Kubernetes and Software Load-Balancers

Agenda

- Definition of Software Load-Balancer
- An overview of Kubernetes: from a high level introduction to explanation about networking
- Load-Balancing in / with Kubernetes
- How to integrate a software Load-Balancer within Kubernetes
- Demo!

About the speaker

- Baptiste Assmann (bedis9@gmail.com)
- System Engineer at BROCADE on vADC product portfolio (former Zeus)
- Many contributions to HAProxy Community. Maintainer of the internal DNS resolver

- Learned Kubernetes from a LB point of view… so definitively not an expert...
- Slides contain a lot of information about Kubernetes, but they are not exhaustive and the author may have used some “shortcuts”

Special thanks to Brocade who allow (and indeed encourage) me to continue contributing to HAProxy!

Software load-balancer ???

Definition of a software load-balancer

- A software load-balancer is a software component you can install and run on a raw operating system, without an hypervisor layer
- A Virtual Appliance (VA) is not a software since it embeds its own operating system

- Most software load-balancers are open source: HAProxy, pound, LVS, pen
- There are also proprietary softwares: Brocade vTM (former Zeus)
- Some open source web servers implement load-balancing features: apache, nginx, …

Those are not software load-balancers, despite what their marketing may say:

(A10|kemp|F5|[^]\+) Virtual Appliance

An overview of Kubernetes

Introduction to Kubernetes

- It's a **suite of software components** that runs on Linux
- **Clustering technology** with master and slave roles
- From kubernetes.io homepage:
  - *Production-Grade Container Orchestration*
  - *Automated container deployment, scaling, and management*
- It allows:
  - Simple management of containerized applications
  - Microservices
  - ...
- Point of view of the speaker: YAAL*

* *Yet Another Abstraction Layer* :)

An overview of Kubernetes

Kubernetes best friends

Because Kubernetes itself can’t do anything…, we need at least:

- a **server**: either bare metal server or a VM in an hypervisor
- an **operating system** (“Legacy”: Ubuntu/Centos or container oriented: CoreOS/RancherOS/…)
- a **container engine** (Docker or rocket)
- some **network overlay** routers

OR

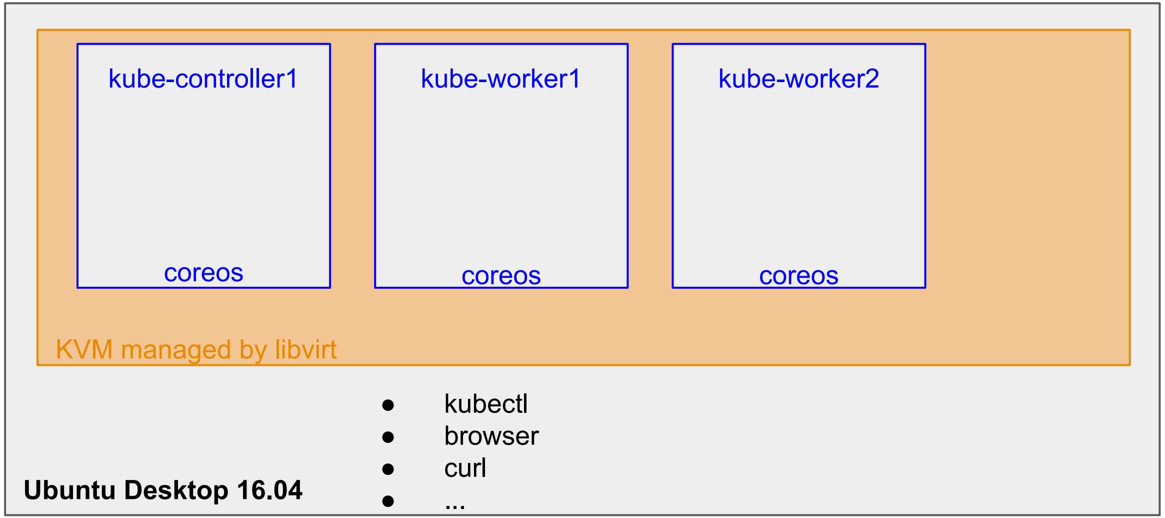
- a cloud provider
- a cloud provider which proposes “KaaS” (*Kubernetes as a Service*)

⇒ Kubernetes is pretty agnostic from the underlying layer point of view

An overview of Kubernetes

Kubernetes lab example

- lab environment set up in **KVM** managed by libvirt on an Ubuntu 16.04 desktop
- kubernetes runs on top of **CoreOS** beta (1248.4.0)
- one **kubernetes master node** + etcd (called kube-controllerX)
- two **kubernetes worker nodes**
- overlay network based on **flannel**
- forget your rant about **systemd** :-)



An overview of Kubernetes

Most important kubernetes **units** to know

- a **namespace** is a virtual cluster in kubernetes. It owns a “delegated” DNS domain name
- a **pod** is a set of **containers** which are grouped together to deliver a service or an application. A **pod** always belongs to a **namespace**
  Read https://kubernetes.io/docs/user-guide/pods/ for more details
- a **service** is the way to expose a **pod** either to other pods in the cluster or to the outside world
- **replication controller** is the automatic **pod** scaler tool
- **deployment**: way to manage **pods** in a certain state. Manages and control the state changes
- **labels** are “tags” which are used to group together different unit types
- **ingress** is the load-balancer which allows “internet” to reach **services**

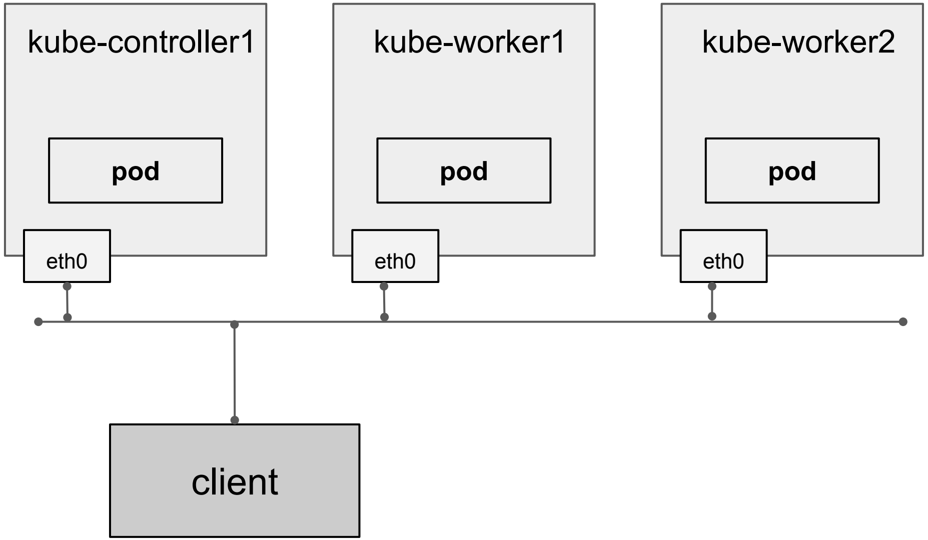
An overview of Kubernetes

Introduction to networking in Kubernetes

From the kubernetes documentation:
(https://kubernetes.io/docs/admin/networking/)

- *Kubernetes assumes that **pods** can communicate with other **pods***
- *Kubernetes imposes the following fundamental requirements on any networking implementation:*
  - all **containers** can communicate with all other **containers** without NAT
  - all **nodes** can communicate with all **containers** (and vice-versa) without NAT
  - the IP that a **container** sees itself as is the same IP that others see it as

What this means in practice is that you can not just take two computers running Docker and expect Kubernetes to work. You must ensure that the fundamental requirements are met



An overview of Kubernetes

A suite of software components

- For devs / ops / devops / admins:
  - kubectl (or curl…)
- For **master nodes**:
  - kube-apiserver
  - etcd *(not provided by kubernetes)*
  - kube-controller-manager
  - kube-scheduler
  - addons (DNS, GUI, resource monitoring, cluster level logging)
- For **worker nodes**:
  - kubelet
  - kube-proxy
  - docker / rkt *(not provided by kubernetes)*

Some other softwares may be installed to customize your cluster

An overview of Kubernetes

The master node

- Manages the Kubernetes cluster
- run **administrative pods** (dashboard, DNS, etc…)
- *Optionally* can run the **etcd** cluster, but on the bare underlying OS directly
- Main processes, based on cluster implementation:
  - **systemd**: manages system start up
  - **journalctl**: manages logs on the system
  - **etcd**: storage for the configuration and health of the cluster *(opt: could run on a dedicated server)*
  - **kube-apiserver**: manages the cluster configuration and state
  - **kube-controller-manager**: embeds core control loop which regulates the state of the system
  - **kube-scheduler**: assign containers to nodes based on performance, capacity, affinity, etc...
  - **flannel**: overlay network to reach containers on third other nodes
  - **fleet**: simple distributed service manager which operates at “cluster” scale (will be deprecated soon in favor of Kubernetes itself)
  - **kubelet**: manages containers on a local node

An overview of Kubernetes

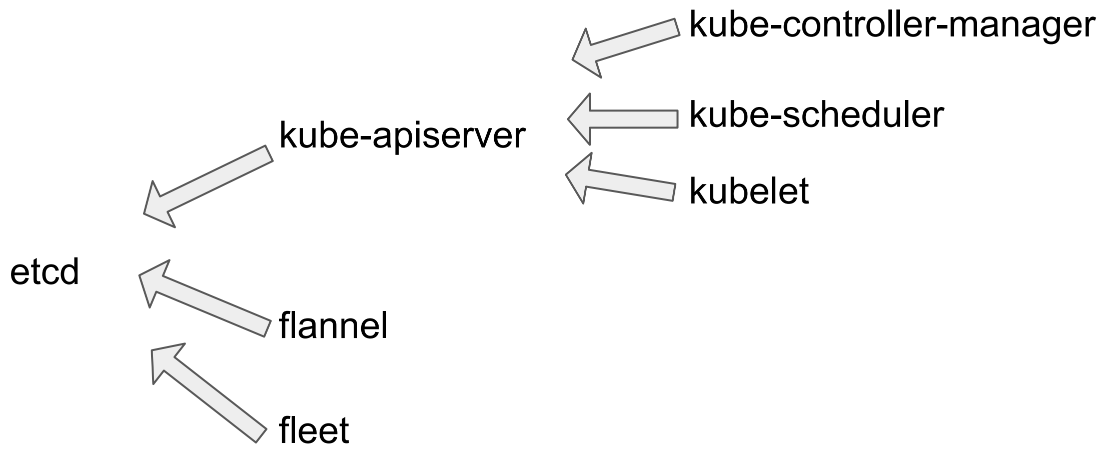
The worker node

- Run the **application pods**
- Main processes, based on cluster implementation:
  - **systemd**: manages system start up
  - **journalctl**: manages logs on the system
  - **flannel**: overlay network to reach containers on third other nodes
  - **fleet**: simple distributed service manager which operates at “cluster” scale (will be deprecated soon in favor of Kubernetes itself)
  - **kubelet**: manages containers on a local node
  - **kubeproxy**: proxifies a “node port” to a “container port”
  - **docker** / **rkt**: container engine

An overview of Kubernetes

A focus on etcd

- Distributed and reliable **key value storage**
- Many services from kubernetes cluster rely on **etcd**, either directly or through the **kube-apiserver**
- etcd is used to mainly store **configuration** and **status** of kubernetes units



An overview of Kubernetes

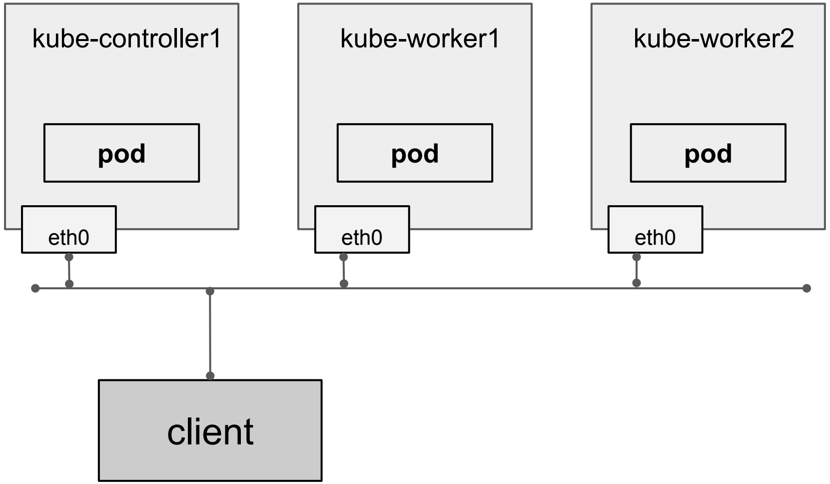
Networking in Kubernetes: traffic between **pods**

Remember, from the introduction:

- *Kubernetes imposes the following fundamental requirements on any networking implementation:*
    - all **containers** can communicate with all other **containers** without NAT
    - all **nodes** can communicate with all **containers** (and vice-versa) without NAT
    - the IP that a **container** sees itself as is the same IP that others see it as

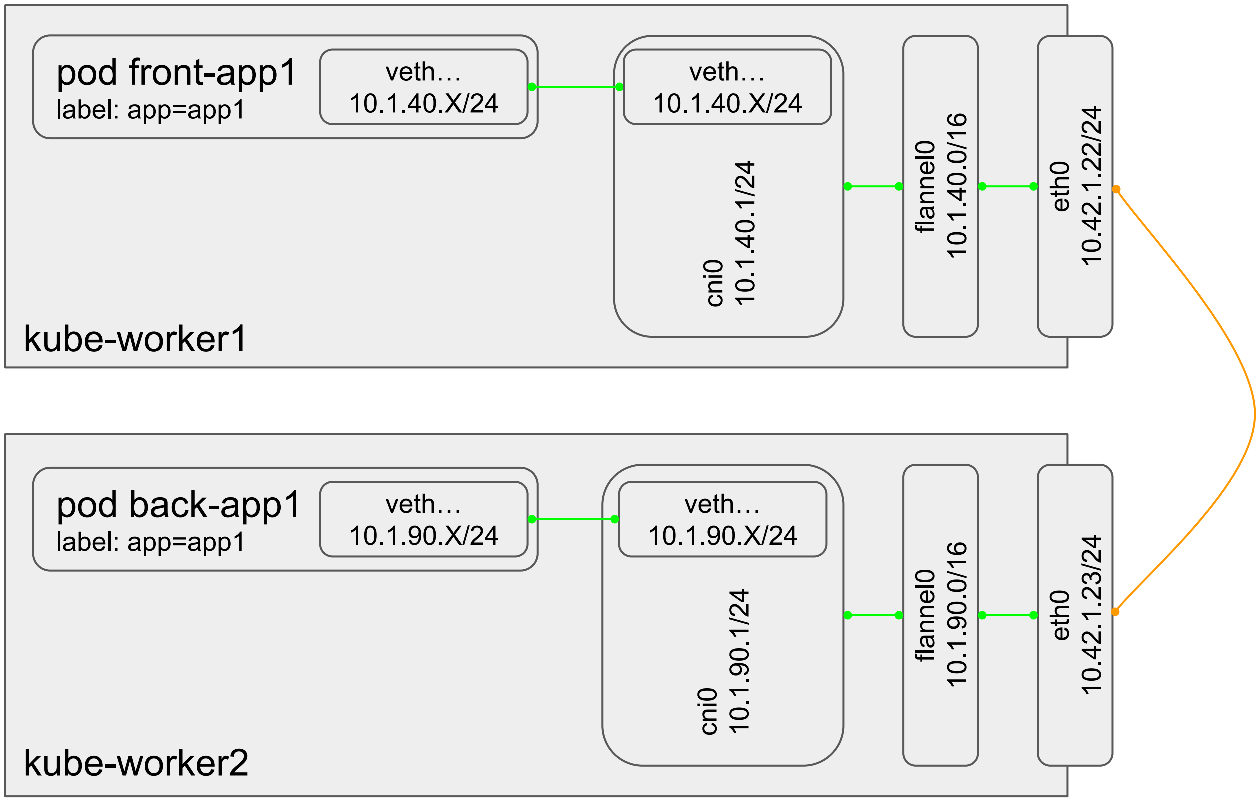
Now, in real life:

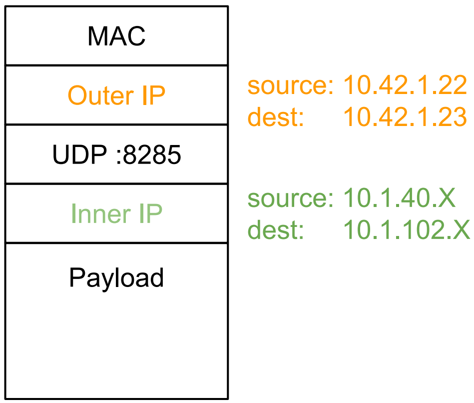
- KISS Overlay networking **flannel**:
  container traffic flowing between **nodes** on the external network is encapsulated into UDP (vxlan)
- Interesting alternatives: **calico** or **contiv**
  (both more secure, more reliable, more scalable)



An overview of Kubernetes

Networking in Kubernetes: traffic between **pods** with **flannel**





cni0: Container Network Interface
veth…: network namespace interface

An overview of Kubernetes

Networking in Kubernetes: from “internet” to **pods**

#define internet “clients who aren’t members of the kubernetes cluster”

- “internet” clients can’t **reach** pods directly, a **service** must be set
- **Services** are managed by **kube-proxy** and are set up using a bunch of iptables rules
- **Pods** are selected using the **endpoints** associated to the **service**, either using iptables or a userland proxy ⇒ a **service** can load-balance at layer 4

There are 2 modes (non exhaustive) to open a service to the outside world:

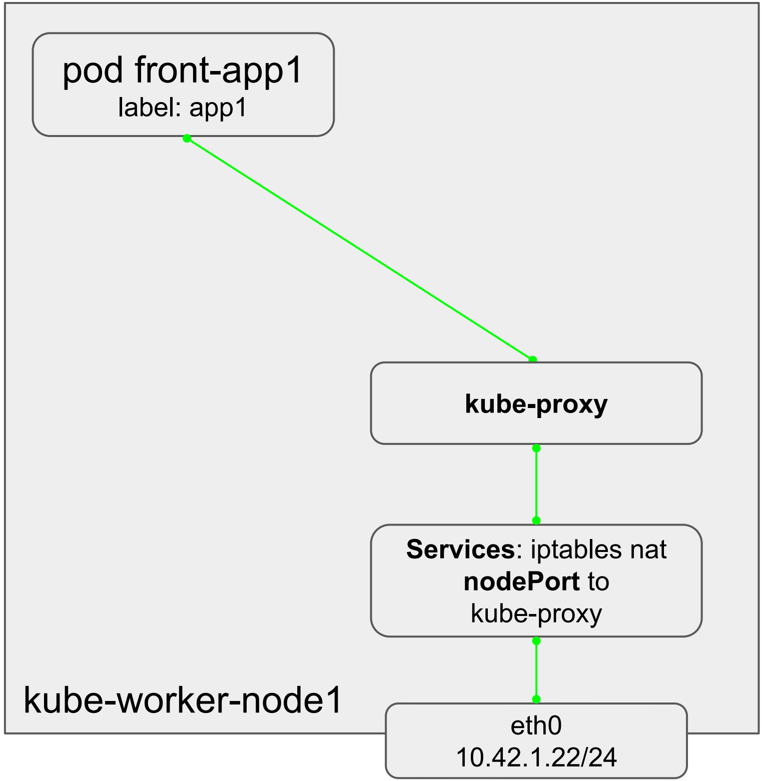
1. **nodePort**
2. **clusterIP**

- Use an **external load-balancer** to reach the **service** on a **node** port
  ⇒ some traffic may be double bounced on the network (if there is no **pod** for this **service** on the hitted **node**)

In our demo lab, my laptop belongs to “internet”

An overview of Kubernetes

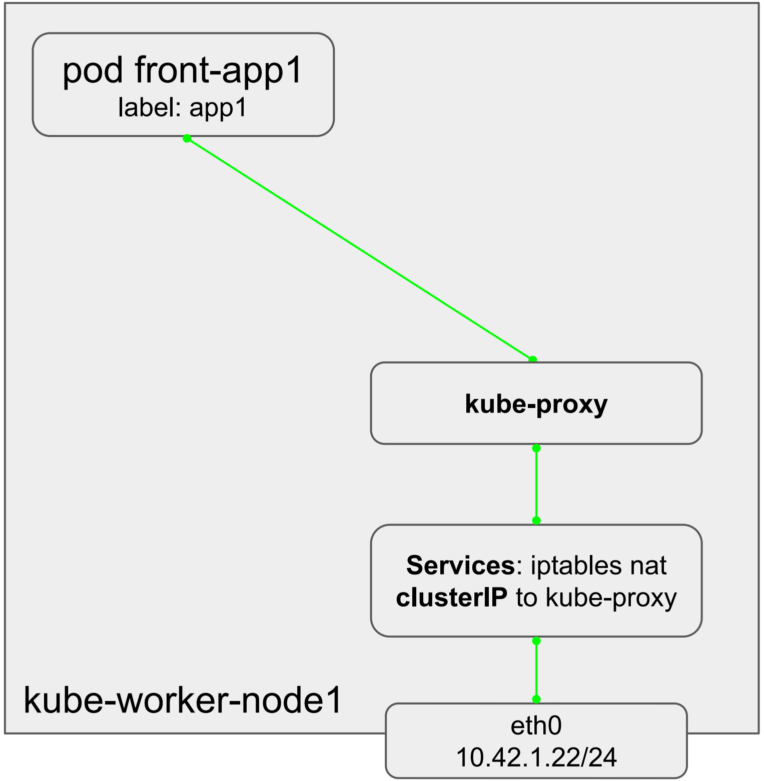
Networking in Kubernetes: from “internet” to **pods**: **Services** and **nodePort**



- “internet” client reaches **Service** on [node IP]:**nodePort** on any cluster **node**
  IE: 10.42.1.22:32004
  or: 10.42.1.23:32004
- kubernetes **Services** is actually iptables which NATs traffic to a local **kube-proxy**
- **kube-proxy** runs in a container and proxifies the connection to one of the **pod** belonging to the **Service** (through the **endpoints**)
- the selected **pod** may be located on an other **node**, then we’ll use overlay network to reach it out

An overview of Kubernetes

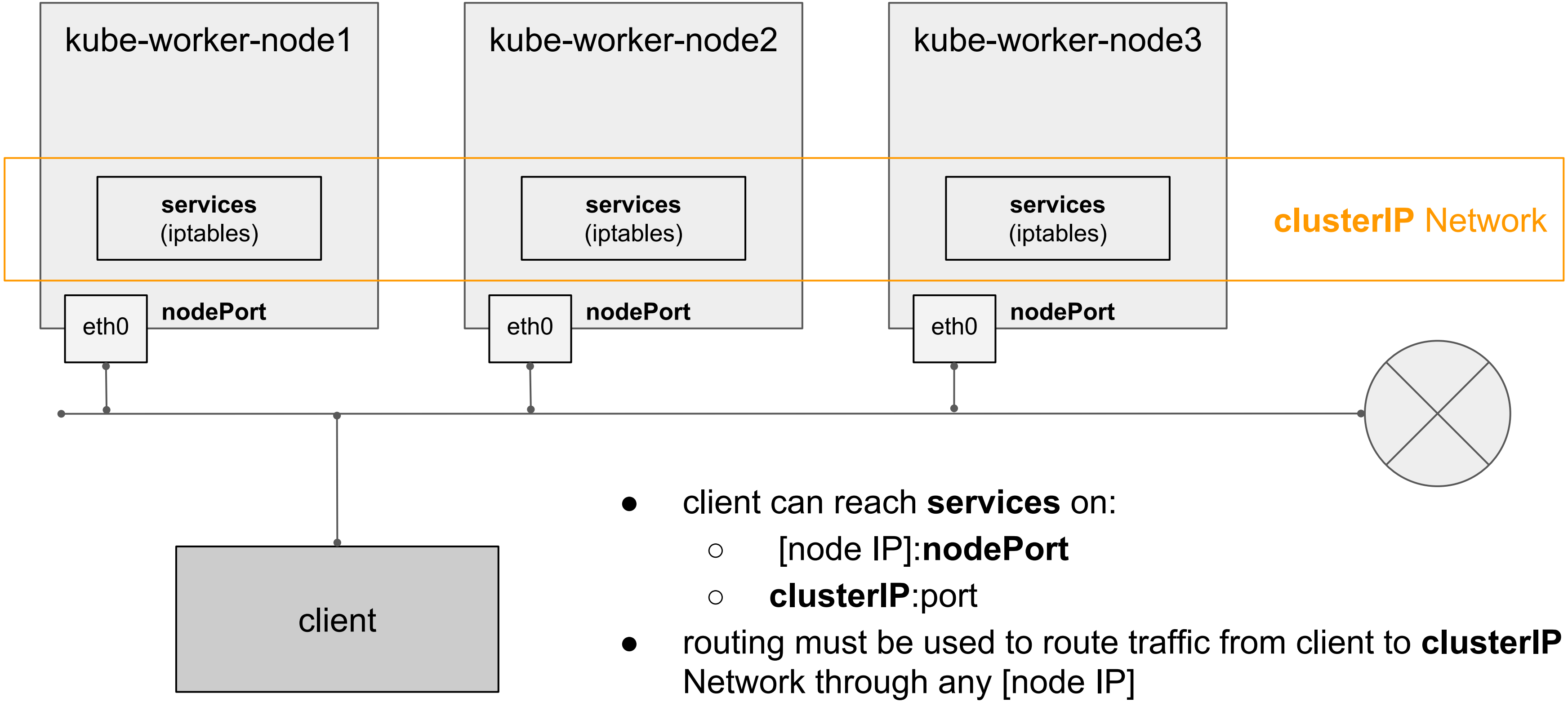
Networking in Kubernetes: from “internet” to **pods**: **Services** and **clusterIP**



- **clusterIP network** can be routed to any **node**
- **clusterIP network** is “virtual” and spans over all **nodes**
- “internet” client reaches **Service** on **clusterIP**:port on any cluster **node**
- kubernetes **Services** is actually iptables which NAT traffic to a local **kube-proxy**
- **kube-proxy** runs in a container and proxifies the connection to one of the **pod** belonging to the **Service**
- the **pod** may be located on an other **node**, then we’ll use overlay network to reach it out

An overview of Kubernetes

Networking in Kubernetes: summary



- client can reach **services** on:
  - [node IP]:**nodePort**
  - **clusterIP**:port
- routing must be used to route traffic from client to **clusterIP** Network through any [node IP]

An overview of Kubernetes

Other **Kubernetes** “cool” features not detailed / presented in these slides

Mainly related to orchestration and automation:

- container placement
- auto-scaling
- auto-healing
- volume management (storage)
- Resource usage monitoring
- health checks
- rolling update

⇒ All those features make **Kubernetes** a very dynamic and challenges environment (changes may happen quite often)
⇒ A **Load-Balancer** between clients and **services** hosted in kubernetes makes sense to deliver a smooth experience.

Load-Balancing in/with Kubernetes

Introduction

- a **Service** can be used to load-balance traffic to **pods** at layer 4
- **Ingress resource** are used to load-balance traffic between **pods** at layer 7 (introduced in kubernetes v1.1)
- we may set up an **external load-balancer** to load balance “internet” traffic to **services**

Load-Balancing in/with Kubernetes

A focus on **ingress**

- an **ingress** load-balancer is composed by the following element:
  - an **ingress resource**: kubernetes piece of configuration describing the load-balancing rule to be applied
  - an **ingress controller**: a software reverse-proxy load-balancer which applies the **ingress resources** rules:
    - the **ingress controller** runs as a pod in the kubernetes cluster
    - default **ingress controller** implementation in kubernetes in based on nginx, but anyone could write his own (hence we are here :))
- we need to use a **service** to give access to “internet” clients to the **ingress** load-balancer

Load-Balancing in/with Kubernetes

A focus on **ingress**: minimal implementation

- The **ingress controller** can self configure using the information provided by environment variables in the **pod** and the **kube-apiserver**:
    - **kube-apiserver** credentials, URL and CA cert
    - ingress identifier

- From there, it has to:
    - parse the **ingress** rules to build up the core configuration
        - for each **ingress** rule, parses the corresponding **Service** / **Endpoint** to fill up the pool / backend
        - create the corresponding HTTP routing rules using Host + path
    - keep watching the **ingress** and **endpoint resources** for updates (new rules, add / remove **pods** in the service)
    - based on the **service** type, adapt the pool / backend: either use DNS resolution (**headless services**) or list of nodes

Load-Balancing in/with Kubernetes

A focus on **ingress**

- The **ingress resources** describes Host headers and URL paths to route to **Services**:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: haproxy-ingress
  labels:
    app: demo
  annotations:
    kubernetes.io/ingress.class: "haproxy"
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /echo
            backend:
              serviceName: echoheaders
              servicePort: 8080
          - path: /
            backend:
              serviceName: default-http-backend
              servicePort: 8080
```

Load-Balancing in/with Kubernetes

Information provided by the **ingress resource**

- a protocol parser (currently only **http** is supported)
- a **host** header to match on incoming requests
- a list of **path** matched on the **host** header above and used to point the incoming request to a **kubernetes service**

```
rules:
  - host: example.com
    http:
      paths:
        - path: /echo
          backend:
            serviceName: echoheaders
            servicePort: 8080
        - path: /
          backend:
            serviceName: default-http-backend
            servicePort: 8080
```

- http://example.com/echo ⇒ **service** echoheaders
- http://example.com/ ⇒ **service** default-http-backend
- deny any other requests

Load-Balancing in/with Kubernetes

MISSING Information in the **ingress resource**

- the load-balancer engine to use (nginx / HAProxy / vTM / etc...)
- load-balancing algorithm and persistence
- timeouts
- health checks
- …

We can use “annotations” for this purpose, with our own parameters:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: haproxy-ingress
  labels:
    app: demo
  annotations:
    kubernetes.io/ingress.class: "haproxy"
    kubernetes.io/ingress.persistence.sourceip: "echoheaders"
```

- use HAProxy as the load-balancer engine
- enable persistence based on client IP for the **service** echoheader

Load-Balancing in/with Kubernetes

Implementing an external load-balancer

- well, the external load-balancer is like the **ingress** one:
  - it needs to self configure by polling / watching the **kube-apiserver**
  - it has to be able to reach **nodes** and **services** configured on top of them
- it can't natively use DNS based **services** (headless), since it does not have access to **kube-dns** server (could be done using network plumbing)
- it can run either on a dedicated **node**: those which have the `role=loadbalancer` flag
- or on a dedicated bare-metal server
- need to implement some cloudprovider features (such as **ExternalIP** management)

Load-Balancing in/with Kubernetes

Integrating a software load-balancer with Kubernetes

- We need a software load-balancer: **HAProxy** or **Zeus/vTM** are rock solid
- We need to write a piece of code (called the **controller**) to:
  - watch the **kube-apiserver**
  - generate the configuration for the load-balancer
  - apply the configuration to the load-balancer
- Create a **pod** with the software load-balancer and its **controller** and integrate it into our private registry
- ah, we need a **private registry** :)

Load-Balancing in/with Kubernetes

Integrating a software load-balancer with Kubernetes

What could be improved on HAProxy:

- make the runtime DNS resolver smarter: when many records are returned, prefer using unused records whenever possible (WIP)
- update the internal DNS resolver to be able to use DNS response records (for ingress headless **services**) to fill up a backend (currently, we only resolve per server) (WIP)
- give the ability to manage more internal objects at run time:
  - create / delete / rename backends
  - create / delete / rename servers in backends
  - set / change a server’s fqdn on the CLI
- develop the **controller** as a built-in feature or as a third party software which can manage HAProxy

Load-Balancing in/with Kubernetes

Integrating a software load-balancer with Kubernetes

What could be done on vTM / zxtm:

- integration with vTM will require development of a **controller**, preferably as a built in feature
- make the cluster mode more flexible
- check how traffic IPs group can be set up into the external network (using the **node** host IP stack)

TIPs

Useful tips

- How to use cool (debugging) tools on coreos:
  https://coreos.com/os/docs/latest/install-debugging-tools.html
  WARNING: `echo "TOOLBOX_DOCKER_TAG=24" >>$HOME/.toolboxrc`
- To browse the kubernetes dashboard (GUI):
```
DASHPOD=$(kubectl get pods --namespace=kube-system | sed -n 's/^kubernetes-dashboard-\([^ ]\+\).*/\1/p')
kubectl port-forward kubernetes-dashboard-${DASHPOD} 8888:9090 --namespace=kube-system >/dev/null 2>&1 &
```
- it *seems* better to use **deployment** over **replicationcontroller** for ease of application code rollout
- If you can see a pod with `kubectl get pod`, then try `--namespace <NS>`
- Use BGP to announce kubernetes pod networks to outside world using **contiv**:
  http://contiv.github.io/documents/networking/bgp.html