_KEEPINTVL to 55s.

```
/sbin/sysctl -w net.ipv4.tcp_keepalive_probes=5
# or write to /proc/sys/net/ipv4/tcp_keepalive_probes
```

After five unacknowledged keepalive packets, consider connection closed

TCP keepalives

Istio + EnvoyFilter example

Ref: https://support.f5.com/csp/article/K00026550

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ingress-gateway-socket-options
  namespace: istio-system
spec:
  configPatches:
  - applyTo: LISTENER
    match:
      context: GATEWAY
    patch:
      operation: MERGE
      value:
        socket_options:
        - int_value: 1 # keepalive on
          level: 1
          name: 9
          state: STATE_PREBIND
        - int_value: 30 # seconds
          level: 6 # TCP protocol
          name: 4
          state: STATE_PREBIND
        - int_value: 30 # seconds
          level: 6 # TCP protocol
          name: 5
            state: STATE_PREBIND
```

Open the socket with the SO_KEEPALIVE option.

Set TCP_KEEPIDLE to 30s.

Set TCP_KEEPINTVL to 30s.

Rollouts, scale up, scale down

KubeCon | CloudNativeCon North America 2023

During rollouts, scaling up, and scaling down a Workload...

total nodes **VARIABLE**

nodes with serving Pods **VARIABLE**

GOAL 1 don't let the connection go idle

GOAL 2 allow existing connections to close naturally

We need the load balancer health check to fail *quickly* in order to repel *new* connections.

We need the serving Pod to keep processing packets* for *existing* connections even after the load balancer health check has failed.

* for a duration that meets our needs

Rollouts, scale up, scale down

total nodes **VARIABLE**

nodes with serving Pods **VARIABLE**

GOAL 1 don't let the connection go idle

GOAL 2 allow existing connections to close naturally

GOAL 2b keep processing packets for existing connections

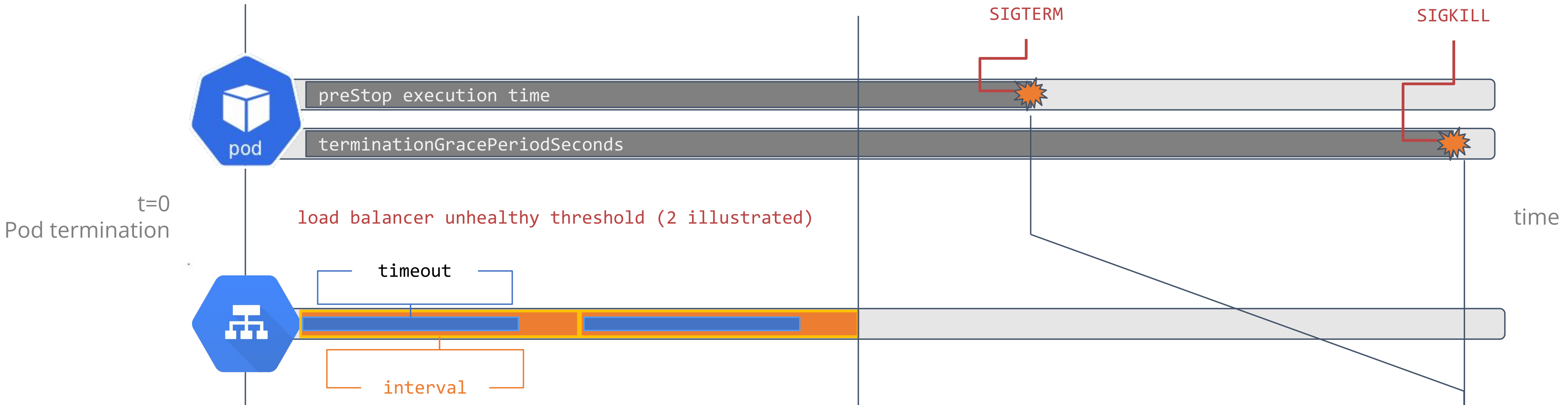
With externalTrafficPolicy **Local**, a node only passes the load balancer health check if it has at least one ready, non-terminating serving Pod.

With externalTrafficPolicy **Cluster**, a node passes the load balancer health check regardless of serving Pod state.

Keep the Pod running for a reasonable amount of time.

Terminating Pod lifecycle

Three timelines in parallel



New connections still arrive

KEP-1669 do the right thing

No new connections
Keep processing existing ones

TEST some software stops serving when it receives SIGTERM

TEST some software keeps serving past SIGTERM

Rollouts, scale up, scale down

If:	Then:
software stops serving at SIGTERM	Make *preStop* execution time and *terminationGracePeriodSeconds* sufficiently long
software keeps serving beyond SIGTERM	Ensure that *terminationGracePeriodSeconds* is long enough

LB HEALTH CHECK time to unhealthy < **preStop** execution time < **terminationGracePeriodSeconds**

Rollouts, scale up, scale down

total nodes **VARIABLE**

nodes with serving Pods **VARIABLE**

GOAL 1 don't let the connection go idle

GOAL 2a fail load balancer health checks quickly

GOAL 2b keep processing packets for existing connections

With externalTrafficPolicy **Local**, a node only passes the load balancer health check if it has at least one ready, non-terminating serving Pod.

With externalTrafficPolicy **Cluster**, a node passes the load balancer health check regardless of serving Pod state.

KEP-1669 route packets to terminating Pods as a last resort

LB HEALTH CHECK time to unhealthy < **preStop** execution time < **terminationGracePeriodSeconds**

Please scan the QR Code above
to leave feedback on this session