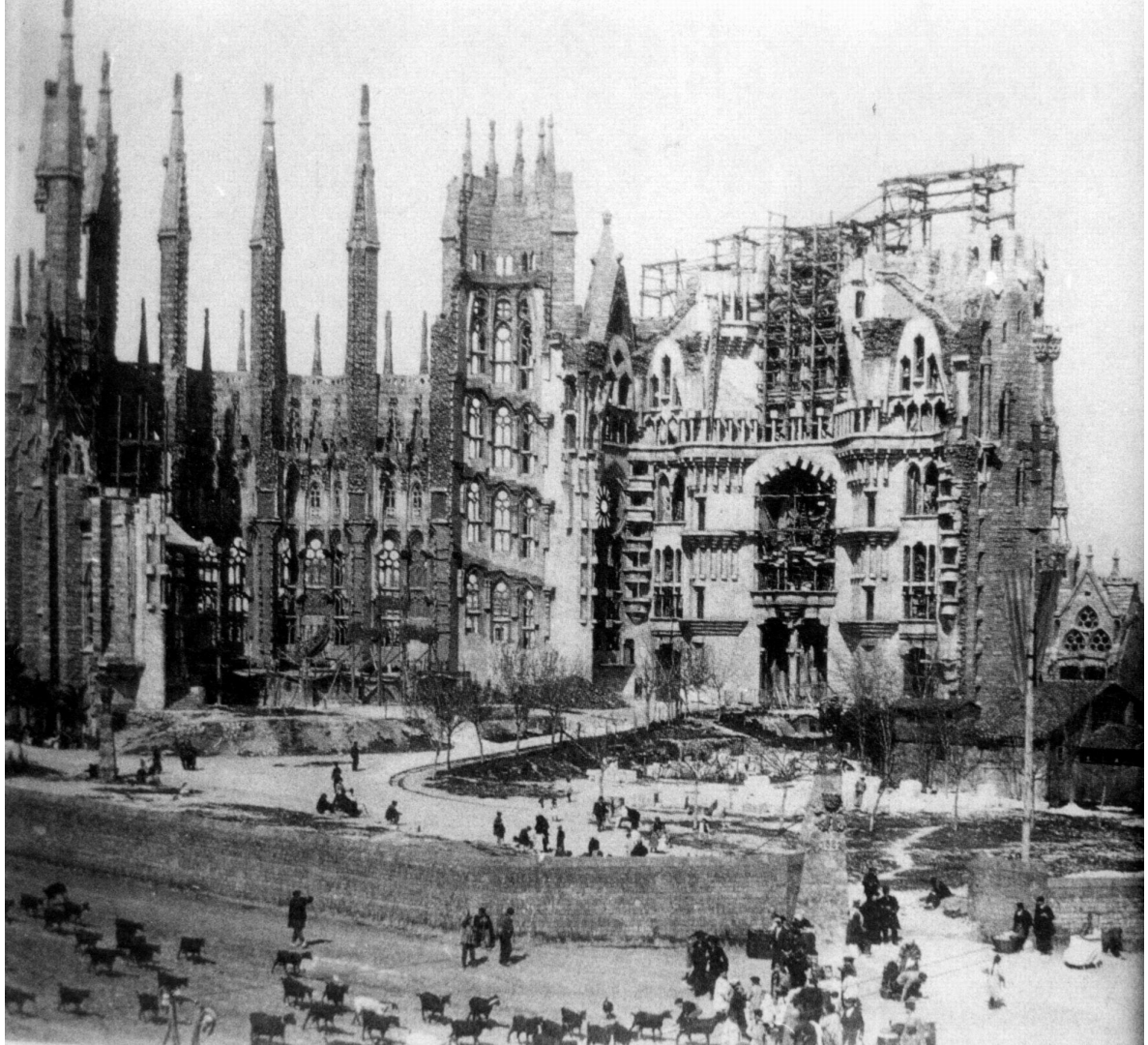
SIG-Network Deep-Dive

KubeCon EU
May 2019

Tim Hockin <thockin@google.com>
@thockin



Agenda

Ingress V1: The path to GA

DNS per-node cache

Service topology

IPv6 and dual-stack

Endpoints API reboot

Ingress V1: The path to GA

Ingress has been beta for years - why do we care NOW?

The `extensions` API group is EOL, Ingress is the last one.

Why bother? Don't we plan to replace Ingress entirely?

Yes, but that will take a long time. In the meantime, users are nervous. Why do you hate our users?

Reality check

The `extensions` API group needs to go, and we're holding up that effort

Perpetual "beta" status makes some (large) users very uncomfortable

- Is is supported? Trustworthy? Will it disappear any day now?

Ingress has trillions* of users and about a dozen implementations

Let's call it what it is - a **fully supported** Kubernetes API

Implication: it's not "going away" without standard deprecation -- O(years)

* Not really, but it makes a great slide

Things we propose to fix

Ingress.spec.rules[*].http.paths[*].path

Currently specified as a POSIX regex

- But most implementations do not support regex

Result: implementation-specific meaning

- Users have to understand the details of the implementation to use it

Proposed: Change it to a simpler user choice: exact match or prefix match

More than just a spec change -- API round-trip compatibility is required!

Details of API are TBD

Ingress.spec.rules[*].host

Current spec does not allow any wildcards

- e.g. “*.example.com”

Most platforms support some form of wildcard already

Proposal: allow one wildcard as the first segment of a host

Implementations must map that to their own details

Ingress.spec.backend

This is what happens when nothing else matches (hosts and paths)

A confusingly bland name

A simple rename should be enough

Proposal: `defaultBackend`

IngressClass

If you have 2 Ingress controllers in one cluster: what happens?

- E.g. GCE and Nginx

Ad hoc design emerged: annotate `kubernetes.io/ingress.class`

Ingresses needs to specify class via annotation, default behavior is unclear

No way to enumerate the options

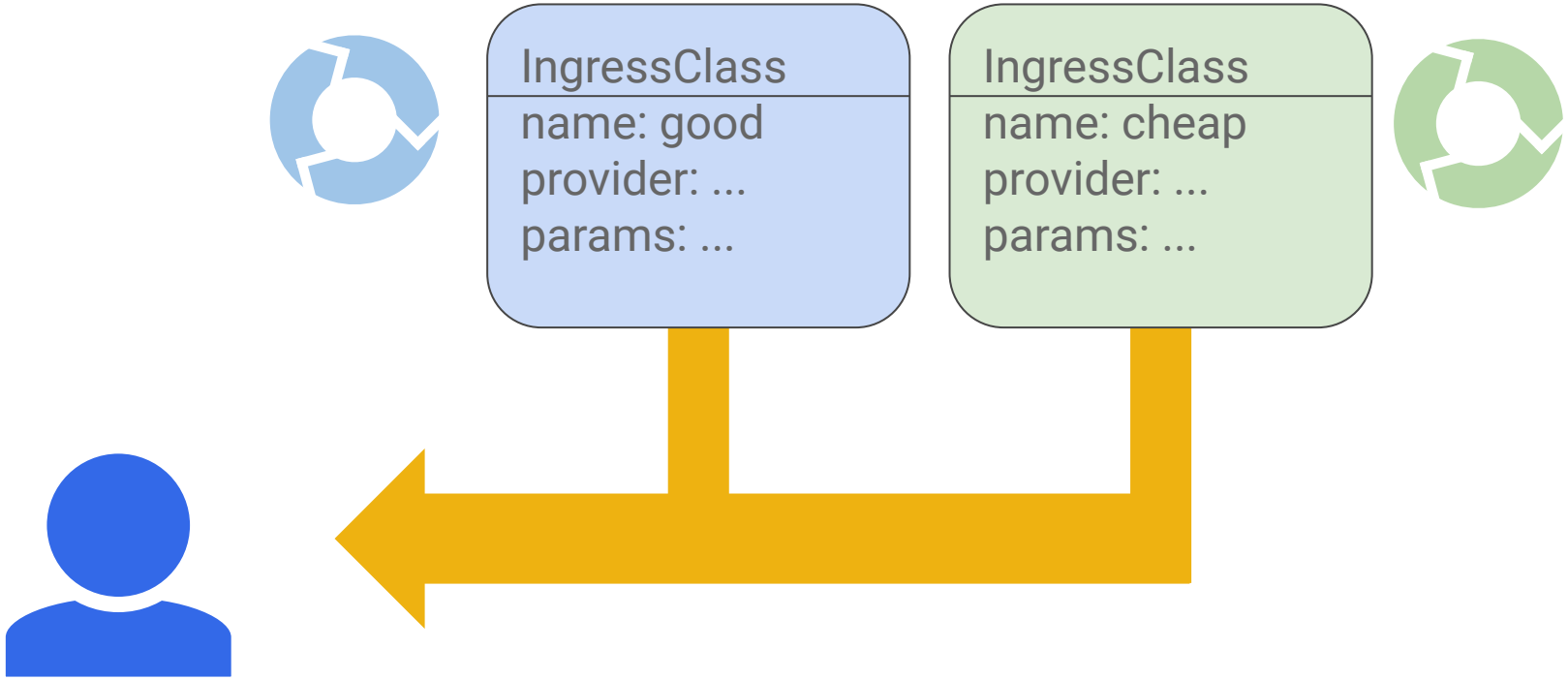
Proposal: formalize `IngressClass` a la `StorageClass`

Details of API are TBD

IngressClass
name: good
provider: ...
params: ...

IngressClass
name: cheap
provider: ...
params: ...

kubectl get IngressClasses

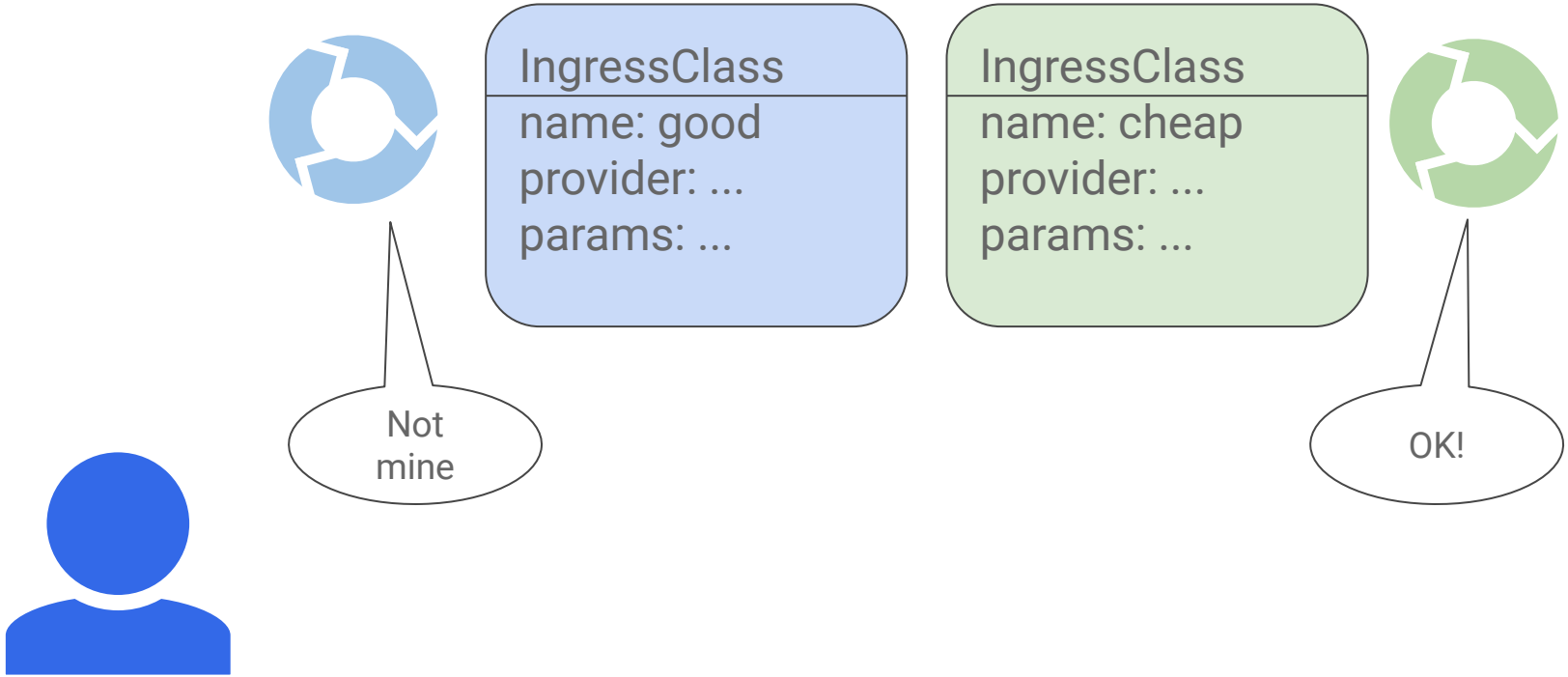

IngressClass
name: good
provider: ...
params: ...
IngressClass
name: cheap
provider: ...
params: ...

IngressClass
name: good
provider: ...
params: ...

IngressClass
name: cheap
provider: ...
params: ...

kubectl create -f ing.yaml

```
Kind: Ingress
apiVersion: networking/v1
metadata:
  name: my-cheap-ingress
spec:
  className: cheap
  defaultBackend:
    serviceName: my-service
```


IngressClass
name: good
provider: ...
params: ...
IngressClass
name: cheap
provider: ...
params: ...
Not mine
OK!

Ingress.status

Ingress says almost nothing about what is happening or why

Some implementations use events, but not consistently

Events are not a good programmatic API, anyway

Proposal: add `status` with more details

Several options:

1. Try to generalize / abstract implementation-specific status
2. TypedObjectReference to an implementation-specific CRD
3. TypedObjectReference to a suck-typed implementation-specific CRD
4. map[string]string

Healthchecks

Most cloud LBs require a health check to be specified

Implementations do different things

- assume /
- look at Pods behind Service for `readinessProbe`

All of them are imperfect

Many reports of confused users and broken LBs

Proposal: add support for specifying some very basic healthcheck params

Non-Service backends

Today a {host, path} resolves to a Service

- Service is assumed to be the only type of backend

Some LBs have other things they can front (e.g. storage buckets)

Proposal: add an option for typed references (e.g. to CRDs)

Implementations can follow those if they understand them

- Ignore or error otherwise

Details of the API are TBD

Things we can fix later

TODO(later)

A way to specify “only HTTPS”

- Some people want to never receive bare HTTP

A way to restrict hostnames per-namespace

- You can use “*.test.example.com” but not “example.com”

Per-backend timeouts

- A change to Service rather than Ingress?

Backend protocol as HTTPS

- Some annotations exist
- A change to Service rather than Ingress?

Things that are hard to fix

TODO(probably never) - Ingress v2?

Protocol upgrades HTTP->HTTPS

- Too many implementations do not support it

Cross-namespace secrets

- Maybe a simpler model for shared secrets

Backend affinity

- Too many divergent implementations

Optional features

- Would need deep API overhaul

Explicit sharing or non-sharing of IPs

- Wants API broken into multiple concepts

TCP support

- Not much demand to date

Status

KEP: http://bit.ly/kep-net-ingress-api-group

Many API details need to be fleshed out

- Round-trip requirement
- Full compatibility
- Scope

Could benefit from implementor and user input

DNS per-node cache

Problems

UDP is lossy

Conntrack for UDP is ugly (must time-out records)

UDP-conntrack bugs in kernel make the above worse

Users have to be aware of scaling for in-cluster DNS servers

- Even with auto-scaling, users often want/need to tweak params

Upstream DNS quotas

- E.g. per-IP throttling hits kube-dns when upstreaming requests

Goals

Lower-latency

- High cache hit rate

No conntrack

- No one-use records, no kernel bugs

TCP to upstreams

- No dropped packets and time-outs

Distribute load and upstream-origins

- Don’t make cluster DNS servers be hot-spots

Design

A per-node cache DaemonSet (CoreDNS with a small cache)

Create a dummy interface and IP on each node

Pass that IP to Pods via kubelet

NOTRACK that IP (disable conntrack)

Only upstream the cluster domain to cluster DNS

- otherwise use the Node's DNS config

Always upstream via TCP

Status

KEP: http://bit.ly/kep-net-nodelocal-dns-cache

Alpha now, moving to Beta with minimal changes

Results look great so far

- Lower latency
- Fewer conntrack entries used
- Node-level DNS metrics
- Less lost queries
- Better upstream load-spreading
- Lower load on cluster-scope servers

Future

Thinking about HA

- Node-agents are always somewhat dangerous
- Node is a failure domain, so HA may not be reasonable
- Looking at options for people who care but off for people who don’t

Offload search-path expansion to the cache and cluster DNS servers (autopath)

- How to do this transparently
- How to allow future schema changes

Service Topology

Problems

Need to access a service backend on the same node as yourself

- E.g. loggers or other per-node agents

Need to keep traffic in-zone whenever possible

- Manage latency
- Cross-zone traffic is chargeable by cloud-providers

Maintain failure domains

- If this rack dies, it has minimal impact on other racks

Design

Add a field to Service:

- `topologyKeys []string`

A strictly ordered list of label keys

- Compare the value of the label on “this” node with value on endpoint node
- Only consider a key if all previous keys have zero matches
- Wildcard for “don’t care”

```
Kind: Service
apiVersion: v1
metadata:
  name: my-services
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - port: 80
  topologyKeys:
    - kubernetes.io/hostname
    - topology.kubernetes.io/zone
    - “*”
```

First look for endpoints that are on a node with the same hostname as me

```
Kind: Service
apiVersion: v1
metadata:
  name: my-services
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - port: 80
  topologyKeys:
    - kubernetes.io/hostname
    - topology.kubernetes.io/zone
    - “*”
```

First look for endpoints that are on a node with the same hostname as me

If there are none, then look for endpoints that are on a node in the same zone as me

```
Kind: Service
apiVersion: v1
metadata:
  name: my-services
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - port: 80
  topologyKeys:
    - kubernetes.io/hostname
    - topology.kubernetes.io/zone
    - “*”
```

First look for endpoints that are on a node with the same hostname as me

If there are none, then look for endpoints that are on a node in the same zone as me

If there are none, pick any endpoint

Design

Every kube-proxy needs to map endpoints -> Node labels

- Could watch all Nodes
  - Map endpoint NodeName -> Node labels
  - Expensive (node is big and churns a lot!)
  - OK for alpha, need to pre-cook a new object or add metadata-only watch past that
- Can introduce a new `PodLocator` resource
  - Map Pod name -> Pod IPs and Node metadata
  - May also be needed for DNS (see next)

Headless services: Need DNS to return IPs that match caller's topology

- DNS doesn't get a NodeName with lookups
- Map pod IP back to Nodes
- Interaction with per-node cache

Status

Design mostly agreed upon for Alpha

- Some nuanced design points TBD

KEP: http://bit.ly/kep-net-service-topology

Some PRs started, but work has stalled

Help wanted!

Dual-stack (IPv6)

Background

We have had IPv6 single-stack support for a while

Stuck in alpha because it needed CI and contributors became unavailable

- Made worse because cloud support for IPv6 is weak/inconsistent/missing

Dual-stack is really the goal

- But much, MUCH more complicated

New plan - do dual-stack and CI that, instead

News

New hands picking it up (thanks @lachie83 and @khenidak)

KEP: http://bit.ly/kep-net-ipv4-ipv6-dual-stack

HUGE effort -- phasing the development

Touches most binaries, many API types, and sets significant API precedents

Phase 1

Make IP-related fields and flags be plural and dual-stack ready

Hard because of API compatibility and round-trip requirements

- Had to establish a new convention for pluralizing fields compatibly

LOTS of fields and flags to process

Make Pods be dual-stack ready

- CNI, etc

Make HostPorts be dual-stack ready

- Run iptables and ip6tables in parallel

Phase 2

Make Endpoints support multiple IPs

Make DNS support A and AAAA for headless Services

Make NodePorts dual-stack

Adapt Ingress controller(s)

Make Pod probes dual-stack

Dual-stack Service VIPs TBD?

Endpoints API reboot

Problem

Endpoints as a monolithic object is hitting scalability problems

- Can get very large
- Running into etcd limits
- Causing pain for apiservers

A rolling update of a 1000 replica service in a 1000 node cluster sends over 250 GB of network traffic from the apiserver!

Way too much serialize/deserialize happening

Proposal: a new API

Proposal: http://bit.ly/sig-net-endpoint-slice-doc

Chop monolithic Endpoints up into smaller “slices”

- Use a selector to join them

Default 100 Endpoints per slice (tunable but not part of API)

- Most services are < 100 Endpoints

Trying to balance number of writes vs total size of writes & watch events

Replaces Endpoints for in-project users (e.g. kube-proxy)

- Keep old API until “core” group is removed

Status: KEP soon

Conclusion

We have a LOT going on

This is not even everything!

We also have bugs and smaller things that need doing

Help wanted!