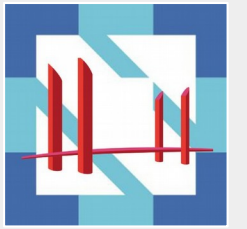
Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

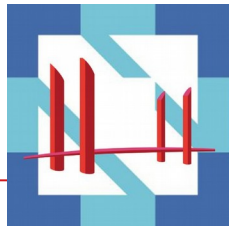


Dans Ton Kube

Retour sur 2 ans d'incidents en production

LECTRA

~# whoami



Denis GERMAIN 

- Ingénieur Cloud chez LECTRA
- Auteur principal sur https://blog.zwindler.fr 

 (perso)

@zwindler

 (blog)

@zwindler_rflx

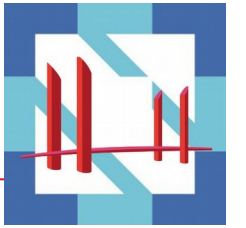
 E-mail

d.germain @lectra.com

LECTRA

Leader mondial des solutions technologiques intégrées pour les entreprises utilisatrices de cuir ou textile

Les équipes de R&D chez LECTRA

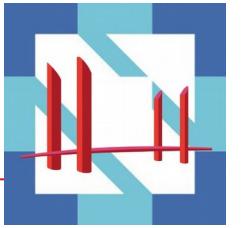


- 200 développeurs
- 22 équipes de développement logiciels
  - Services cloud (pour salle de coupe ou designers), PLM, CAO 2D/3D, ...
- Une équipe dédiée à l'intégration continue

et...

- Un équipe d'OPS pour surveiller que tout se passe bien

LECTRA recrute !!



Je cherche 2 futurs collègues

- Un Cloud Engineer **#Ops #Cloud #K8s #InfraAsCode**
- Un profil « DevSecOps » **#Sécu #MaisPasQue #OpsAussi**

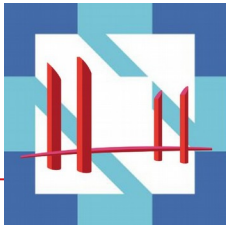
Et plein d’autres encore !!

- goto www.lectra.com/fr/carrieres

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

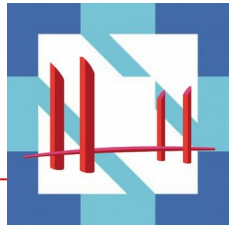
kubernetes

« *There is no cloud...* »



- En 2015, LECTRA se projette vers le cloud & le SaaS
- Deux prérequis :
  - Des ressources accessibles 24x7, worldwide
  - Une architecture µ-services, containerisée et automatisée

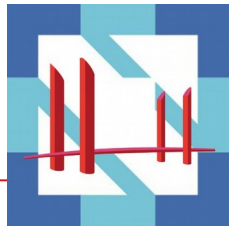
C’est quoi, kubernetes ?



- Système open source
- Historiquement dérivé des travaux de Google
- Projet donné par Google en 2015, genèse de la CNCF

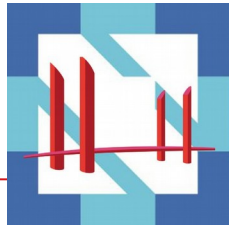
- Permet d’orchestrer (automatiser, gérer)
- Déploiement et mise à l’échelle des applications

Déployer kubernetes



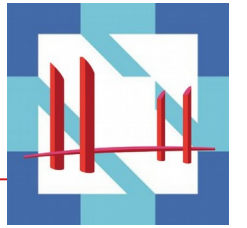
- Nombreuses façons de déployer Kubernetes
  - (documentation officielle) Picking the Right Solution
- Kubernetes the Hard way
  - Github Kelsey Hightower
- Kubernetes Managé / Kubernetes As a Service
  - GKS, EKS, AKS, ...
- Kubernetes déployé par script /outil
  - Kubeadm, Kubespray, AKS Engine, ...

kubernetes chez LECTRA



- 7 clusters indépendants
  - 5 dans Azure avec AKS-Engine
  - 2 *on-prem* avec Kubespray (Ansible)
- ~50 Nodes
- ~500 Deployments / ~1500 Containers
  - Très majoritairement Stateless
  - Le Stateful arrive doucement

kubernetes chez LECTRA

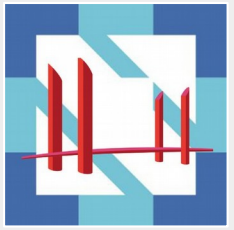


2 ans d'utilisation en développement, test et production

- Pas d'incident majeur sur K8s ayant écroulé un environnement complet
- Quelques incidents, aux impacts plus ou moins importants

D'où l'idée d'en faire un REX

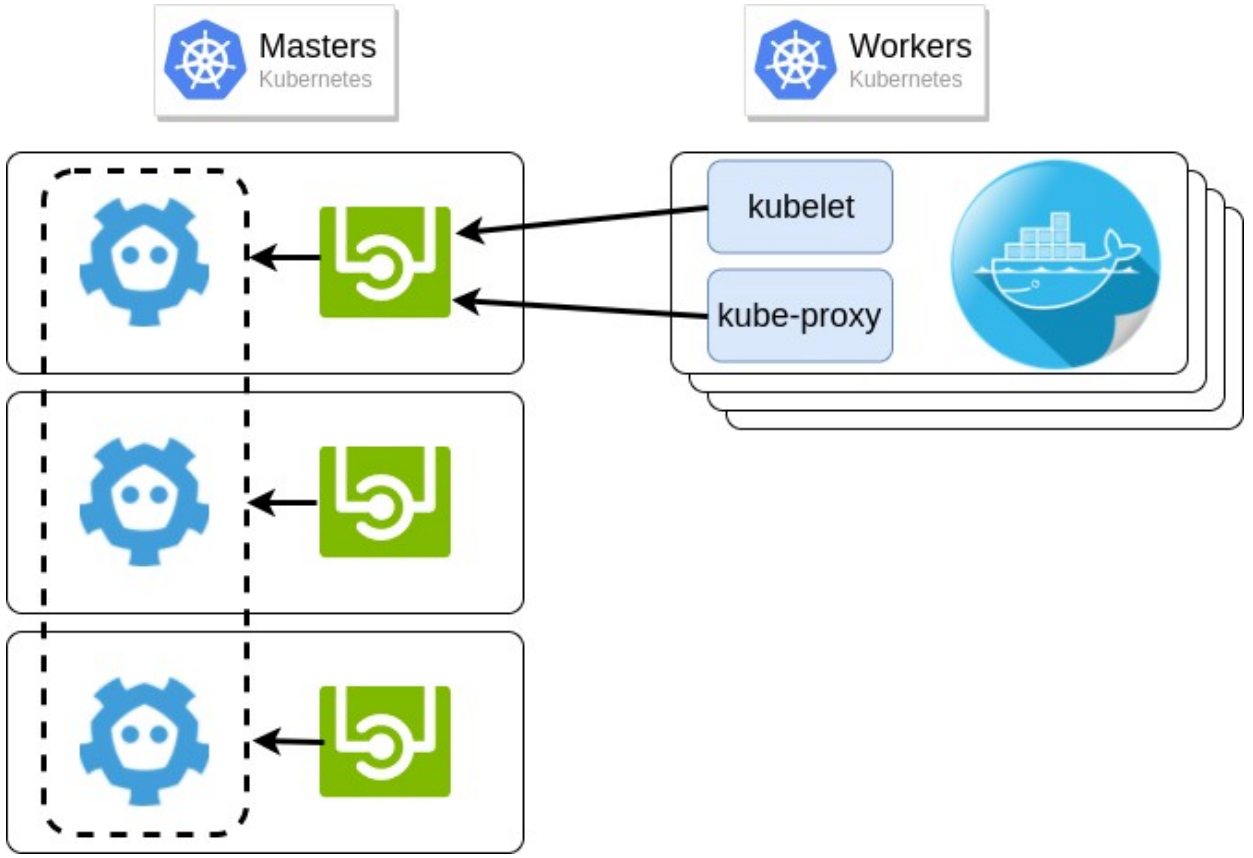
Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

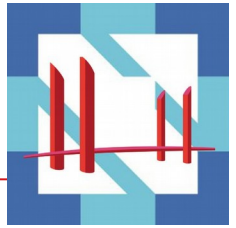


etcd

Base de données distribuée « clé - valeur »

- « Consistance » prioritaire « Disponibilité »
- Utilisé par Kubernetes pour décrire l’état du cluster





Deux versions d’**etcd** sont majoritairement utilisées

- Version 2, obsolète
- Version 3, actuelle

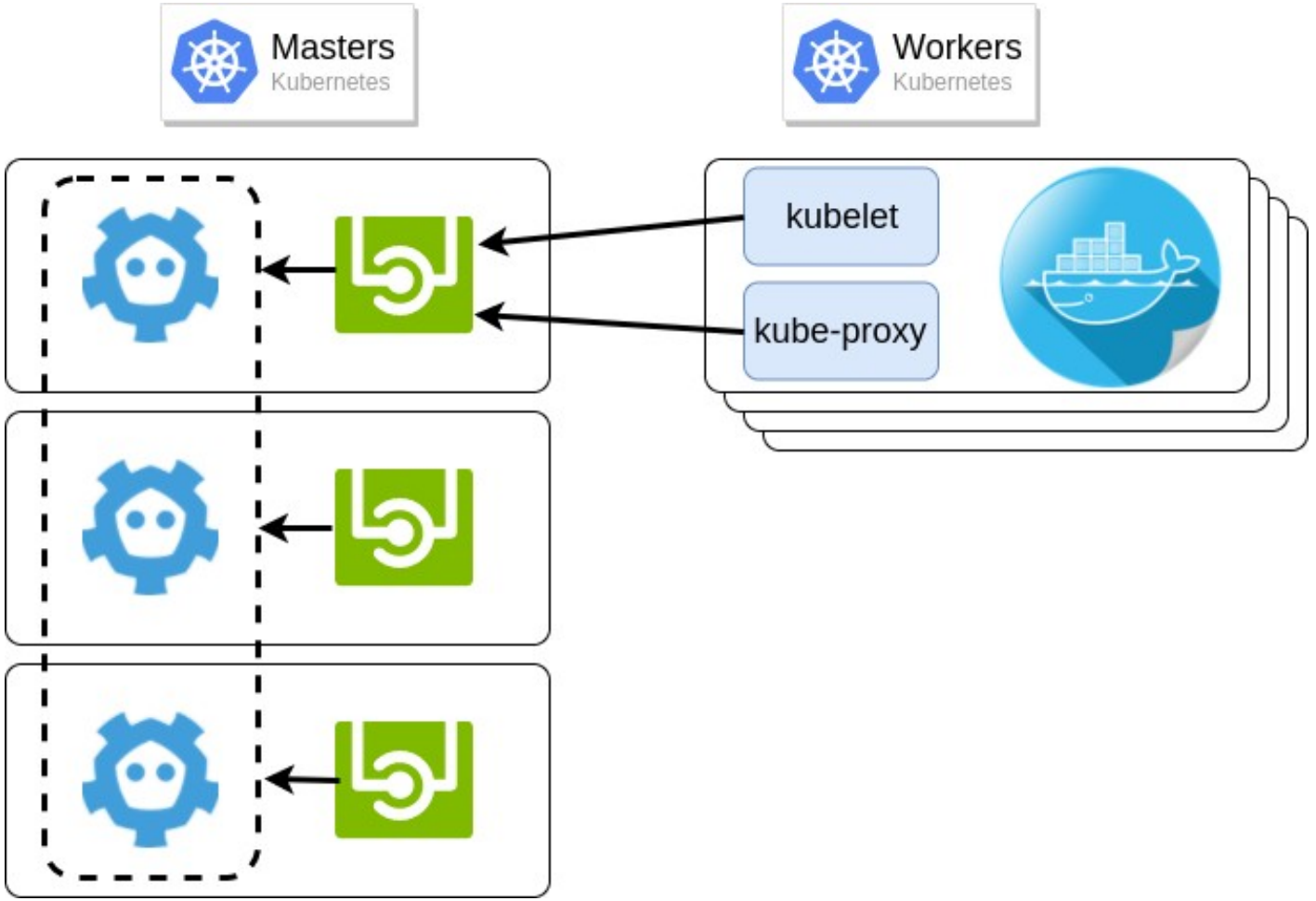
AKS engine utilisait la version 2 jusqu’à mi 2018

Au bout de quelques mois, **etcd** consomme

- 500 IOPS par master
- un CPU à 100 % en I/O wait

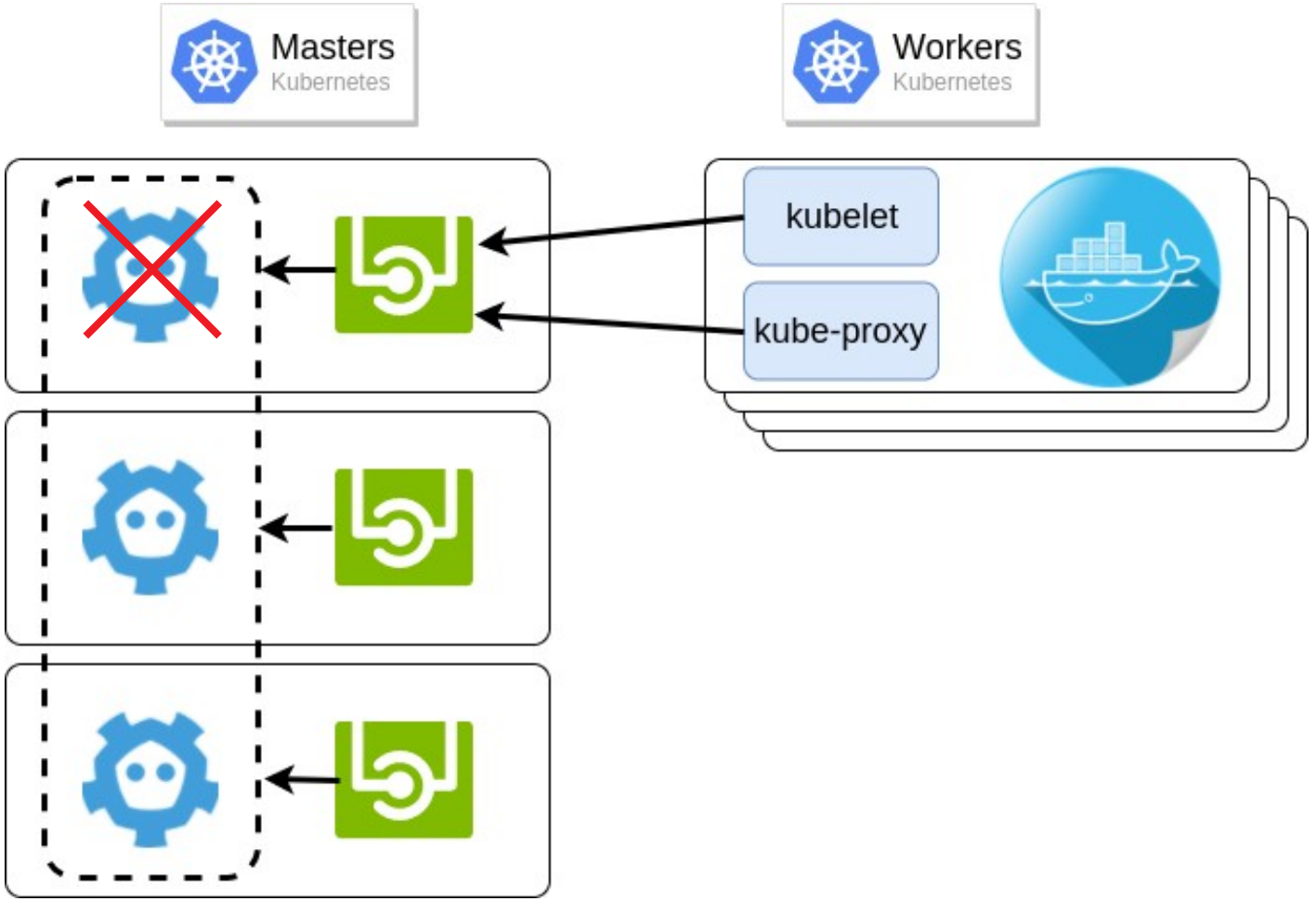
etcd

100 % I/O wait ? Bof… c’est sûrement une erreur normale !



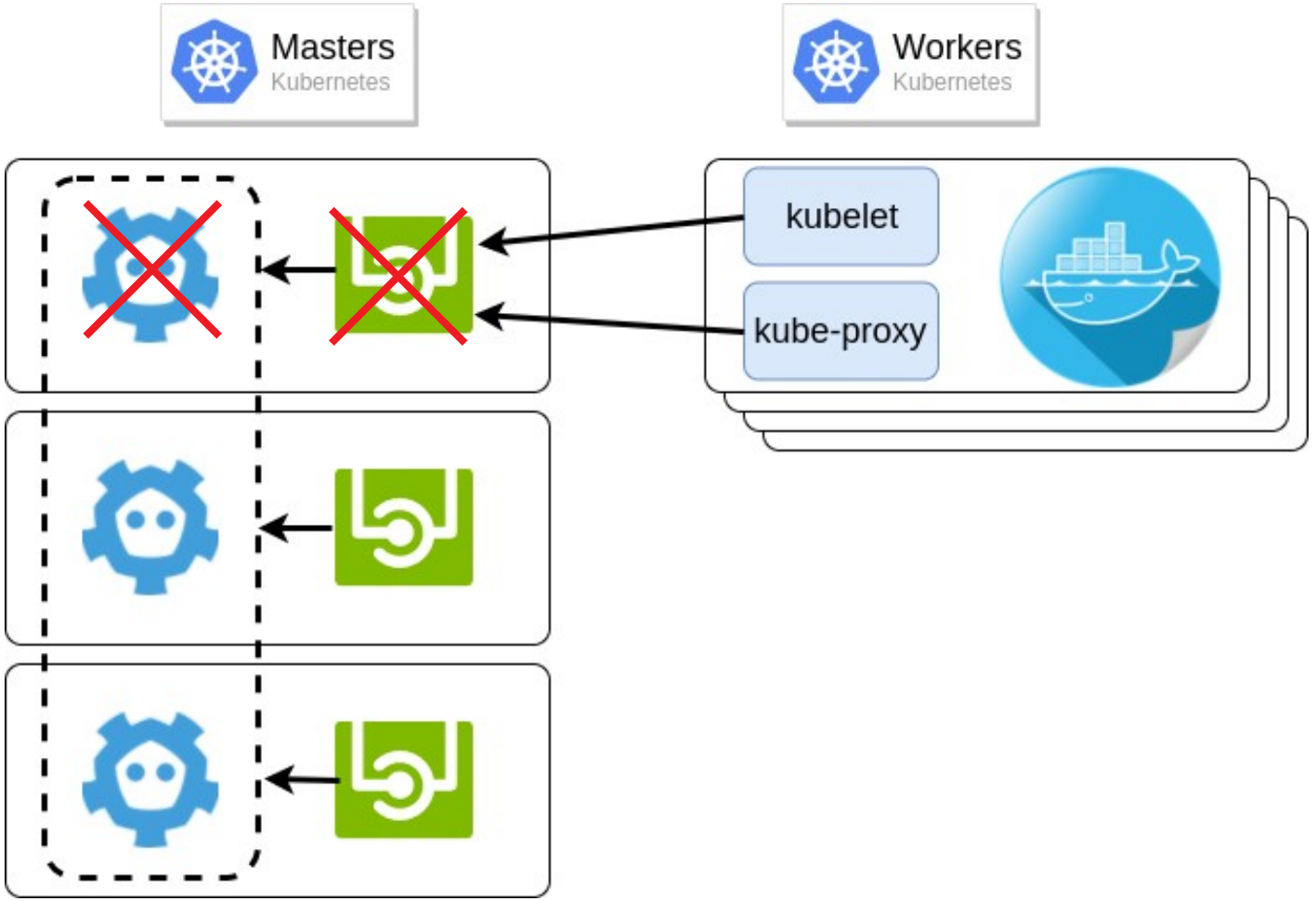
etcd

100 % I/O wait ? Bof… c’est sûrement une erreur normale !



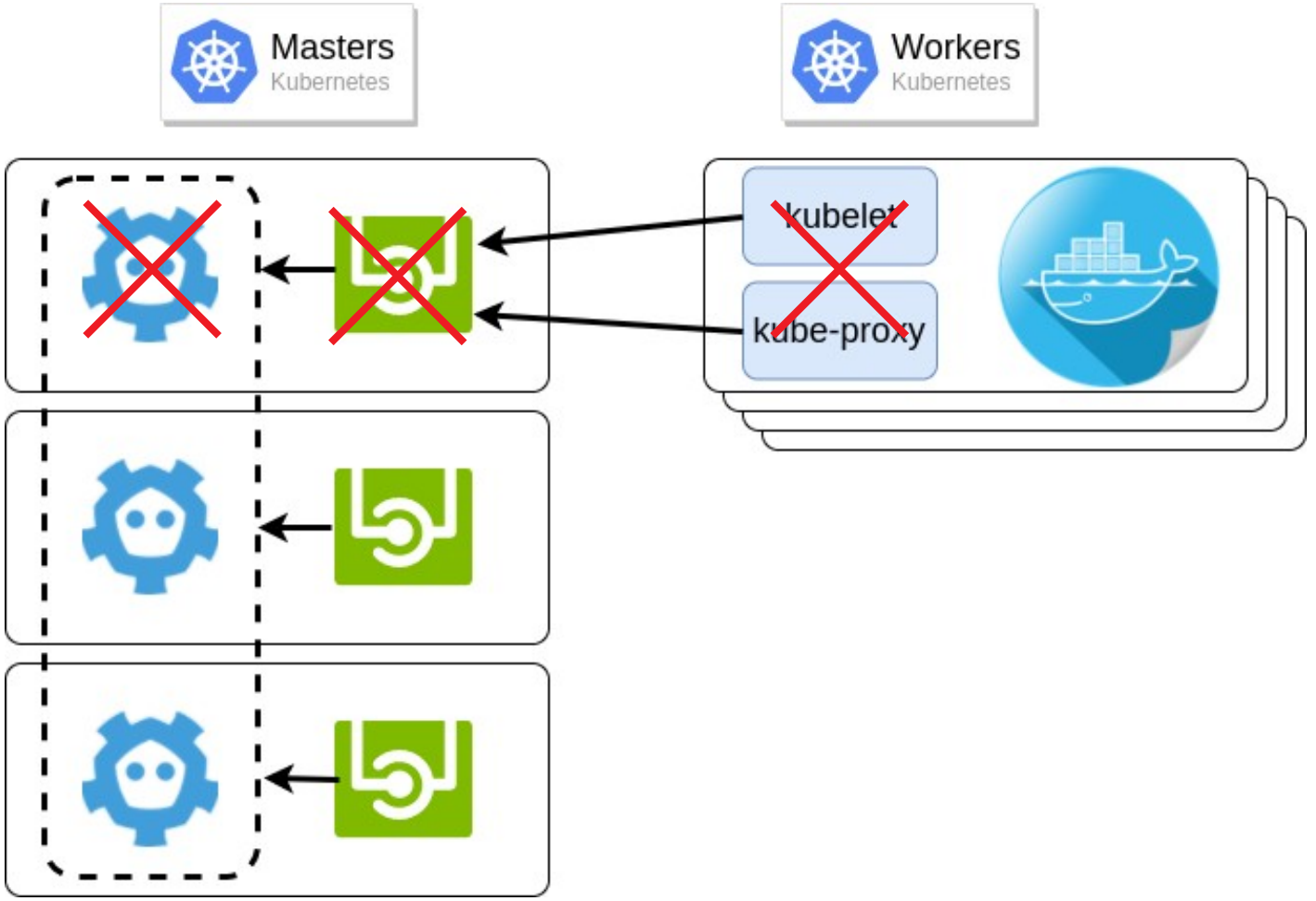
etcd

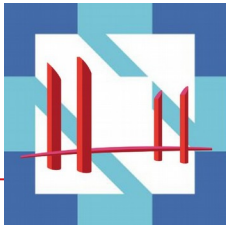
100 % I/O wait ? Bof… c’est sûrement une erreur normale !



etcd

100 % I/O wait ? Bof… c’est sûrement une erreur normale !





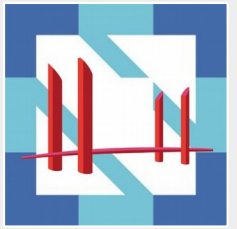
La solution ?

- Pas d’autre choix que de passer en v3

Attention à la migration !!

- Une bonne partie des portmortems d’incidents graves sur Kubernetes impliquent 

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com



Haute disponibilité

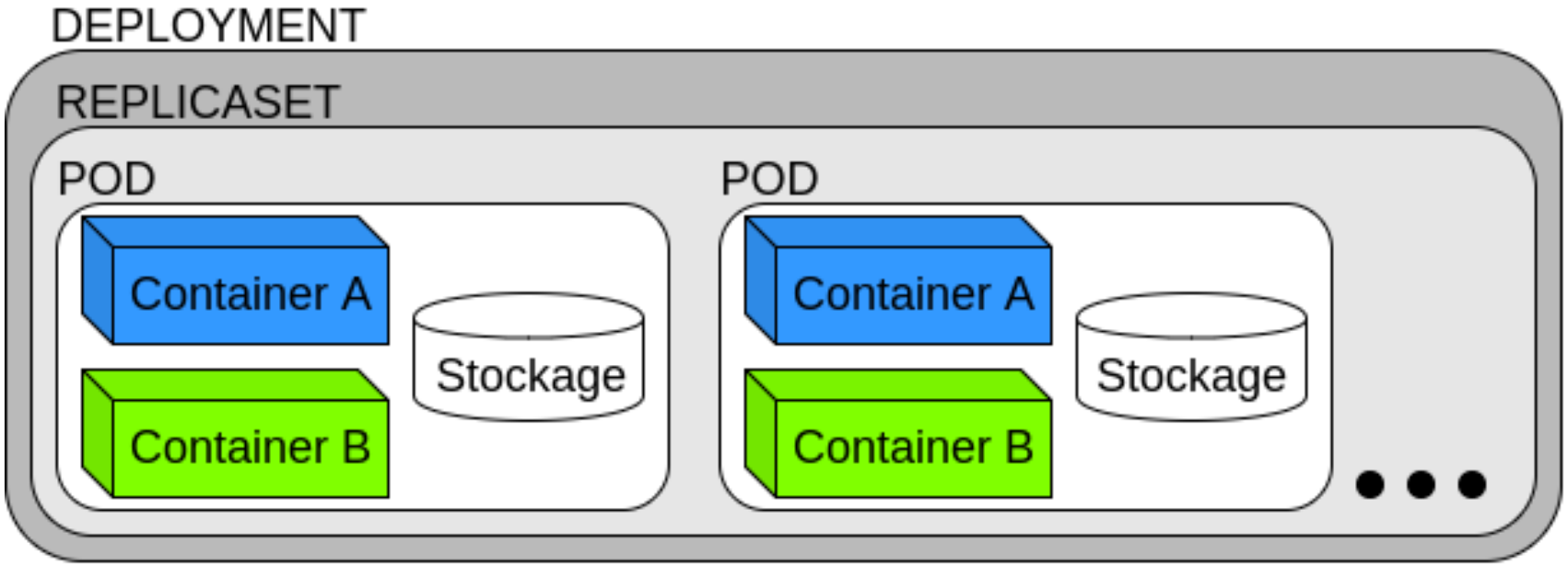
Déploiements et Pods

Dans Kubernetes, l'unité de base est le Pod

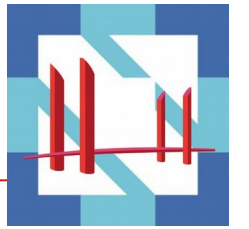
- Un ou plusieurs containers / « application atomique »

La mise à l'échelle d'un Pod est gérée via un ReplicaSet

Les applications sont installées via les Déploiements

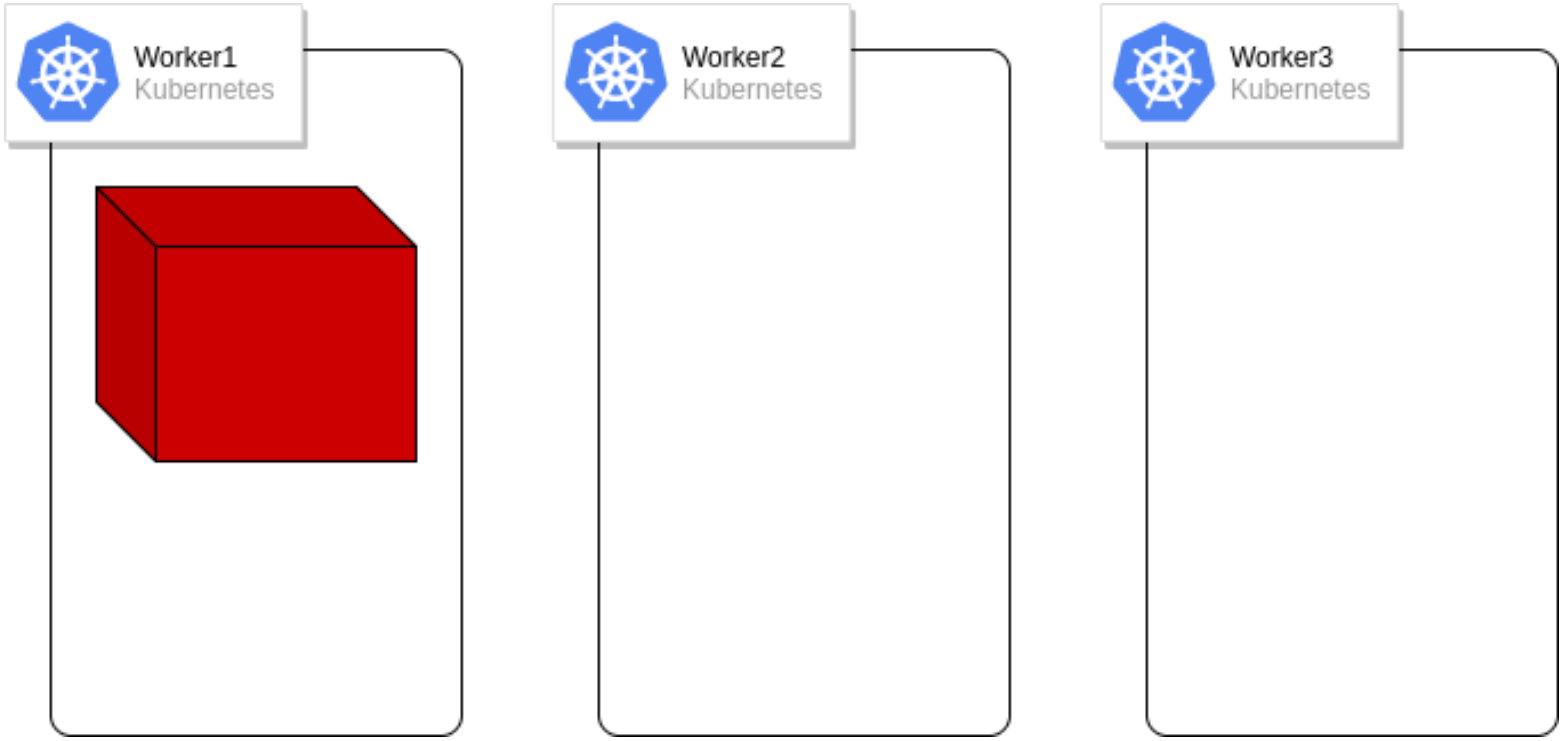


Affinity et AntiAffinity

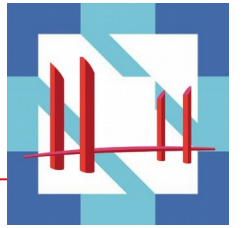


Par défaut, Kubernetes réparti les Pods

… parfois pas de manière optimale (en disponibilité) !

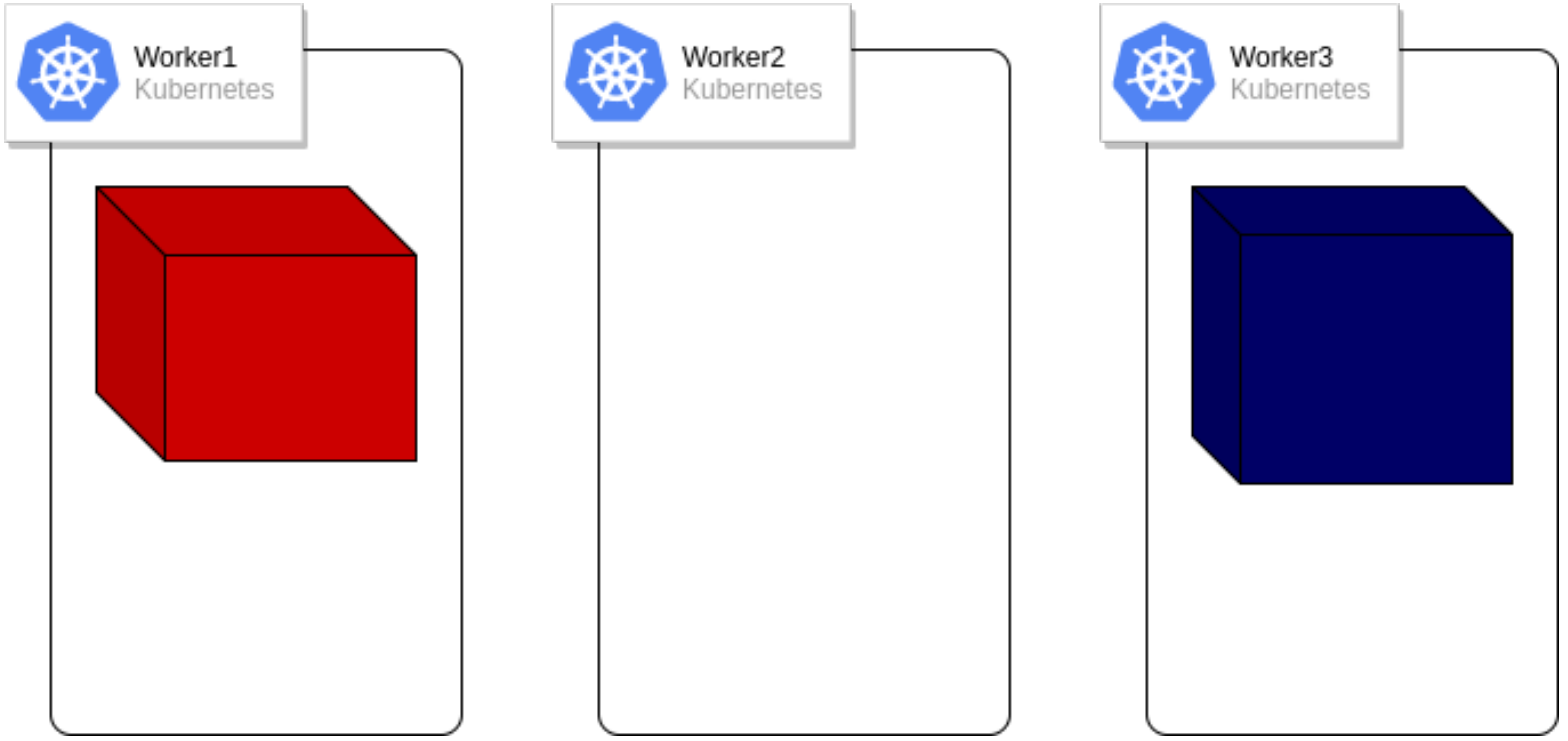


Affinity et AntiAffinity

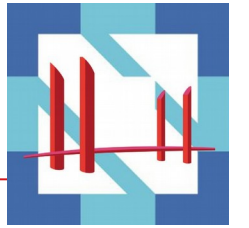


Par défaut, Kubernetes réparti les Pods

… parfois pas de manière optimale (en disponibilité) !

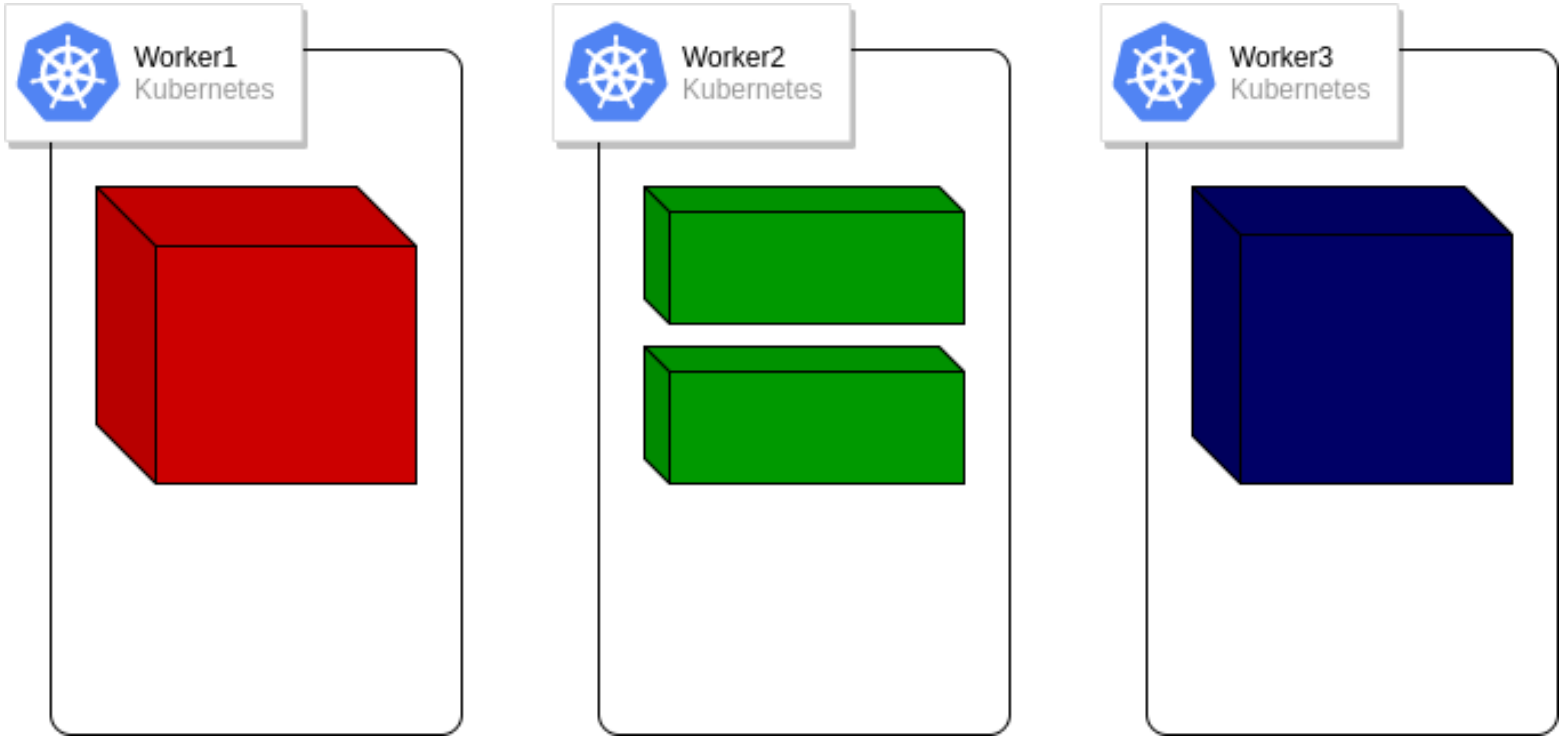


Affinity et AntiAffinity

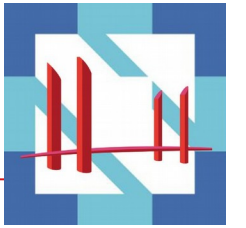


Par défaut, Kubernetes réparti les Pods

… parfois pas de manière optimale (en disponibilité) !



Affinity et AntiAffinity



Depuis K8s 1.4, on peut ajouter des contraintes (affinité/anti affinité) sur les labels

```
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: component
              operator: In
              values:
              - web
          topologyKey: kubernetes.io/hostname
        weight: 100
```

Availability zones

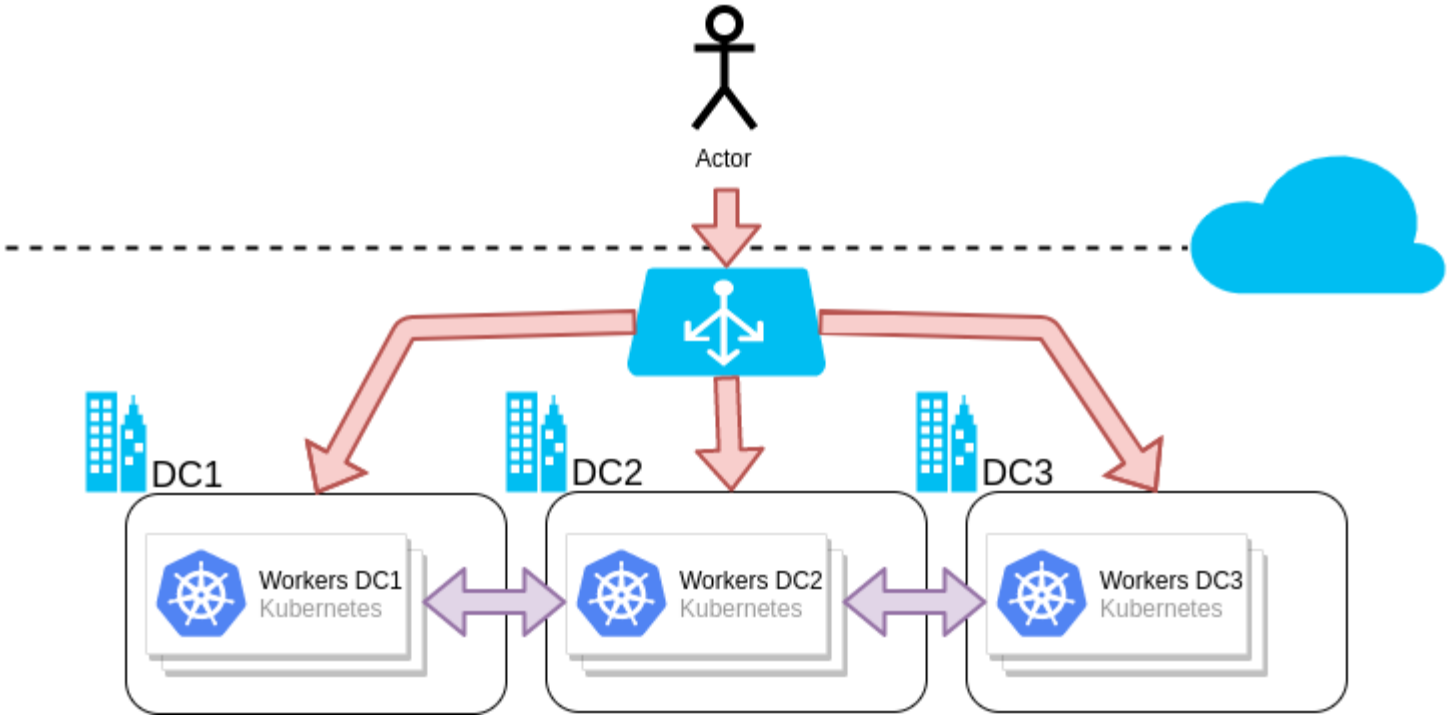
« Si un serveur s’éteint, l’application fonctionne encore ! On est hautement disponibles !!! »

Ceci est un arrêt « coup de poing »

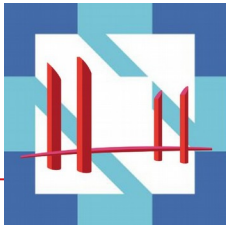
Availability zones

Les clouds publics proposent des DCs distincts mais proches :

- Protection en cas de panne générale
- Peu d’impact sur les latences internes



Attention, slide disruptif



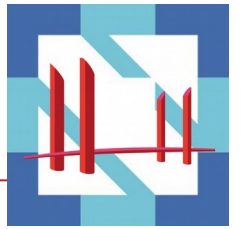
Avec de la réplication, de l’affinité et les AZ, on s’est prémuni de la disruption « involontaire » (aka les pannes)

Mais quid de la disruption « volontaire » ?

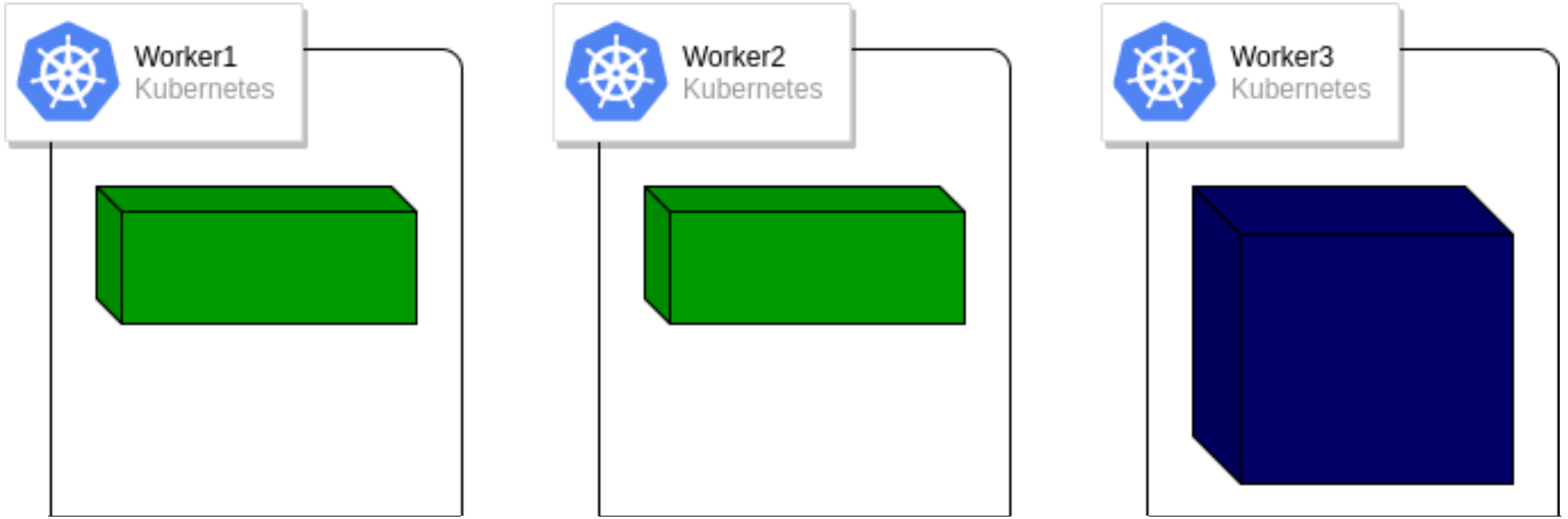
- Mise à jour nécessitant la régénération d’un objet
- Suppression manuelle d’un objet
- « Drain » d’un node pour maintenance

https://www.revolgy.com/blog/kubernetes-in-production-poddisruptionbudget

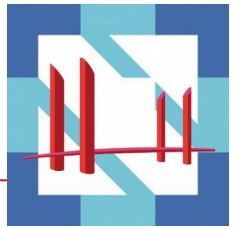
Disruption volontaire



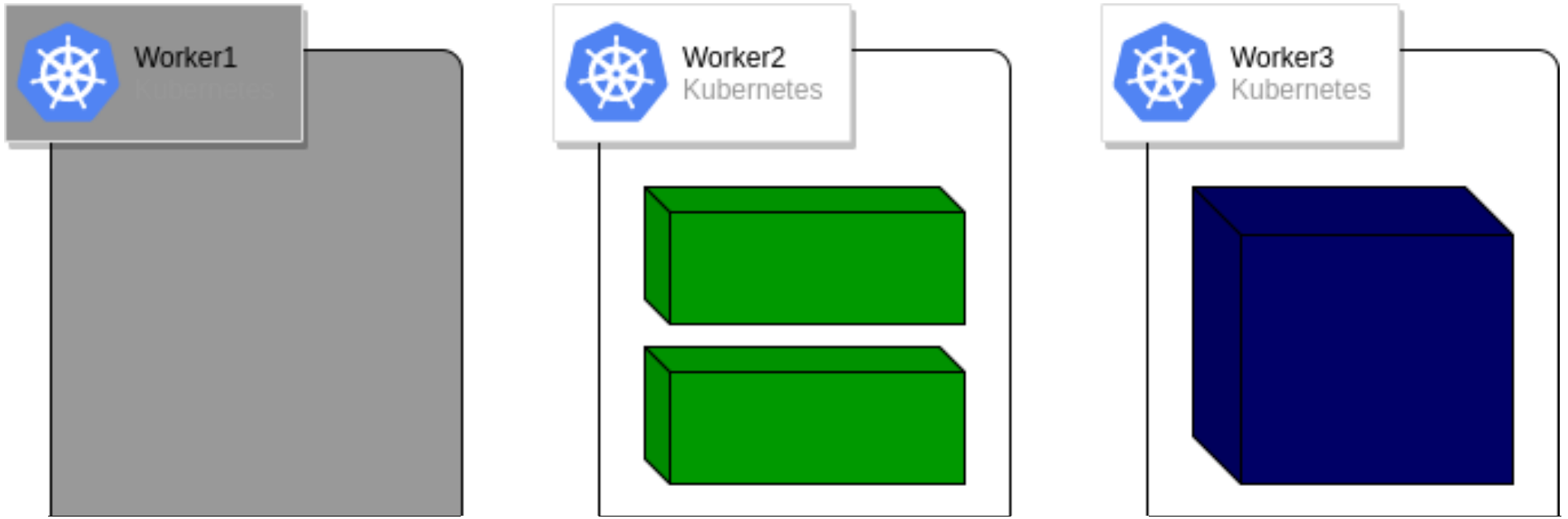
L’anti-affinité par hôte n’aide pas dans ce cas !



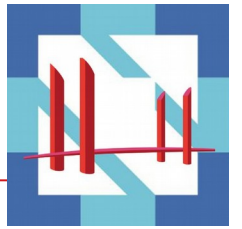
Disruption volontaire



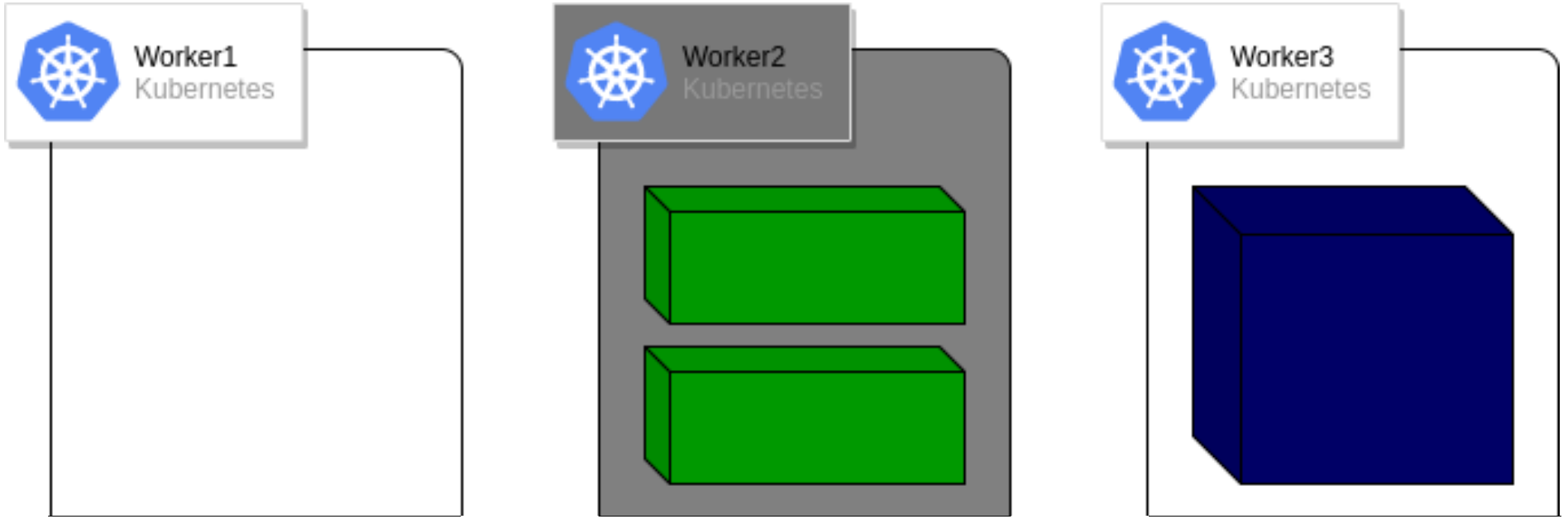
L’anti-affinité par hôte n’aide pas de ce cas !



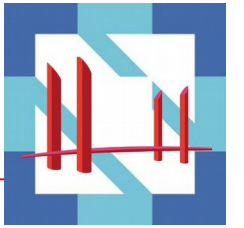
Disruption volontaire



L’anti-affinité par hôte n’aide pas de ce cas !



PodDisruptionBudget

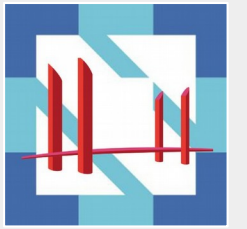


Création d’un PodDisruptionBudget (si nécessaire)

- Actions disruptives bloquées si les conditions ne sont pas respectées

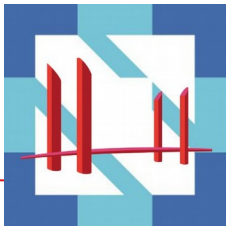
```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com



Helm / Tiller

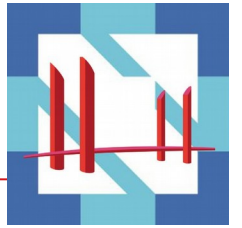
Déployer « a la mano » dans K8s



Objets dans Kubernetes ⇒ YAML (== très verbeux)

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8 # Update the version of nginx from 1.7.9 to 1.8
        ports:
        - containerPort: 80
```

Helm

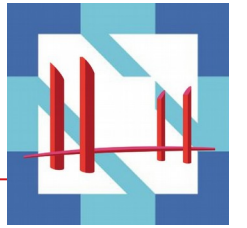


En 2018, le projet HELM a rejoint la CNCF (sous-projet de Kubernetes avant)

Helm (client) et Tiller (serveur) offrent :

- Créer des templates variabilisés d’applications
- Déployer/Mettre à jour/Rollback ces templates
- Magasin d’application disponible sur Github

Tiller is down !



« Dev : Ma pipeline est **rouge** car mon *helm install* échoue !

Error: UPGRADE FAILED: transport is closing »

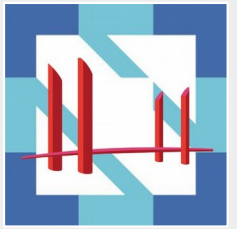
2 problèmes rencontrés :

- Tiller n’est pas hautement disponible
- Tiller stocke l’historique de tous les déploiements

Pistes d’amélioration :

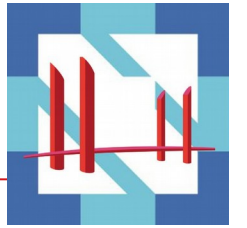
- Restreindre l’historique par déploiement
- 1 tiller par namespace

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com



Des quotas ? Pour quoi faire ?

Limits / requests



Colocation ⇒ Partage des ressources

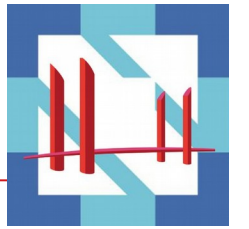
Partage des ressources ⇒ Fixer des limites

- Limits
  - CPU : au-delà, throttling
  - RAM : au-delà, kill du pod
- Requests
  - Utilisé lors du scheduling des pods
  - et pour l’autoscaling des nœuds

```
resources:
  limits:
    cpu: "4"
    memory: "1Gi"
  requests:
    cpu: "2"
    memory: "128Mi"
```

Si limits = requests, le pod est garanti de pouvoir exister

Quota et LimitRange



Il arrive que certains « oublient » de fixer des limites

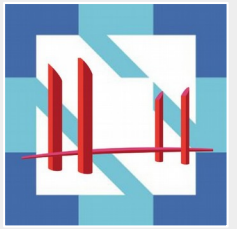
Deux solutions simples :

- Fixer des quotas totaux, par namespace
- Fixer des limites par défaut, par namespace et par objet (Pod par ex.)

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
    type: Container
```

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

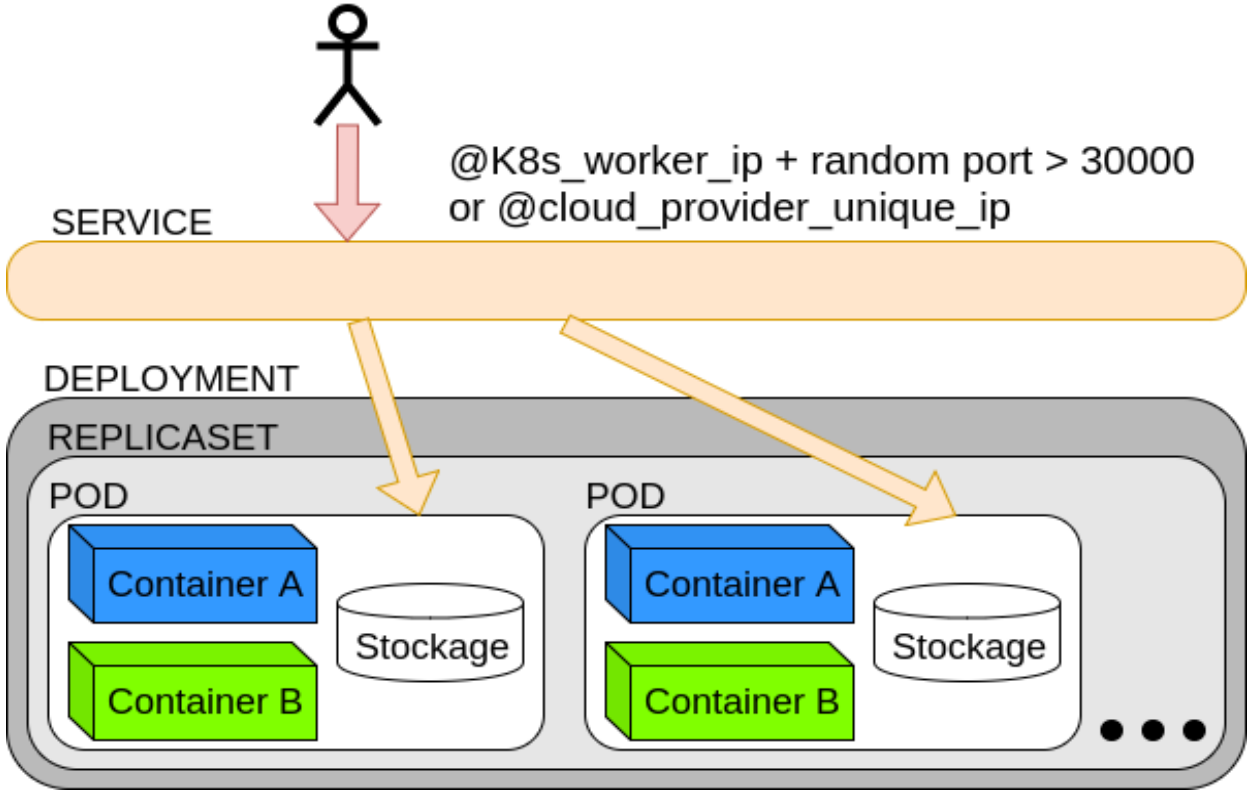


Ingress controllers

Services et Ingress

Pour accéder aux Pods depuis l'extérieur

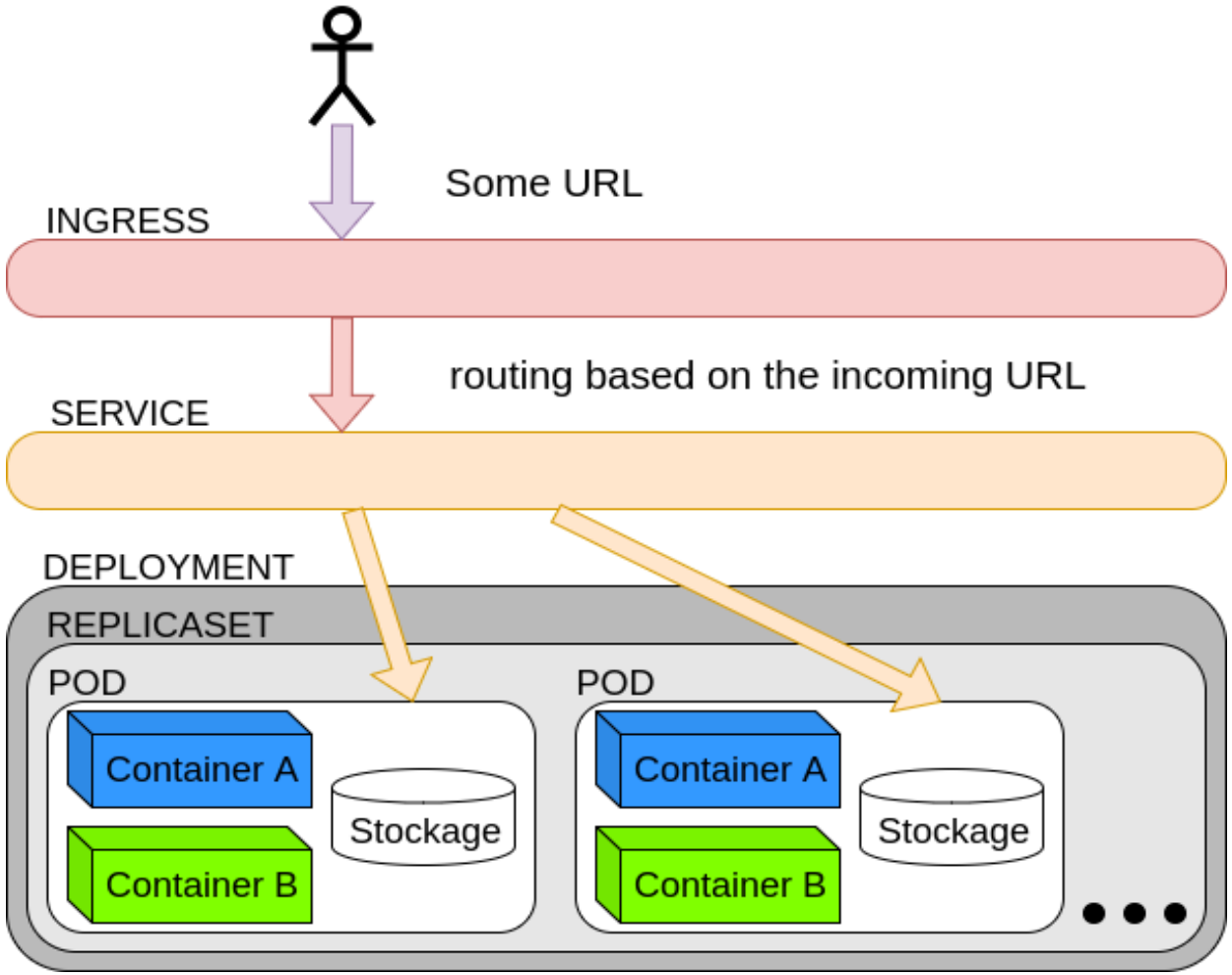
- Services
- Ingress



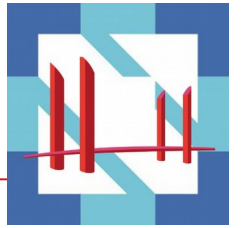
Services et Ingress

Pour accéder aux Pods depuis l'extérieur

- Services
- Ingress



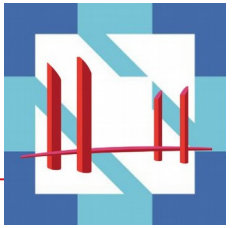
NGINX Ingress Controller



est l’Ingress Controller « par défaut »

- Historiquement le « de facto standard » du loadbalancing + reverse proxy
- Seul « Ingress Controller » sur le Github de Kubernetes
- Pourtant pas membre de la CNCF, contrairement à Envoy

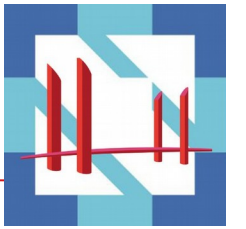
« Ça va trancher, chérie »



Pourtant, il n’est pas forcément le plus adapté à Kubernetes :

- Pas de « vrai » rechargement à chaud de la configuration
- Dommage pour les connexions actives (websockets / 10s) !

Pain(less) NGINX Ingress Controller



Deux liens utiles sur le sujet :

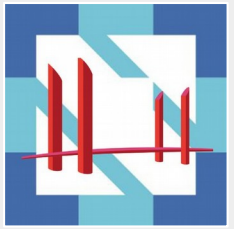
- https://github.com/kubernetes/ingress-nginx/issues/2461
- https://danielfm.me/posts/painless-nginx-ingress.html

Solutions (« workarounds » plutôt) :

- Limiter les reloads (*--enable-dynamic-configuration*)
- timeout+++ sur les connexions actives (*--worker-shutdown-timeout*)
- Segmenter les namespaces (1 Ingress Controller par namespace)
- ou...

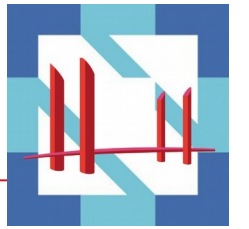
- Changer d'Ingress Controller pour un qui se recharge VRAIMENT à chaud (Envoy, Traefik, ...)

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com



Healthchecks

Healthchecks, Liveness, Readiness



Dans tous les Pods Kubernetes :

- **Readiness probe** ⇒ vérifie qu'un Pod est prêt à recevoir du trafic
- **Liveness probe** ⇒ vérifie l'état de santé d'un Pod

```
livenessProbe:
  httpGet:
    path: /ping
    port: 8080
```

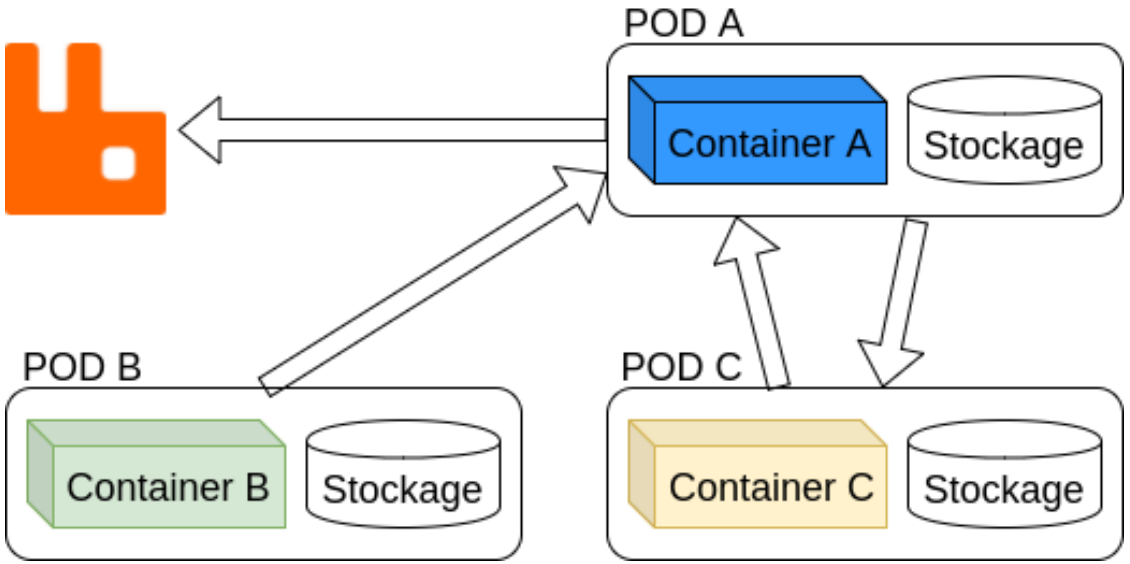
```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
```

Chez Lectra, nous avons aussi une bonne pratique :

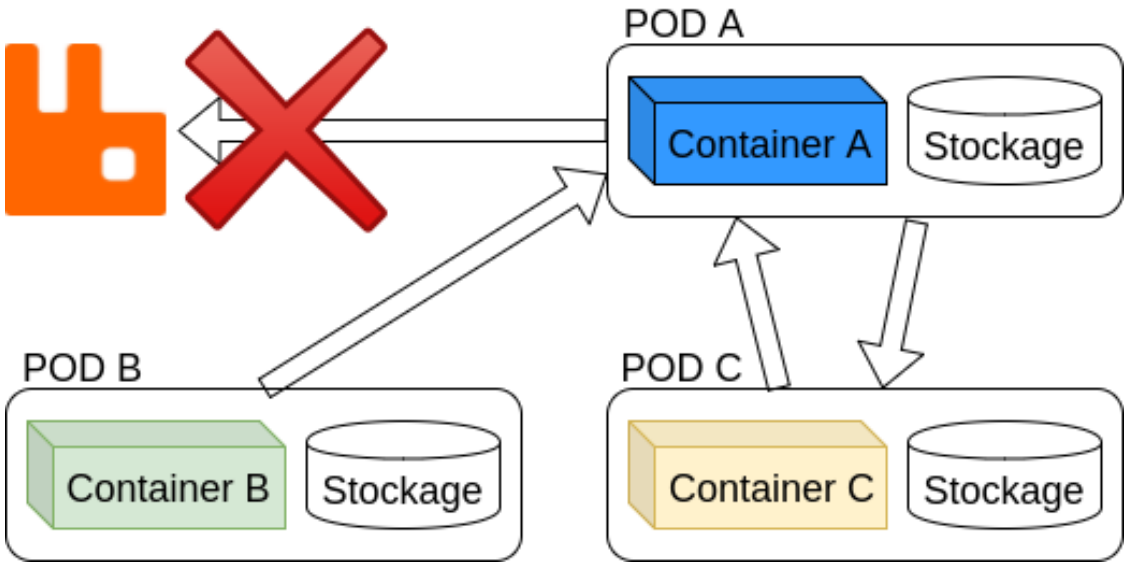
- Les µservices exposent un /health avec test des dépendances

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

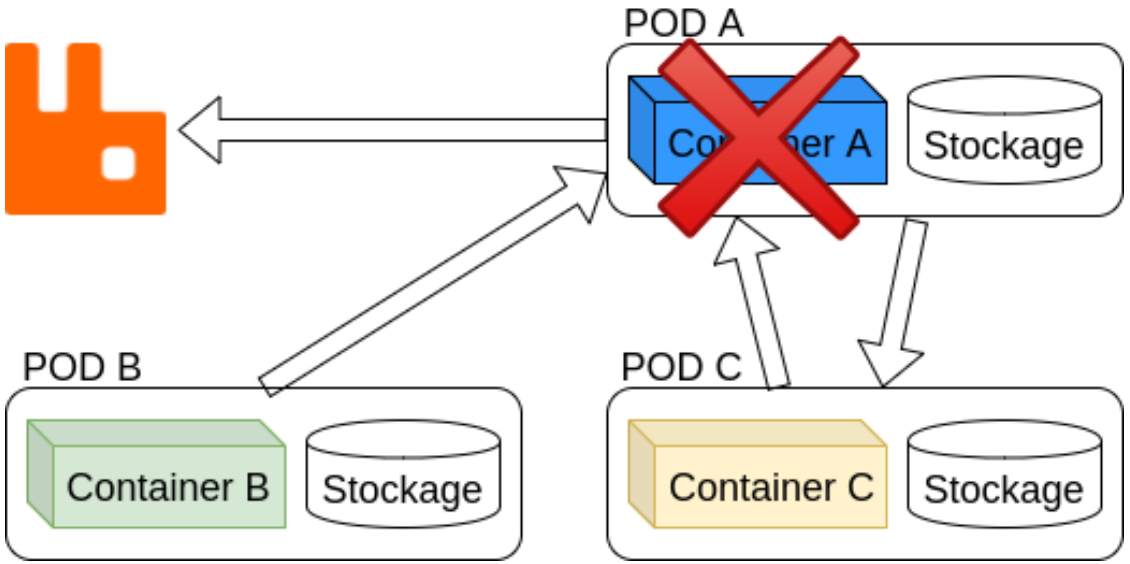
Healthchecks, Liveness, Readiness



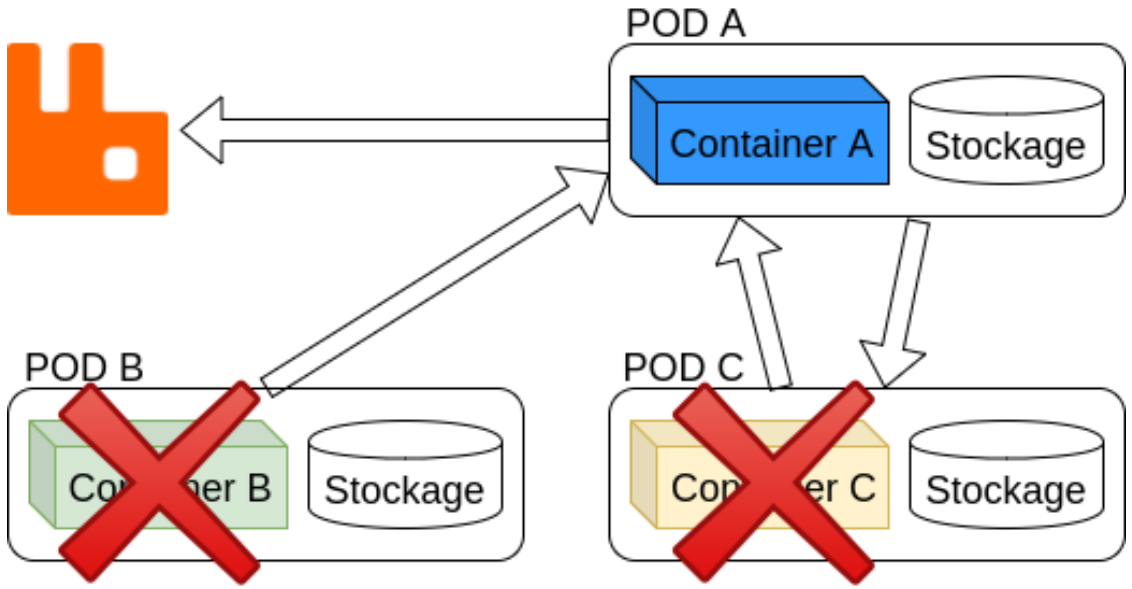
Healthchecks, Liveness, Readiness



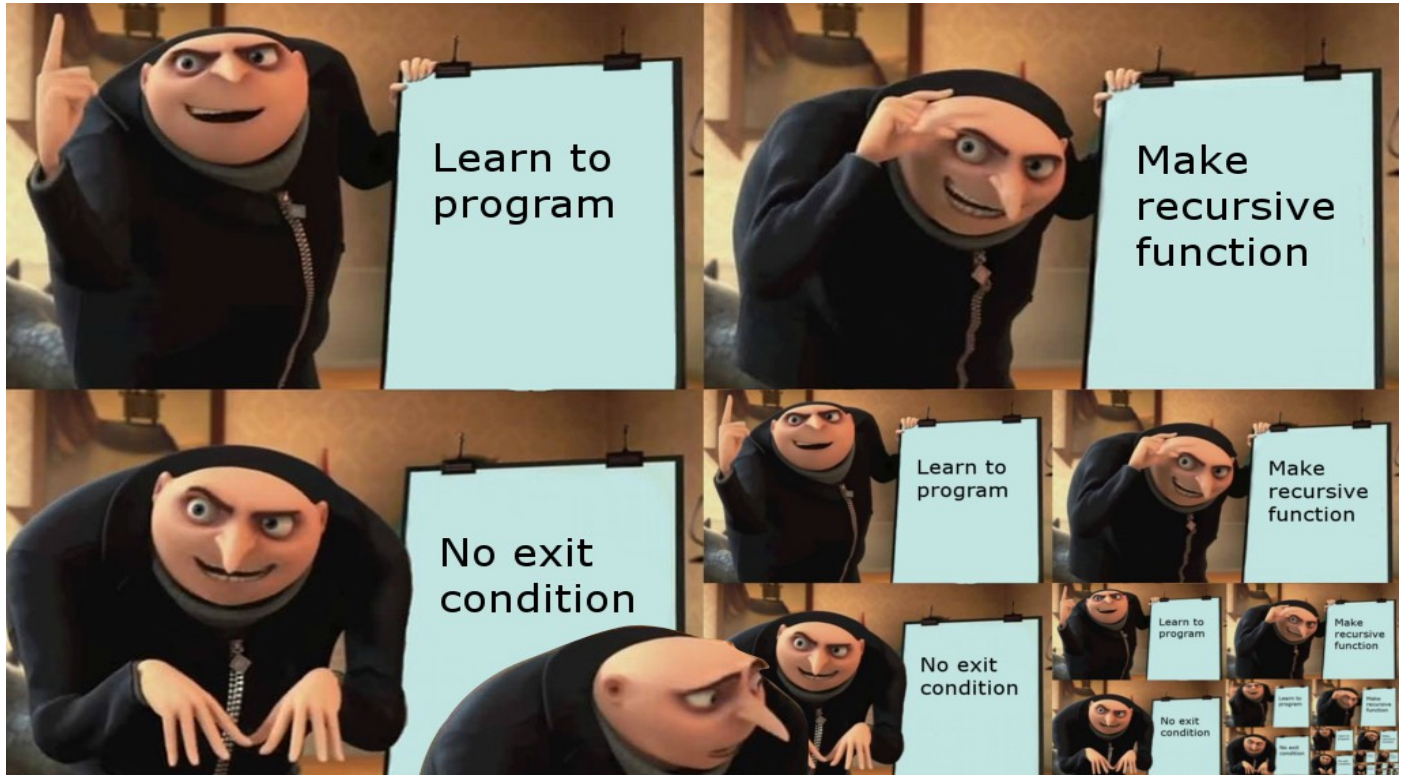
Healthchecks, Liveness, Readiness



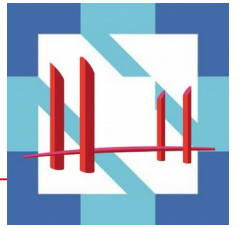
Healthchecks, Liveness, Readiness



Healthchecks, Liveness, Readiness



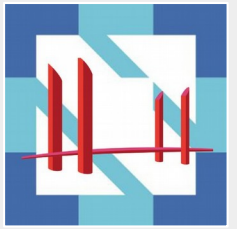
Healthchecks, Liveness, Readiness



Avoir un **/health** pour vérifier les dépendances externes, c’est bien.

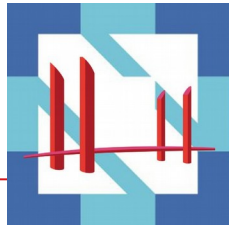
MAIS il ne faut pas l’utiliser pour la **Readiness**, et encore moins pour la **Liveness** !

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com



C’est déjà fini ?

Reste à faire (2019)



Non, ce n’est pas fini !

- Ingress Controller est toujours NGINX
- 1 seul Tiller par cluster
- Tuning du monitoring (Prometheus, alertes)
- Nodes qui passent « Not Ready » sans explication

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

Denis GERMAIN
@zwindler / @zwindler_rflx / d.germain@lectra.com

Des questions ?