Kubernetes the Very Hard Way

Laurent Bernaille
Staff Engineer, Infrastructure
@lbernail
Datadog

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 k8s clusters with 50-2500 nodes
Multi-cloud
Very fast growth
Why Kubernetes?

Dogfooding

- Improve k8s integrations

Immutable

- Move from Chef

Multi Cloud

- Community

Common API

- Large and Dynamic
The very hard way?

Bootstrap Kubernetes the hard way on Google Cloud Platform. No scripts.
It was much harder
This talk is about the fine print

“Of course, you will need a HA master setup”

“Oh, and yes, you will have to manage your certificates”

“By the way, networking is slightly more complicated, look into CNI / ingress controllers”
What happens after “Kube 101”

1. Resilient and Scalable Control Plane
2. Securing the Control Plane
   a. Kubernetes and Certificates
   b. Exceptions?
   c. Impact of Certificate Rotation
3. Efficient networking
   a. Giving pod IPs and routing them
   b. Ingresses: Getting data in the cluster
What happens after “Kube 101”

1. Resilient and Scalable Control Plane
2. Securing the Control Plane
   a. Kubernetes and Certificates
   b. Exceptions?
   c. Impact of Certificate Rotation
3. Efficient networking
   a. Giving pod IPs and routing them
   b. Ingresses: Getting data in the cluster
Resilient and Scalable Control Plane
Kube 101 Control Plane

- **Master**
  - etcd
  - apiserver
    - scheduler
    - controllers
  - Service
    - in-cluster apps
    - kubectl
    - kubelet

- **In-Cluster**
  - apps
Making it resilient

- etcd
- apiserver
- scheduler
- controllers
- Service
  - in-cluster apps
- LoadBalancer
  - kubelet
  - kubectl

Master

kubelet

kubectl
Kube 101 Control Plane

master

apiserver

etcd

scheduler

collectors

Service

in-cluster apps

cubelet

cubectl
Separate etcd nodes
Single active Controller/scheduler

- etcd
- apiserver
- controllers
- scheduler
- Master
- Service
- LoadBalancer
- in-cluster apps
- kubelet
- kubectl
Split scheduler/controllers
Split etcd

[Diagram showing the split of etcd and its components, including APIserver, etcd, etcd events, Service, LoadBalancer, in-cluster apps, kubelet, kubectl, controllers, and schedulers.]
Sizing the control plane
What happens after “Kube 101”

1. Resilient and Scalable Control Plane
2. Securing the Control Plane
   a. Kubernetes and Certificates
   b. Exceptions?
   c. Impact of Certificate Rotation
3. Efficient networking
   a. Giving pod IPs and routing them
   b. Ingresses: Getting data in the cluster
Kubernetes and Certificates
From “the hard way”

cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": ["signing", "key encipherment", "server auth", "client auth"],
        "expiry": "8760h"
      }
    }
  }
}
EOF
“Our cluster broke after ~1y”

cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": ["signing", "key encipherment", "server auth", "client auth"],
        "expiry": "8760h"
      }
    }
  }
}
EOF
Certificates in Kubernetes

- Kubernetes uses certificates everywhere
- Very common source of incidents
- Our Strategy: Rotate all certificates daily
Certificate management

- etcd
- apiserver
- etcd PKI
- Vault

Connections:
- Peer/Server cert from etcd to etcd PKI
- Etcd Client cert from apiserver to etcd PKI
Certificate management

- **etcd**
  - Peer/Server cert
- **apiserver**
  - Etd Client cert
  - Apiserver/kubelet client cert
- **controllers**
- **scheduler**
- **kubelet**
  - Controller client cert
  - Scheduler client cert
  - Kubelet client/server cert

Vault
Certificate management

- etcd
  - etcd PKI
  - Peer/Server cert
  - Vault
- apiserver
  - etcd
    - etcd Client cert
  - kube PKI
    - kube kv
      - SA public key
      - Controller client cert
      - Scheduler client cert
      - Kubelet client/server cert
- controllers
- scheduler
  - SA token
- kubelet
- In-cluster app
  - SA token
  - kubelet
  - Apiserver/kubelet client cert
    - SA private key
Certificate management

- **etcd**
- **apiserver**
- **controllers**
- **scheduler**
- **kubelet**
- **apiservice**
- **Vault**

**etcd PKI**: Peer/Server cert

**apiservice PKI**: Apiservice cert (proxy/webhooks)

**kube PKI**: Apiservice cert (proxy/webhooks)

**kube kv**: SA public key

**In-cluster app**: SA token

- **apiservice webhook**
- **kubelet**
- **Controller client cert**
- **Scheduler client cert**
- **Kubelet client/server cert**
Certificate management

- etcd
- etcd PKI
- apiservice PKI
- kube PKI
- kubectl
- OIDC provider
- OIDC auth
- SA token
- In-cluster app
- apiservice webhooks...
- controllers
- scheduler
- kubelet
- Peer/Server cert
- Etcd Client cert
-Apiservice cert (proxy/webhooks)
-Apiserver/kubelet client cert
-SA public key
-SA private key
-Controller client cert
-Scheduler client cert
-Kubelet client/server cert
Kubelet: TLS Bootstrap

1. Create Bootstrap token
2. Add Bootstrap token to vault
3. Get signing key

Vault

apiserver
controllers
kube PKI
kube kv
admin
Kubelet: TLS Bootstrap

1- Get Bootstrap token
2- Authenticate with token
4- Create CSR
7- Download certificate
8- Authenticate with cert
9- Register node

3- Verify Token and map groups
5- Verify RBAC for CSR creator
6- Sign certificate

Vault

apiserver
controllers
kubelet
kube PKI
kube kv
Kubelet certificate issue

1. One day, some Kubelets were failing to start or took 10s of minutes
2. Nothing in logs
3. Everything looked good but they could not get a cert
4. Turns out we had a lot of CSRs in flight
5. Signing controller was having a hard time evaluating them all

CSR resources in the cluster

Lower is better!
# Why?

## Kubelet Authentication
- Initial creation: bootstrap token, mapped to group "system:bootstrappers"
- Renewal: use current node certificate, mapped to group "system:nodes"

## Required RBAC permissions
- CSR creation
- CSR auto-approval

<table>
<thead>
<tr>
<th></th>
<th>CSR creation</th>
<th>CSR auto-approval</th>
</tr>
</thead>
<tbody>
<tr>
<td>system:bootstrappers</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>system:nodes</td>
<td>OK</td>
<td>❌</td>
</tr>
</tbody>
</table>
Exception 2?
Incident 2...
Temporary solution

One day, after ~1 year

- Creation of resources started failing (luckily only a Custom Resource)
- Cert had expired...
Take-away

- Rotate server/client certificates
- Not easy

But, “If it’s hard, do it often”

> no expiration issues anymore
Impact of Certificate rotation
Apiserver certificate rotation
Impact on etcd

apiserver restarts

We have multiple apiservers
We restart each daily

etcd traffic

Significant etcd network impact
(caches are repopulated)

etcd slow queries

Significant impact on etcd performances
Impact on Load-balancers

apiserver restarts

Significant impact on LB as connections are reestablished

Mitigation: increase queues on apiservers
net.ipv4.tcp_max_syn_backlog
net.core.somaxconn
Impact on apiserver clients

- Apiserver restarts
- Clients reconnect and refresh their cache

> Memory spike for impacted apps

No real mitigation today
Impact on traffic balance

Number of connections / traffic very unbalanced
Because connections are very long-lived

More clients => Bigger impact clusterwide

15MB/s
2.5MB/s

2300 connections
300 connections
Why? Simple simulation

Simulation for 48h
- 5 apiservers
- 10000 connections (4 x 2500 nodes)
- Every 4h, one apiserver restarts
- Reconnections evenly dispatched

Cause
- Cloud TCP load-balancers use round-robin
- Long-lived connections
- No rebalancing
Kubelet certificate rotation
Pod graceful termination

admin or controller ➔ Delete pod ➔ apiserver ➔ Stop Container with timeout "terminationGracePeriodSeconds" ➔ kubelet ➔ containerd ➔ Send SIGTERM After timeout, send SIGKILL ➔ container
Restarts impact graceful termination

Kubelet restarts end graceful termination
Fixed upstream
“Do not SIGKILL container if container stop is cancelled”
https://github.com/containerd/cri/pull/1099
Impact on pod readiness

kubelet restarts on “system” nodes (coredns + other services)

On kubelet restart
- Readiness probes marked as failed
- Pods removed from service endpoints
- Requires readiness to succeed again

Issue upstream
“pod with readinessProbe will be not ready when kubelet restart”
https://github.com/kubernetes/kubernetes/issues/78733

coredns endpoints NotReady

Graph showing time stamps from 07:30 to 18:30 with marked periods indicating kubelet restarts and coredns endpoints NotReady.
Take-away

Restarting components is not transparent

It would be great if

○ Components could transparently reload certs (server & client)
○ Clients could wait 0-Xs to reconnect to avoid thundering herd
○ Reconnections did not trigger memory spikes
○ Cloud TCP load-balancers supported least-conn algorithm
○ Connections were rebalanced (kill them after a while?)
What happens after “Kube 101”

1. Resilient and Scalable Control Plane
2. Securing the Control Plane
   a. Kubernetes and Certificates
   b. Exceptions?
   c. Impact of Certificate Rotation
3. Efficient networking
   a. Giving pod IPs and routing them
   b. Ingresses: Getting data in the cluster
Efficient networking
Network challenges

**Throughput**
- Trillions of data points daily

**Scale**
- 1000-2000 nodes clusters

**Latency**
- End-to-end pipeline

**Topology**
- Multiple clusters
- Access from standard VMs
Giving pods IPs & Routing them
From “the Hard Way”

Routes

Create network routes for each worker instance:

```bash
for i in 0 1 2; do
  gcloud compute routes create kubernetes-route-10-200-$i-0-24 \
    --network kubernetes-the-hard-way \
    --next-hop-address 10.240.0.2$i\n    --destination-range 10.200.$i.0/24
done
```

- **node IP**: 10.240.0.2$i
- **Pod CIDR for this node**: 10.200.$i.0/24
Small cluster? Static routes

Node 1
IP: 192.168.0.1
Pod CIDR: 10.0.1.0/24

Node 2
IP: 192.168.0.2
Pod CIDR: 10.0.2.0/24

Routes (local or cloud provider)
10.0.1.0/24 => 192.168.0.1
10.0.2.0/24 => 192.168.0.2

Limits
local: nodes must be in the same subnet
cloud provider: number of routes
Mid-size cluster? Overlay

Node 1
IP: 192.168.0.1
Pod CIDR: 10.0.1.0/24

Node 2
IP: 192.168.0.2
Pod CIDR: 10.0.2.0/24

VXLAN
Tunnel traffic between hosts
Examples: Calico, Flannel

Limits
Overhead of the overlay
Scaling route distribution (control plane)
Large cluster with a lot of traffic?
Native pod routing

<table>
<thead>
<tr>
<th>Performance</th>
<th>Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath: no overhead</td>
<td>Pod IPs are accessible from</td>
</tr>
<tr>
<td>Control plane: simpler</td>
<td>● Other clusters</td>
</tr>
<tr>
<td></td>
<td>● VMs</td>
</tr>
</tbody>
</table>
## In practice

<table>
<thead>
<tr>
<th>On premise</th>
<th>GCP</th>
<th>AWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGP</td>
<td>IP aliases</td>
<td>Additional IPs on ENIs</td>
</tr>
<tr>
<td>Calico</td>
<td></td>
<td>AWS EKS CNI plugin</td>
</tr>
<tr>
<td>Kube-router</td>
<td></td>
<td>Lyft CNI plugin</td>
</tr>
<tr>
<td>Macvlan</td>
<td></td>
<td>Cilium ENI IPAM</td>
</tr>
</tbody>
</table>
How it works on AWS

AWS

Attach ENI
Allocate IPs

agent

kubelet
containerd

CRI
CNI

cni

Create veth

Routing rule
“From IP1, use eth1”

eth1

eth0

pod 1

pod 2

Allocate IPs

Routing

“From IP1, use eth1”
Address space planning

- /24 leads to inefficient address usage
- sig-network: remove contiguous range requirement for CIDR allocation
- But also
  - Address space for node IPs (another /20 per cluster for 4096 nodes)
  - Service IP range (/20 would make sense for such a cluster)
- Total: 1 /15 for pods, 2 /20 for nodes and service!
Take-away

• Native pod routing has worked very well at scale
• A bit more complex to debug
• Much more efficient datapath
• Topic is still dynamic (Cilium introduced ENI recently)
• Great relationship with Lyft / Cilium
• Plan your address space early
Ingress: cross-clusters, VM to clusters

Cluster 1

Classic (VM)

Cluster 2

C?

B?

C?
Kubernetes default: LB service

- **Master**
  - service-controller

- **External Client**

- **LoadBalancer**

- **Healthchecker**

- **Pods**
  - kube-proxy
  - pod

Data path

Health checks

Configuration (from watching ingress on apiservers)
Inefficient Datapath & cross-application impacts

Master
  service-controller

Healthchecker

Load-Balancer
  data path

Web traffic

Load-Balancer

kube-proxy

web-1, web-2, web-3

kube-proxy

kafka

configuration (from watching ingresses on apiservers)
ExternalTrafficPolicy: Local?

- Master
  - service-controller

- Web traffic
  - Load-Balancer
    - Healthchecker

- Load-Balancer
  - NP kube-proxy
  - web-1
  - NP kube-proxy
  - web-2
  - NP kube-proxy
  - web-3
  - NP kube-proxy
  - kafka

- data path
- health checks
- configuration (from watching ingresses on apiservers)
L7-proxy ingress controller

Create l7proxy deployments
Update backends using service endpoints

Data path from watching ingresses/endpoints on apiservers (ingress-controller)
from watching LoadBalancer services (service-controller)
Challenges

Limits

- All nodes as backends (1000+)
- Inefficient datapath
- Cross-application impacts

Alternatives?

- ExternalTrafficPolicy: Local?
  - Number of nodes remains the same
  - Issues with some CNI plugins
- K8s ingress
  - Still load-balancer based
  - Need to scale ingress pods
  - Still inefficient datapath
Our target: native routing

- **alb-ingress-controller**
- **External Client**
- **Healthchecker**
- **ALB**
- pods

Data path and health checks configuration (from watching ingresses/endpoints on apiservers)
Remaining challenges

Limited to HTTP ingresses

- No support for TCP/UDP
- Ingress v2 should address this

Registration delay

- Slow registration with LB
- Pod rolling-updates much faster

Mitigations
- MinReadySeconds
- Pod ReadinessGates
Workaround

TCP / Registration delay not manageable

> Dedicated gateways

- **External Client**
  - Load-Balancer
    - Heathchecker
    - Not managed by k8s

- **I7proxy**
  - Dedicated nodes
  - Pods in host network

Pods in host network
Take-away

- Ingress solutions are not great at scale yet
- May require workarounds
- Definitely a very important topic for us
- The community is working on v2 Ingresses
Conclusion
A lot of other topics

- Accessing services (kube-proxy)
- DNS (it’s always DNS!)
- Challenges with Stateful applications
- How to DDOS <insert ~anything> with Daemonsets
- Node Lifecycle / Cluster Lifecycle
- Deploying applications
- ...

Ibernail
Getting started?

“Deep Dive into Kubernetes Internals for Builders and Operators”
Jérôme Petazzoni, Lisa 2019
Minimal cluster, showing interactions between main components

“Kubernetes the Hard Way”
Kelsey Hightower
https://github.com/kelseyhightower/kubernetes-the-hard-way
HA control plane with encryption
You like horror stories?

“Kubernetes the very hard way at Datadog”
https://www.youtube.com/watch?v=2dsCwp_j0yQ

“10 ways to shoot yourself in the foot with Kubernetes”
https://www.youtube.com/watch?v=QKI-JRs2RIE

“Kubernetes Failure Stories”
https://k8s.af
Key lessons

Self-managed Kubernetes is hard
> If you can, use a managed service

Networking is not easy (especially at scale)

The main challenge is not technical
> Build a team
> Transforming practices and training users is very important
Thank you

We’re hiring!
https://www.datadoghq.com/careers/

laurent@datadoghq.com
@lbernaill