Acronis

#CyberFit

Deep dive into load balancing

Migrating a front-end proxy to Kubernetes

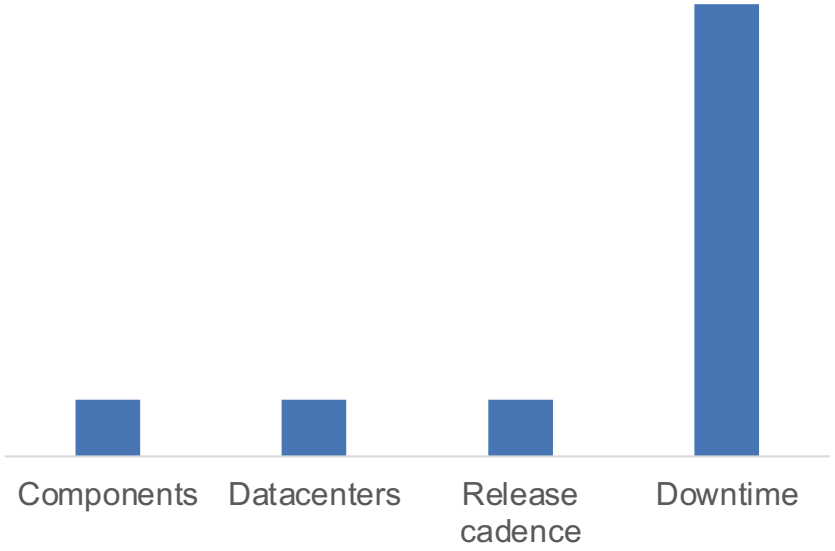
But why?

- Everybody

Getting up to speed

- Initial datacenter deployment
  - Manual deployments via shell/bash
  - Works for a few services

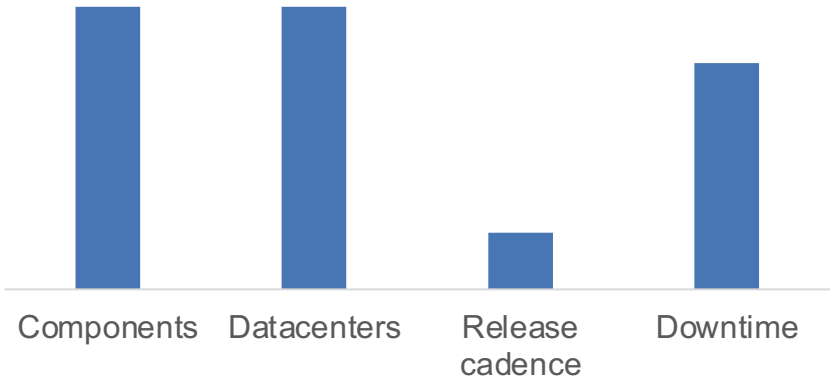
- New problem: unsustainable for many components and datacenters



Getting up to speed

- Addition of Ansible
  - Increased maintainability
  - More datacenter/components
  - Downtime scales with the number of components

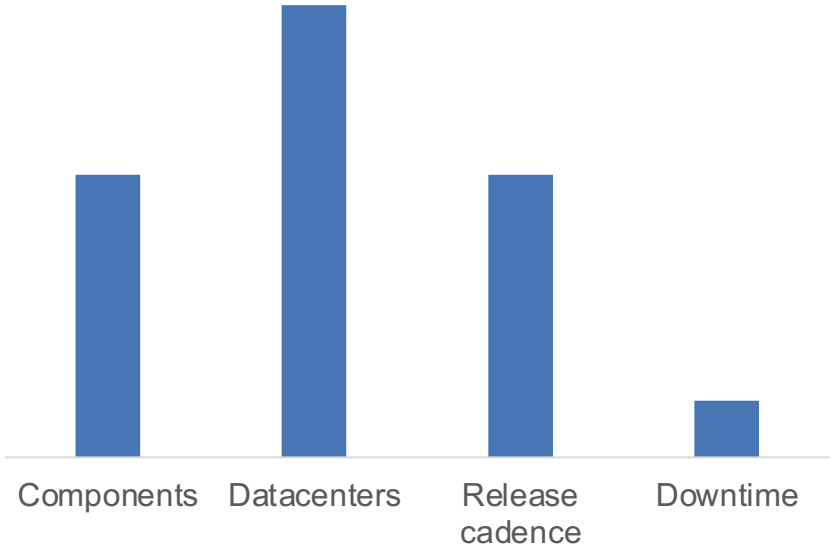
- New problem: downtime scales linearly with the number of components



Getting up to speed

- Parallel Ansible/Jenkins tooling
  - Automation allows to scale up releases and datacenters
  - Parallel Ansible execution minimizes downtime

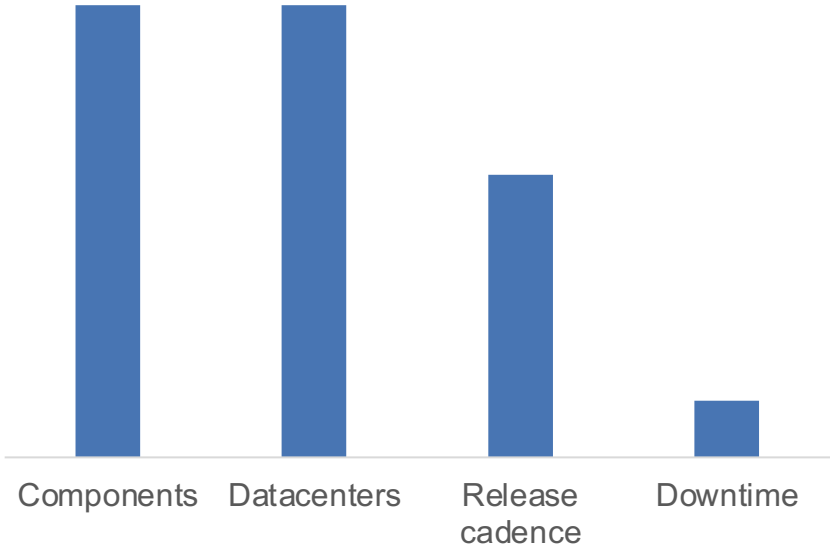
- New problem: operations, monitoring and resource use scale linearly with the number of components



Getting up to speed

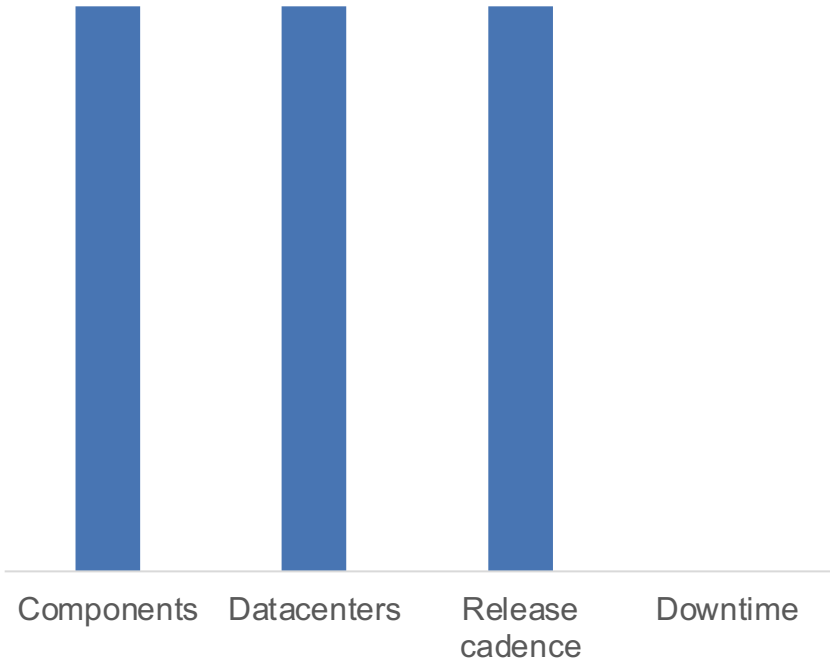
- Introduction of Kubernetes
  - Reduced resource use for components
  - Unified monitoring/operations solutions
  - Faster component deployment times

- New problem: component description in both Ansible and Helm now



We need to go further

- Time to do continuous delivery
  - Will allow us to scale to even more components
  - Almost infinite release cadence
  - No downtime
- How to achieve this?
  - Infra configuration should consist purely of Kubernetes manifests
  - No component-specific Ansible code
  - Biggest offender is Nginx configuration

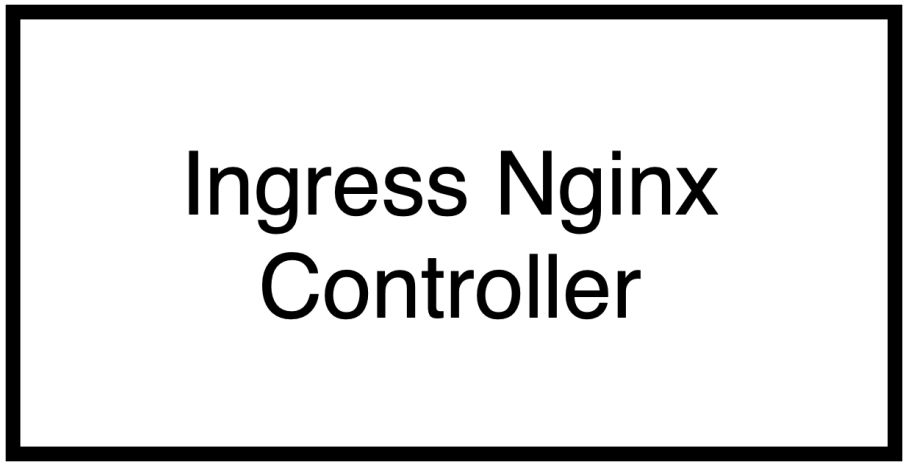


Velocity!

- DevOps

Let’s migrate front-ends to Kubernetes!

Let’s migrate front-ends to Kubernetes…

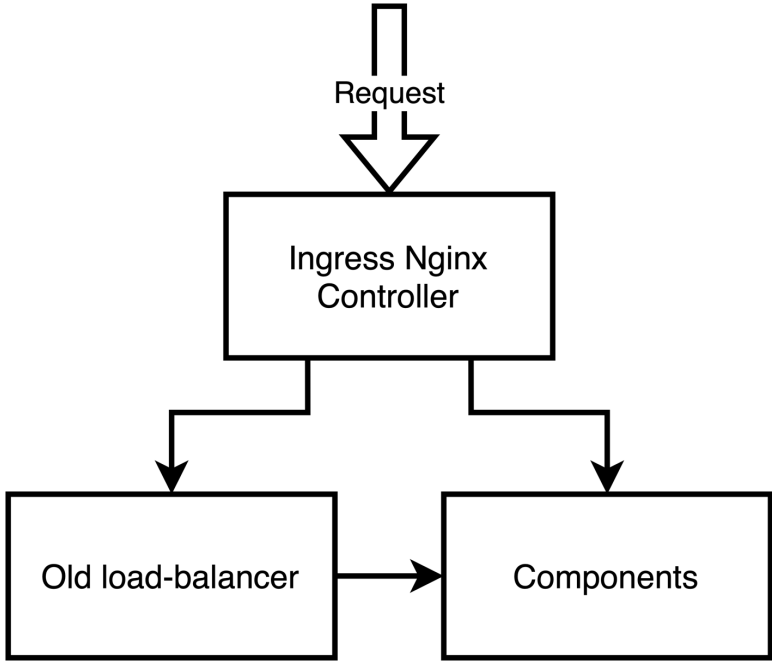


Migrating existing routes

- 100+ components
- Migrating everyone at once is error-prone and complicated
- Various business priorities in component teams

Migrating existing routes

- 100+ components
- Migrating everyone at once is error-prone and complicated
- Various business priorities in component teams
- Automatic fallback is needed
- A catch-all Ingress object



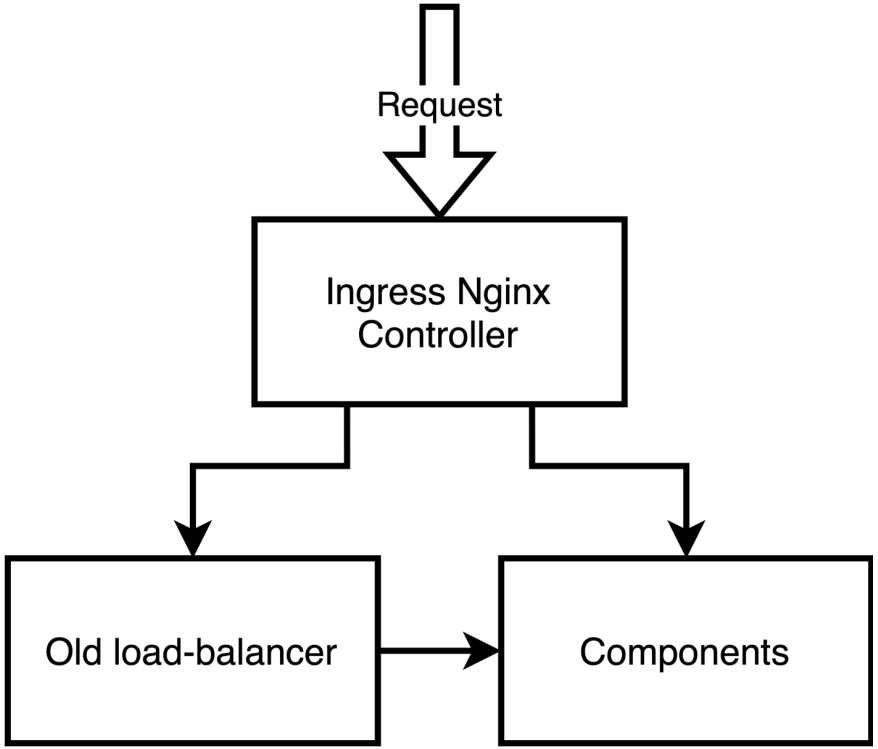
Migrating existing routes

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: routing-nginx
  name: legacy-fallback
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: legacy-fallback
            port:
              name: http
```

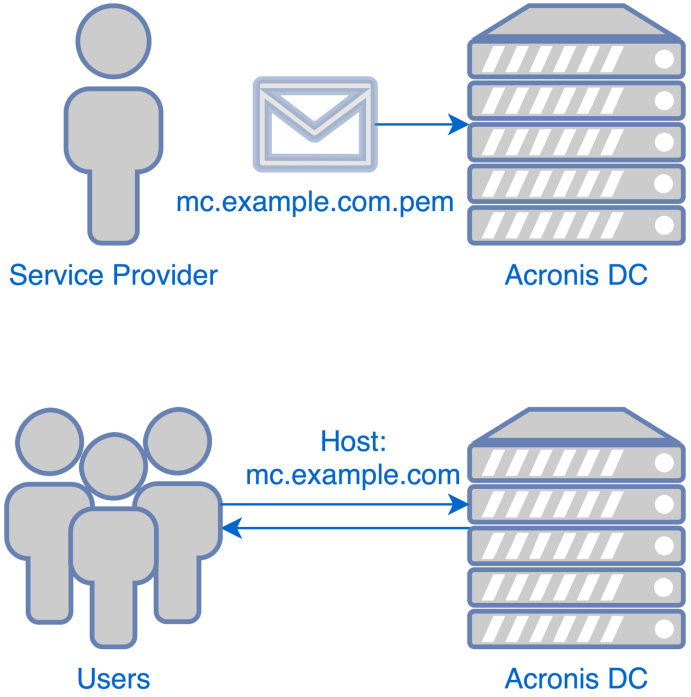
```yaml
apiVersion: v1
kind: Service
metadata:
  name: legacy-fallback
spec:
  selector: {}
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
```

```yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: legacy-fallback
subsets:
- addresses:
  - ip: 10.34.145.199
  ports:
  - name: http
    port: 8080
```

Migrating existing routes



Dynamic changes in SSL certificates (Branding)



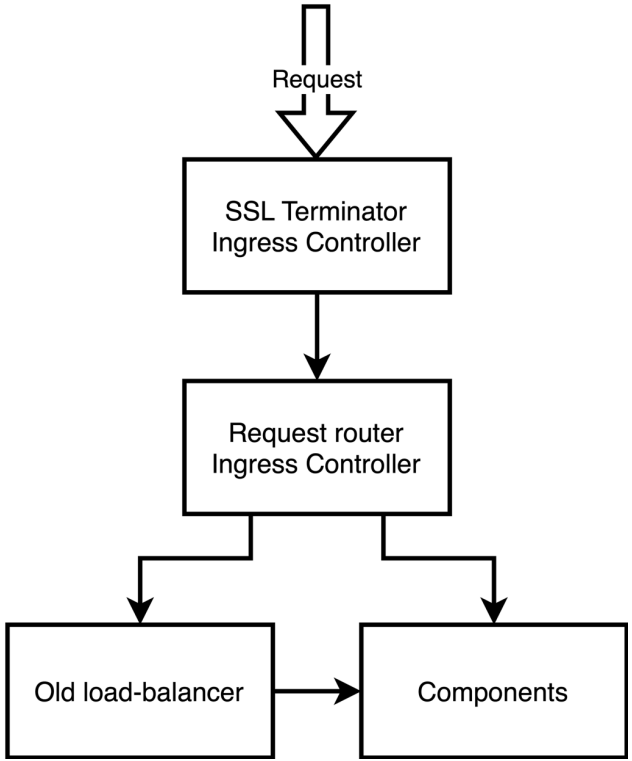
Dynamic changes in SSL certificates (Branding)

- The list of domains and certificates changes on-the-fly
- Current Ingress API merges SSL termination and request routing together
- Request routing is handled by multiple Ingress objects
- Easy solution: each Ingress should need to contain the list of all the domains

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /api/example-service/
        backend:
          service:
            name: example-service
            port:
              name: api
        pathType: Prefix
  tls:
  - hosts:
    - example.com
    secretName: example.com
```

Dynamic changes in SSL certificates (Branding)

- Solution: two Ingress Controllers
  - One for SSL termination
  - One for request routing
- Two hops instead of one
  - An extra performance hit
  - To be fixed once Gateway API is out
    - https://gateway-api.sigs.k8s.io

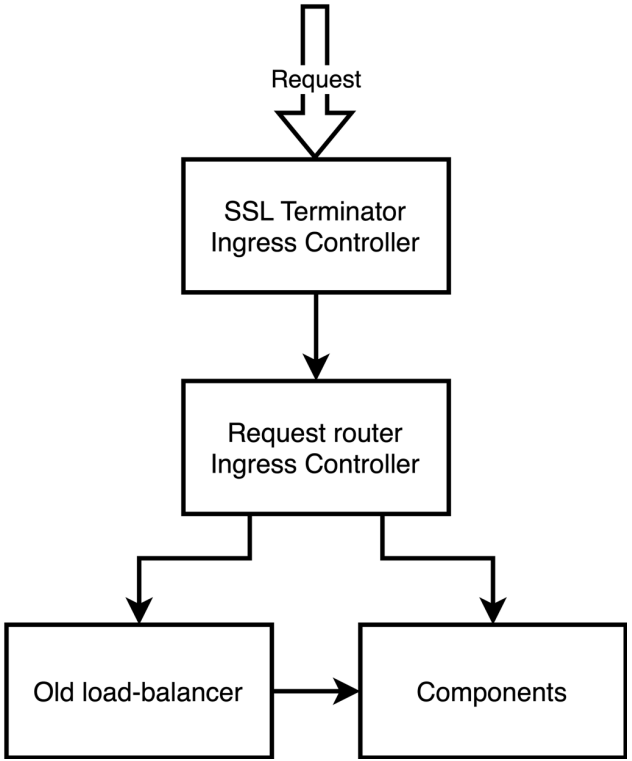


Dynamic changes in SSL certificates (Branding)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: public-nginx
  name: branded-domain.example.com
spec:
  rules:
  - host: branded-domain.example.com
    http:
      paths:
      - backend:
          service:
            name: routing-ingress-controller
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
  tls:
  - hosts:
    - branded-domain.example.com
    secretName: branded-domain.example.com
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: routing-nginx
  name: example-ingress
spec:
  rules:
  - host: ~
    http:
      paths:
      - path: /api/example-service/
        backend:
          service:
            name: example-service
            port:
              name: api
        pathType: Prefix
```

Dynamic changes in SSL certificates (Branding)



Running multiple ingress controllers on same nodes

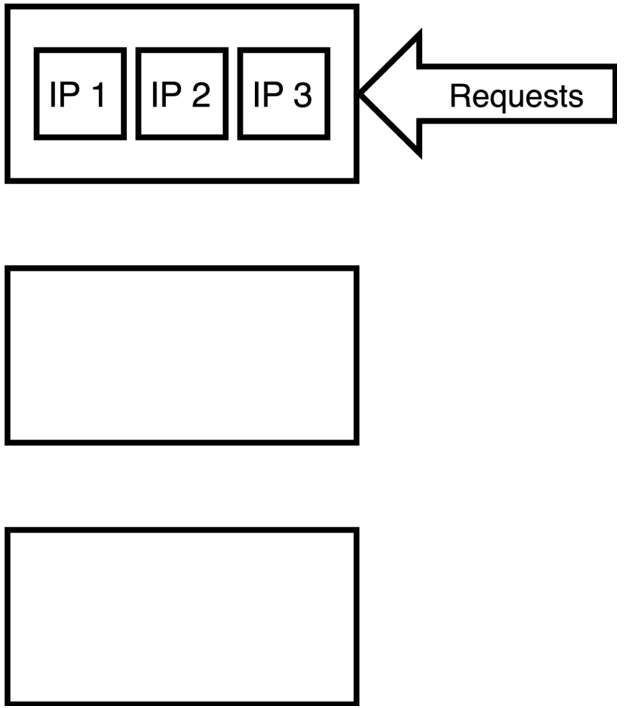
- Two kinds of endpoints
  - REST API Endpoints
  - Websocket API Endpoints
- Separation is needed to contain the blast radius
  - Better to not have two separate kinds of nodes
  - Dst-ip-based routing is needed

High Availability

- Current situation is not truly HA
  - Multiple Nginx instances
  - DNS-based High Availability
- Kubernetes-based solution should achieve HA
  - But we are on bare-metal…
  - Time for MetalLB?

Tiny rant about MetalLB

- The only mainstream solution for baremetal LB
- Each IP address is a separate leader election group
- No “node affinity” impedes troubleshooting
- No protection against address bunching



High Availability

- Time to look outside of the k8s ecosystem
- Keepalived to the rescue!
  - Battle-tested
  - High control over configuration
  - Customizable via health-checks and notify scripts
- First generate the configuration with Ansible, then implement as an operator

What do load balancers do?

- kube-proxy POV: LoadBalancer service is the same as a regular service
- Rewrite IP to the Pod IPs
- External load balancer assigns IP to the node's network interface + sets up node networking

```
apiVersion: v1
kind: Service
metadata:
  name: public-nginx
spec:
  externalIPs:
  - 10.34.8.49
  - 10.34.8.50
  externalTrafficPolicy: Local
  healthCheckNodePort: 32110
  type: LoadBalancer
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: public-nginx
    app.kubernetes.io/name: ingress-nginx
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
```

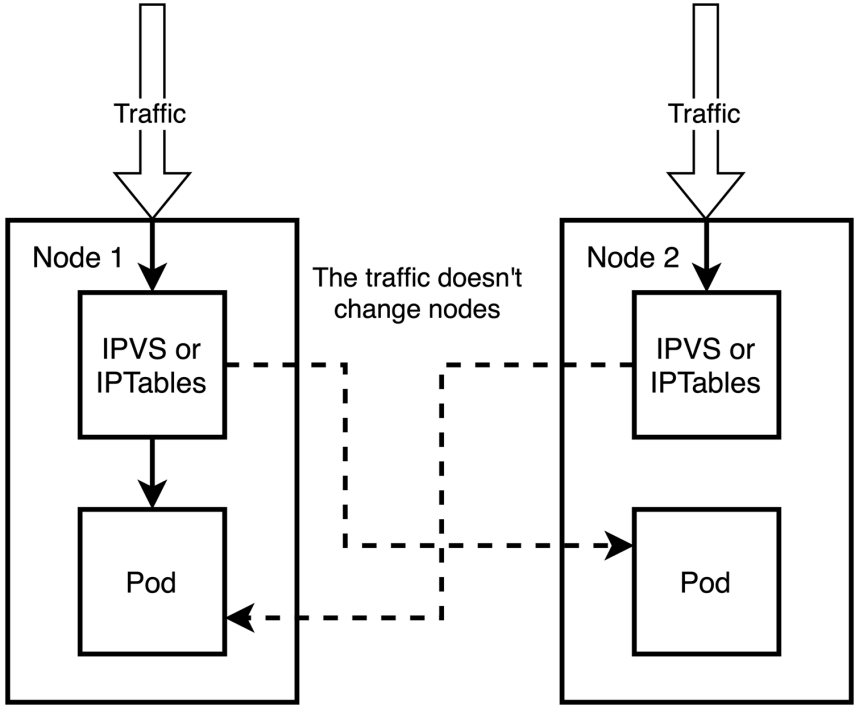
Kube-proxy and ExternalTrafficPolicy

- In order not to lose the client IP, policy should be set to Local
- Traffic is only routed to the ready pods on the current node to avoid NATs losing client IPs

```
apiVersion: v1
kind: Service
metadata:
  name: public-nginx
spec:
  externalIPs:
  - 10.34.8.49
  - 10.34.8.50
  externalTrafficPolicy: Local
  healthCheckNodePort: 32110
  type: LoadBalancer
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: public-nginx
    app.kubernetes.io/name: ingress-nginx
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
```

Kube-proxy and ExternalTrafficPolicy

- In order not to lose the client IP, policy should be set to Local
- Traffic is only routed to the ready pods on the current node to avoid NATs losing client IPs

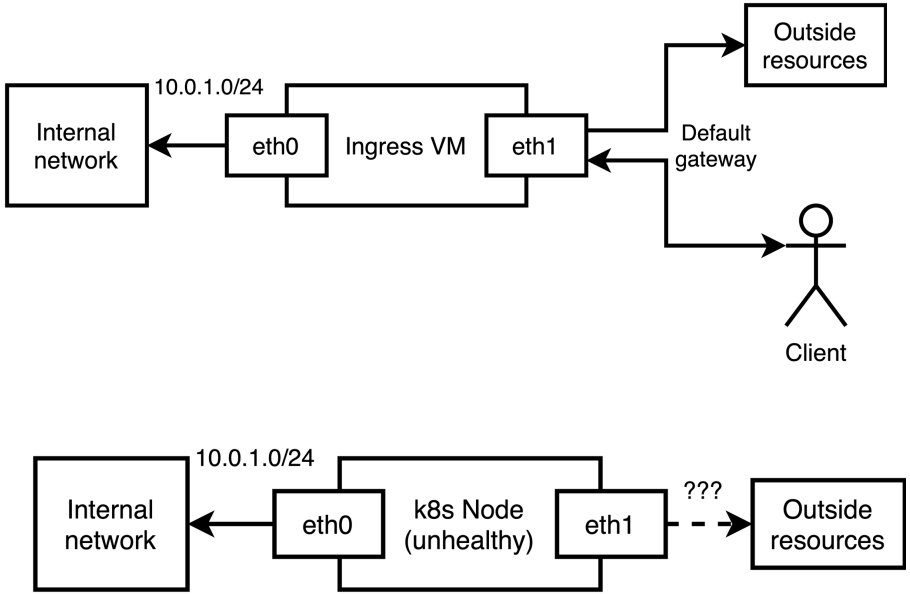


Setting up keepalived

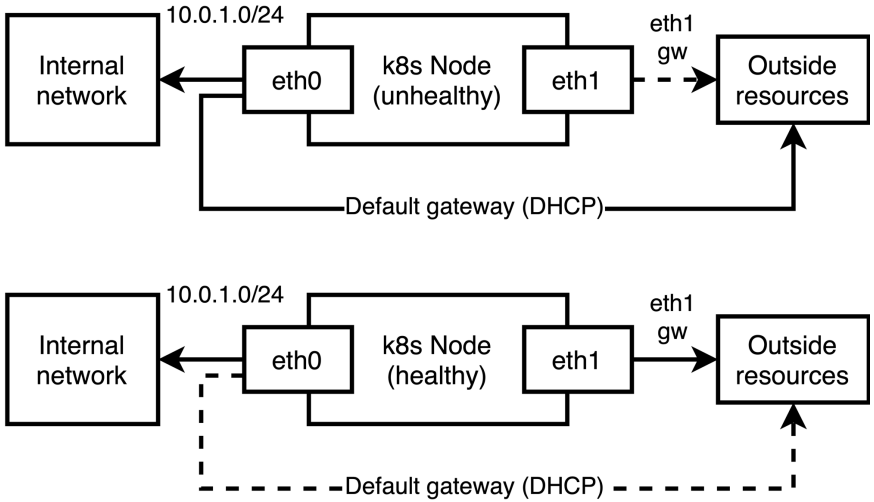
- Assigning IP addresses is pretty simple
- The more interesting challenge is setting up the network routing

```
vrrp_instance load-balancer-10.34.8.52 {
    state BACKUP
    interface eth0
    virtual_router_id 4
    priority 100

    unicast_src_ip 10.34.137.52
    unicast_peer {
        10.34.202.145
    }

    virtual_ipaddress {
        10.34.8.52/21 dev eth1
    }
}
```

Setting up keepalived



Setting up keepalived



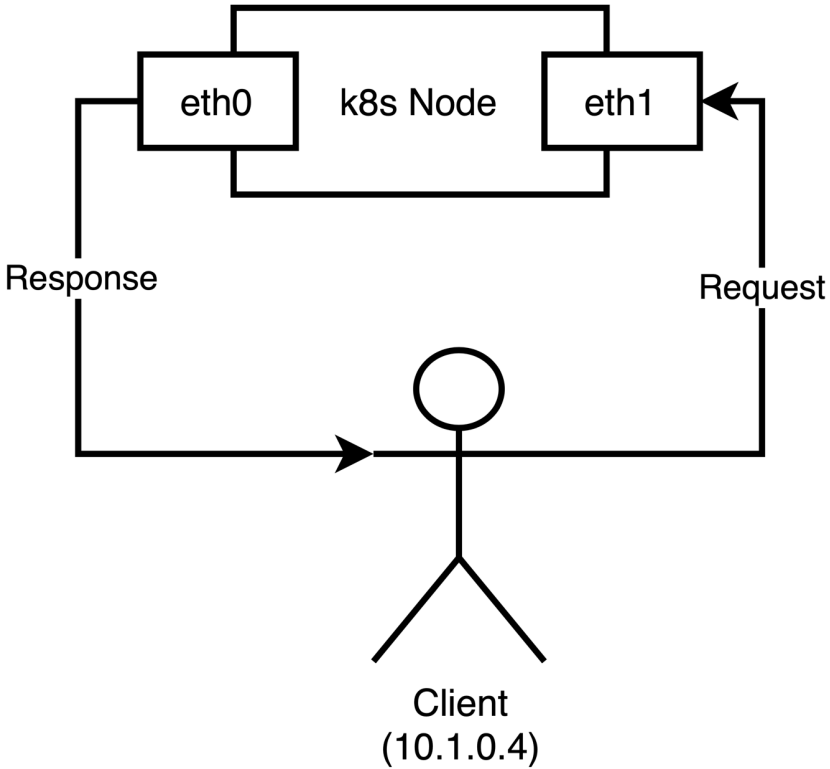

```
vrrp_instance load-balancer-10.34.8.52 {
    state BACKUP
    interface eth0
    virtual_router_id 4
    priority 100

    unicast_src_ip 10.34.137.52
    unicast_peer {
        10.34.202.145
    }

    virtual_ipaddress {
        10.34.8.52/21 dev eth1
    }

    virtual_routes {
        default via 10.34.8.1 dev eth1 metric 99
    }
}
```

Setting up keepalived

- Seems good
- Works in production
- Requires strict separation of the “internal” and “external” networks
- Impossible to test within the intranet

```
vrrp_instance load-balancer-10.34.8.52 {
    state BACKUP
    interface eth0
    virtual_router_id 4
    priority 100

    unicast_src_ip 10.34.137.52
    unicast_peer {
        10.34.202.145
    }

    virtual_ipaddress {
        10.34.8.52/21 dev eth1
    }

    virtual_routes {
        default via 10.34.8.1 dev eth1 metric 99
    }
}
```

Asymmetric routing

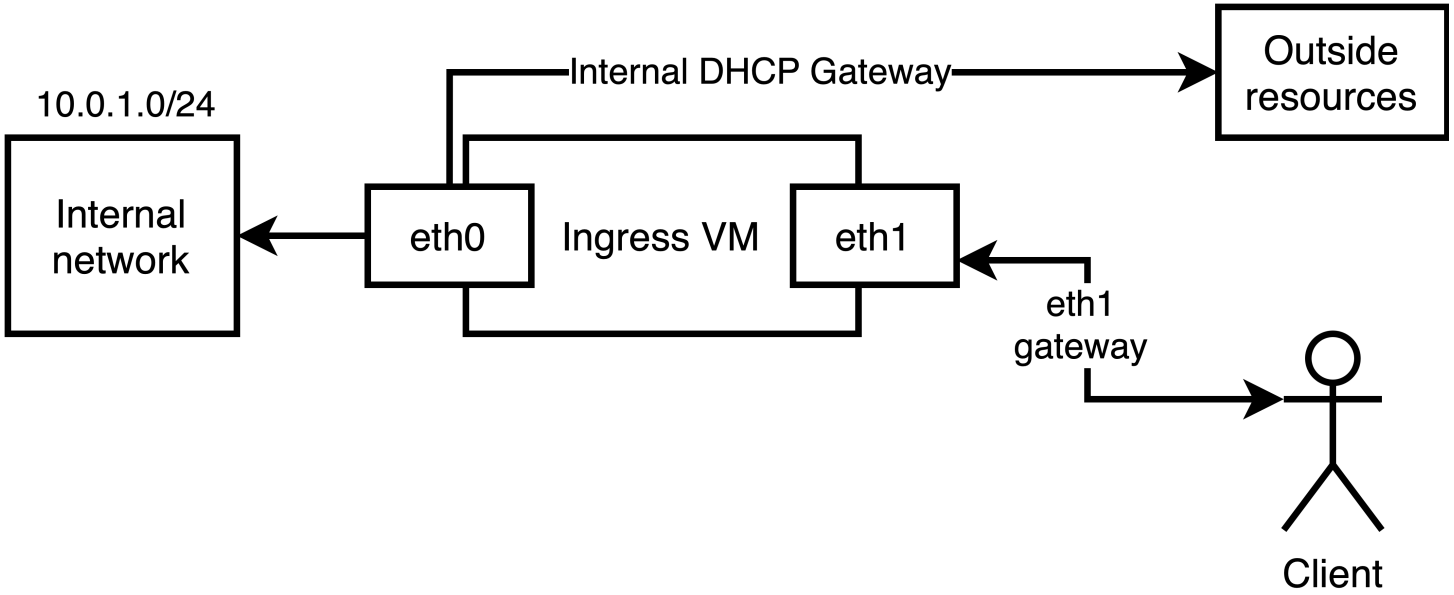


Policy-based routing to the rescue

- Built into the Linux Kernel
- Rules allow to direct specific packets to tables
- Tables contain regular routes

```
ip rule add from 10.34.8.52 lookup 100
ip route add default via 10.34.8.1 dev eth1 table 100
```

Policy-based routing to the rescue

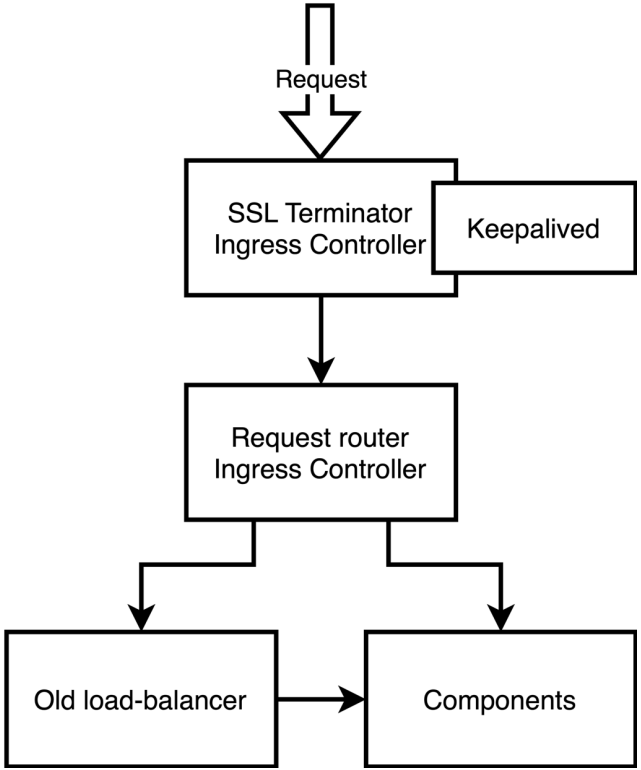


Setting up keepalived

- Keepalived supports virtual rules
- Assigned only while the IP is “owned” by this specific instance
- All incoming requests to public IPs go throught the gateway
- All outgoing requests go through the default routes

```
    interface eth0
    virtual_router_id 4
    priority 100

    unicast_src_ip 10.34.137.52
    unicast_peer {
        10.34.202.145
    }

    virtual_ipaddress {
        10.34.8.52/21 dev eth1
    }

    virtual_routes {
        default via 10.34.8.1 dev eth1 \
            table 131 src 10.34.8.52
    }

    virtual_rules {
        from 10.34.8.52 lookup 131
    }
}
```

Setting up keepalived



Kube-proxy and ExternalTrafficPolicy

- In order not to lose the client IP, policy should be set to Local
- Traffic is only routed to the ready pods on the current node
- By default (pre k8s 1.24) traffic is blackholed if no pod is ready
- Also affects in-cluster services trying to reach the public IP

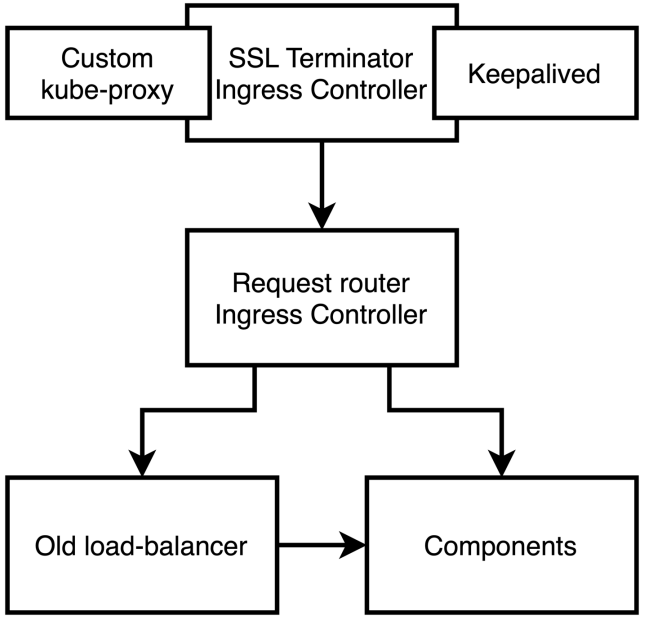
```
apiVersion: v1
kind: Service
metadata:
  name: public-nginx
spec:
  externalIPs:
  - 10.34.8.49
  - 10.34.8.50
  externalTrafficPolicy: Local
  healthCheckNodePort: 32110
  type: LoadBalancer
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: public-nginx
    app.kubernetes.io/name: ingress-nginx
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
```

Kube-proxy and ExternalTrafficPolicy

- In order not to lose the client IP, policy should be set to Local
- Traffic is only routed to the ready pods on the current node
- By default (pre k8s 1.24) traffic is blackholed if no pod is ready
- Also affects in-cluster services trying to reach the public IP
- Required a manual patch
- Fixed in k8s 1.24

```
apiVersion: v1
kind: Service
metadata:
  name: public-nginx
spec:
  externalIPs:
  - 10.34.8.49
  - 10.34.8.50
  externalTrafficPolicy: Local
  healthCheckNodePort: 32110
  type: LoadBalancer
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: public-nginx
    app.kubernetes.io/name: ingress-nginx
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
```

Mission accomplished!



Honorable mentions

- Messing around with strictARP
- Migrating keepalived to run inside k8s pods
- Aligning all network configurations across all datacenters
- Manually patching VM guest tools to stop breaking our networking
- Observing fights between sysctl.conf and kube-proxy

Lessons learned

- Migrating an operationally simple component can be difficult
- High Availability on bare-metal is not as hard as it seems
- Networking can be tricky

Acronis
Cyber Foundation

Building a more knowledgeable future

#CyberFit

Create, spread and protect knowledge with us!

- Building new schools
- Providing educational programs
- Publishing books

www.acronis.org