Orchestrating Linux Containers

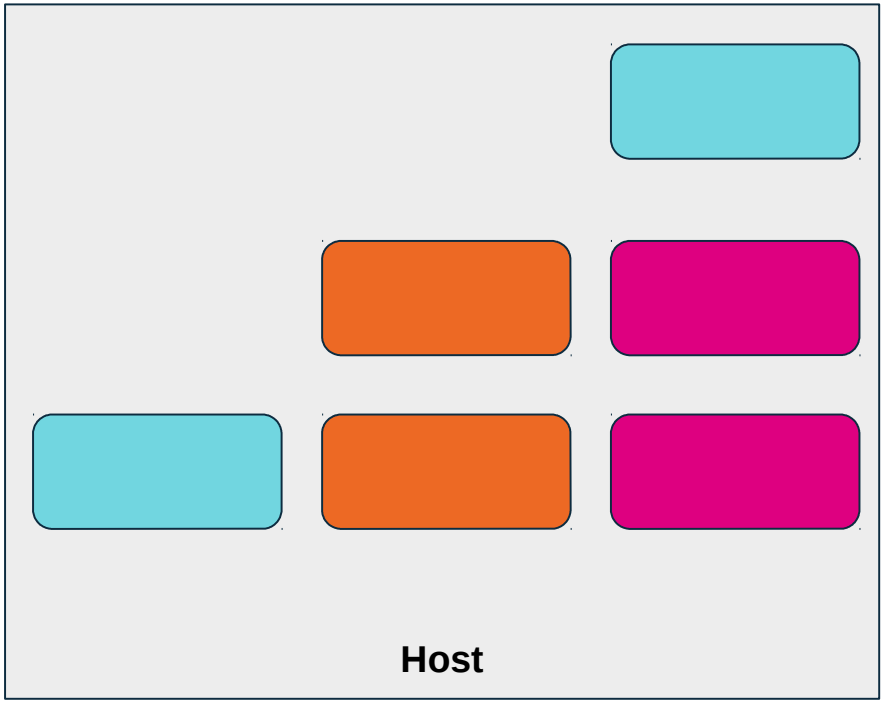
Flavio Castelli
Engineering Manager
fcastelli@suse.com

New development challenges

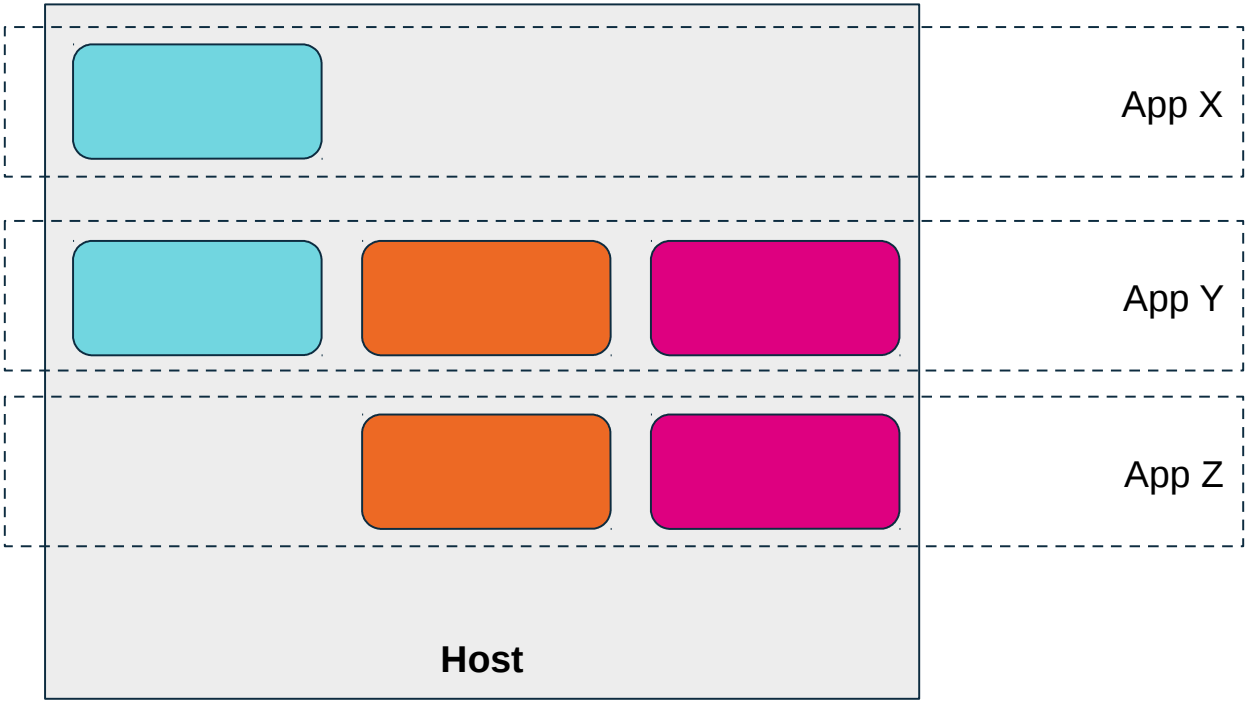
- Release early, release often
- Be flexible: react to changes quickly
- Adoption of micro services architectures
- Different cloud providers
- Higher complexity of cloud environments
- Hybrid cloud deployments
- Application super portability

Let’s talk about application containers...

Lightweight

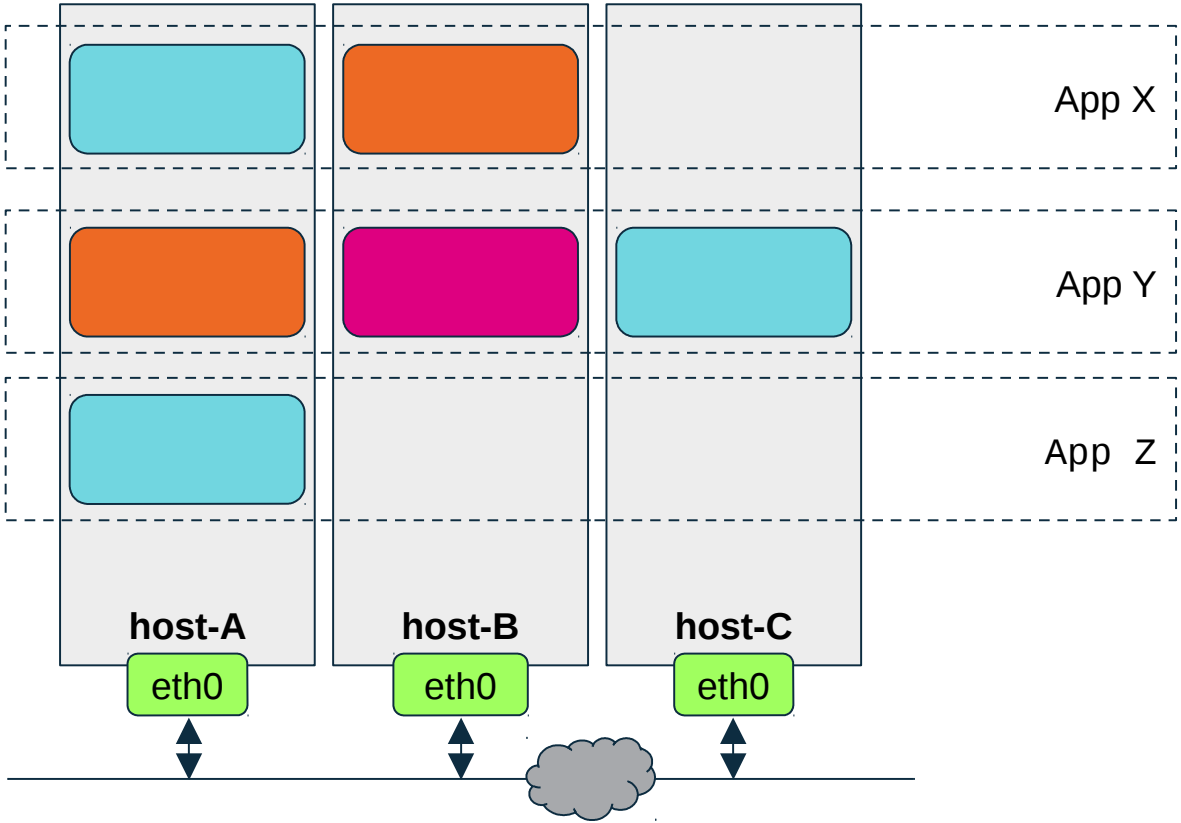


Composable



Getting serious...

Multi-host deployment



Some of the challenges

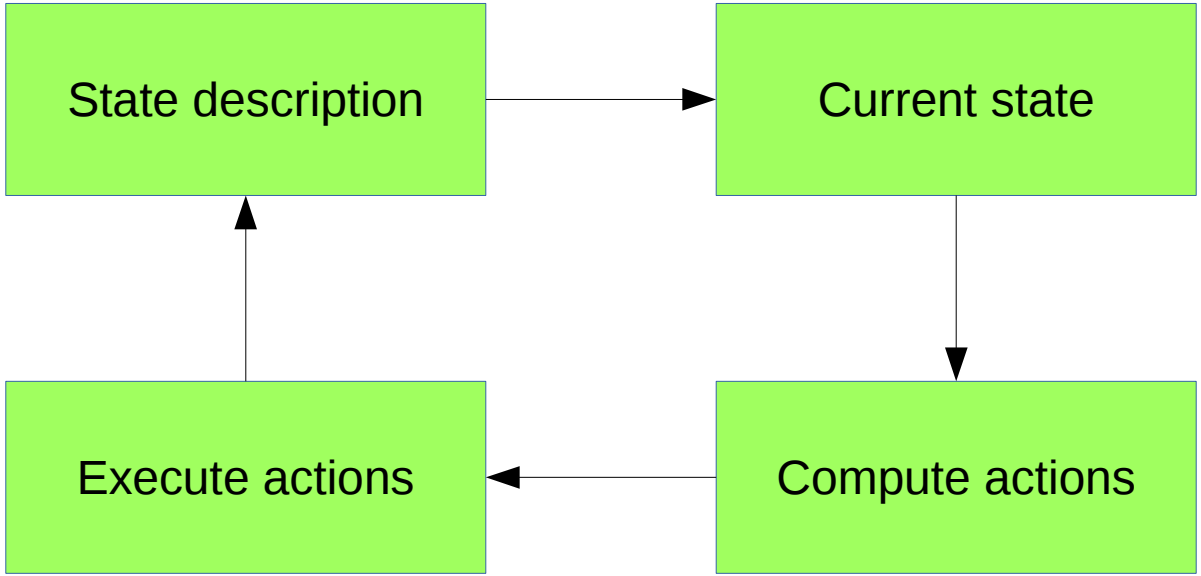
- Decide where to run each container
- Monitor nodes and containers
- Recover from failures
- Service discovery
- Expose services to external consumers
- Handle secrets (eg: credentials, certificates, keys,…)
- Handle data persistence

Container orchestration engines

How can they help us?

- No need to find the right placement for each container
- No manual recovery from failures
- Just declare a ***“desired state”***

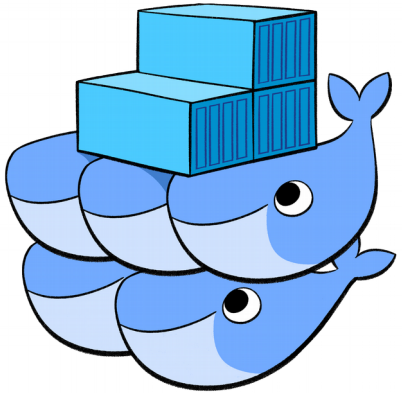
Desired state reconciliation



Manage applications, not machines

Which one?

kubernetes



Docker Swarm

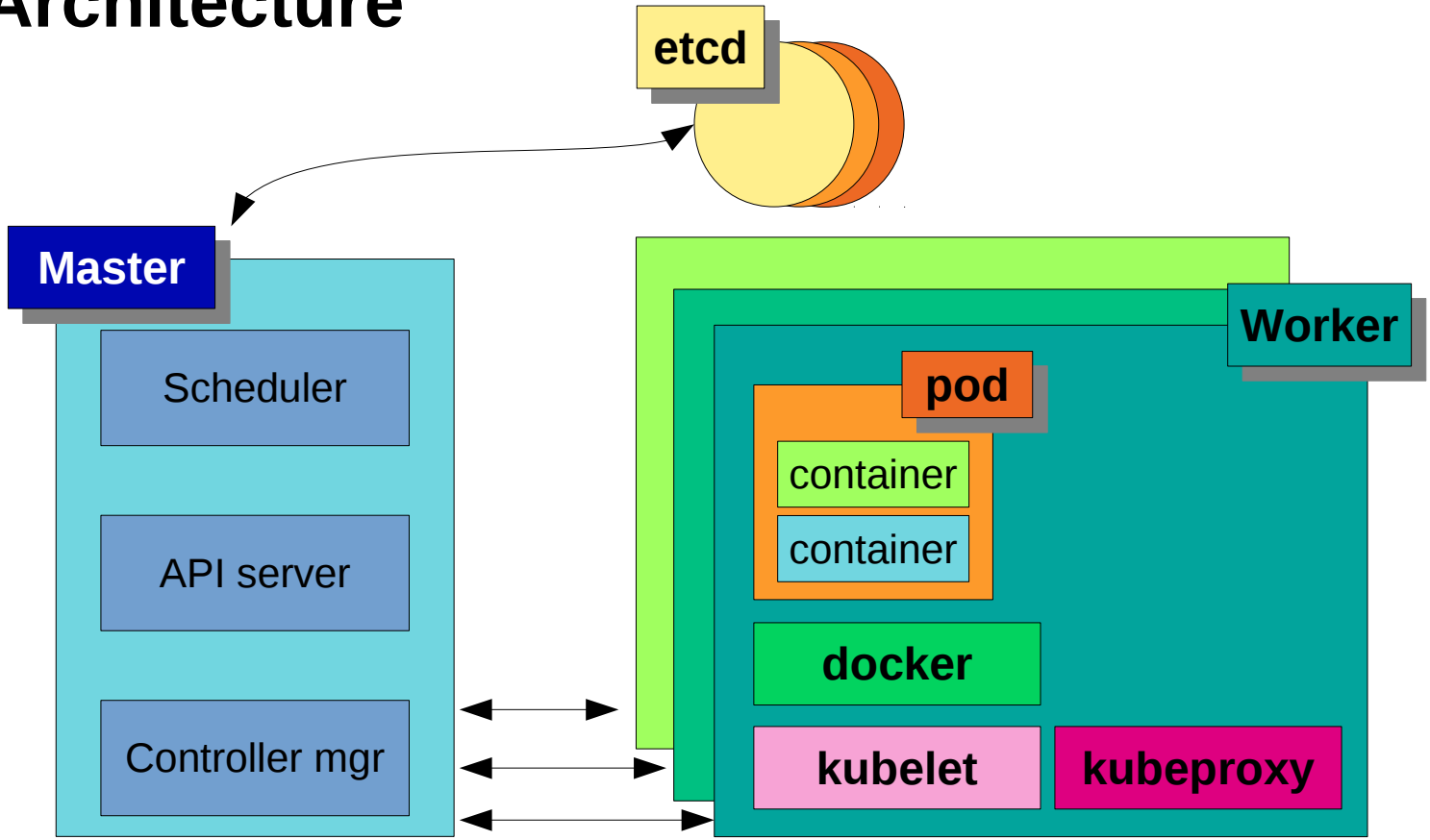
Nomad

MESOS

Kubernetes

- Created by Google, now part of CNCF
- Solid design
- Big and active community
- Opinionated solution, has an answer to most questions

Architecture



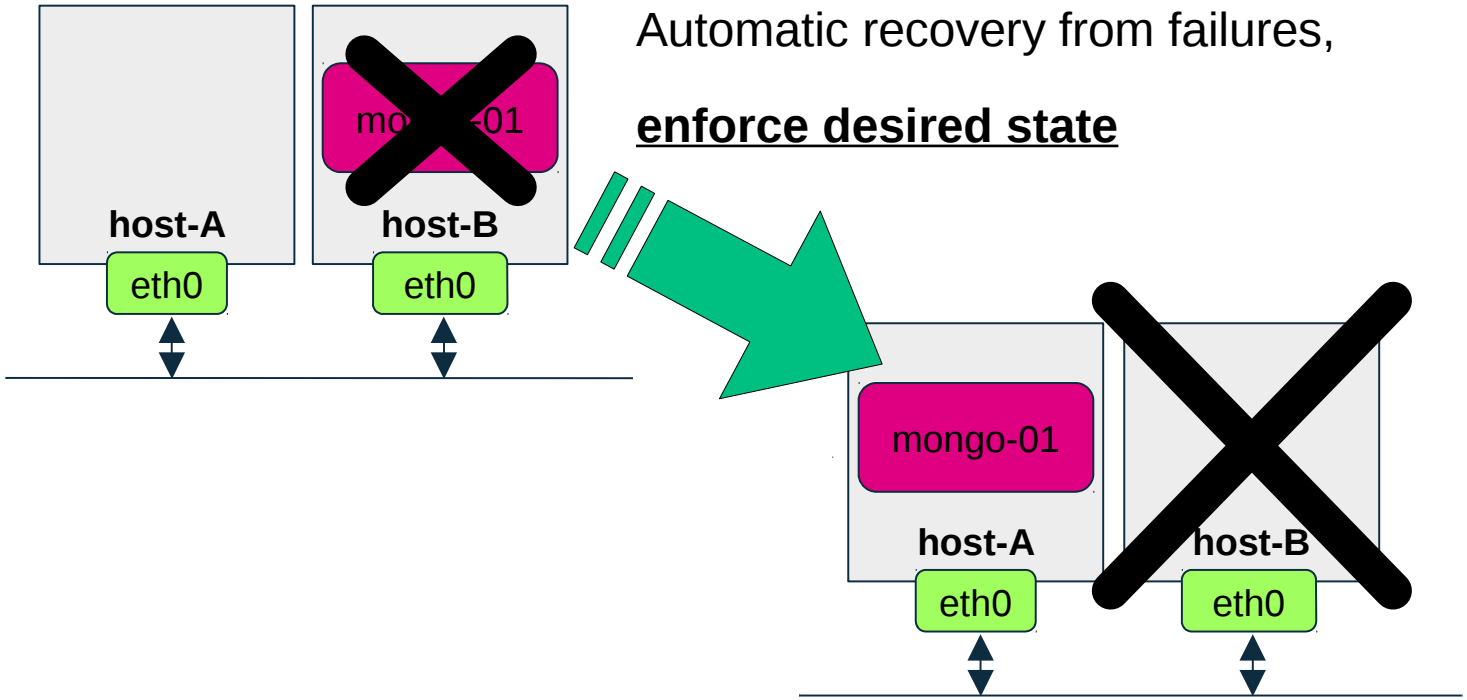
A concrete example

A simple web application

- Guestbook application
- Mongodb as database
- Both running inside of dedicated containers

Self-healing

Challenge #1: self-healing



Automatic recovery from failures,

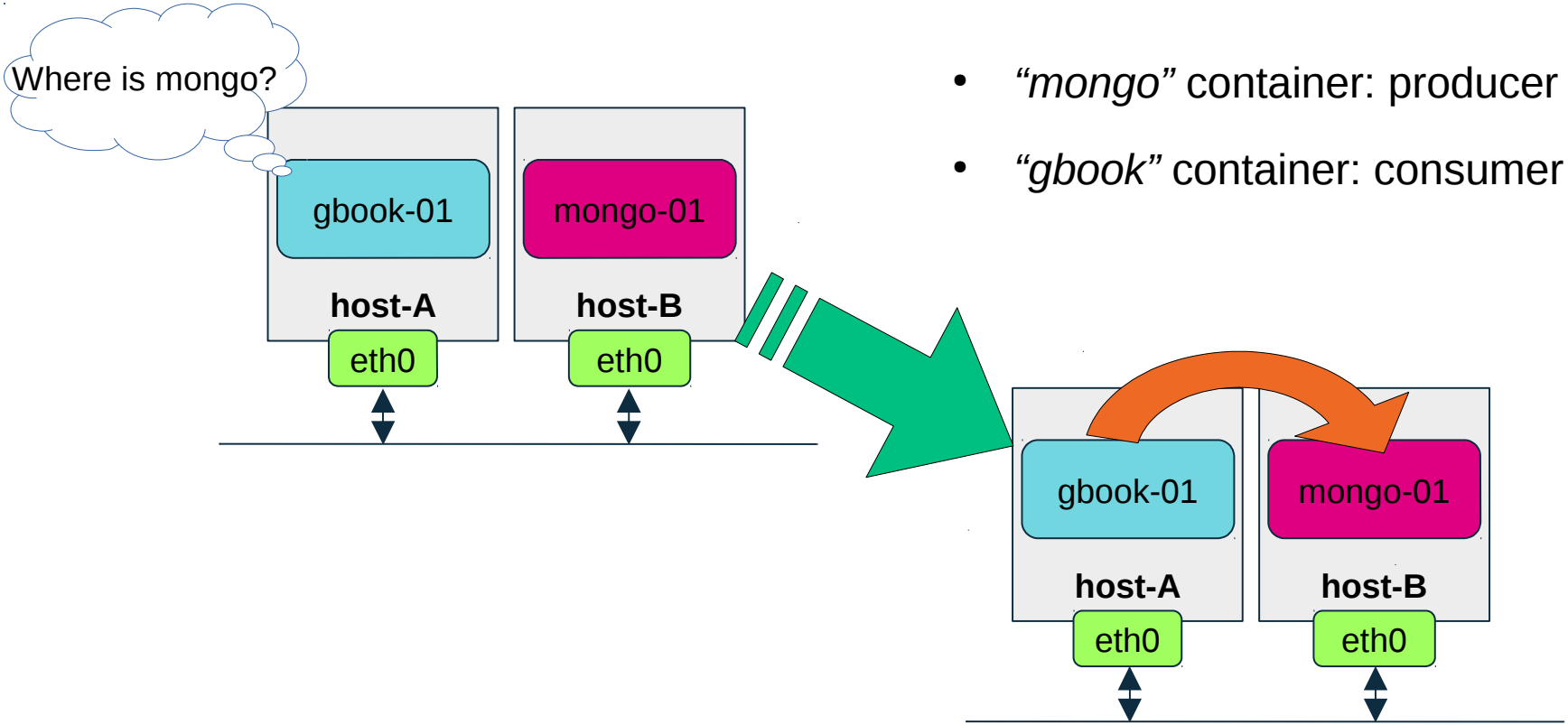
enforce desired state

Replica Set

- Ensure that a specific number of “replicas” are running at any one time
- Recovery from failures
- Automatic scaling

Service discovery

Challenge #2: service discovery



- *"mongo"* container: producer
- *"gbook"* container: consumer

Use DNS

- Not a good solution:
  - Containers can die/be moved somewhere more often
  - Return DNS responses with a short TTL → more load on the server
  - Some clients ignore TTL → old entries are cached

Note well:

- Docker < 1.11: updates `/etc/hosts` dynamically
- Docker >= 1.11: integrates a DNS server

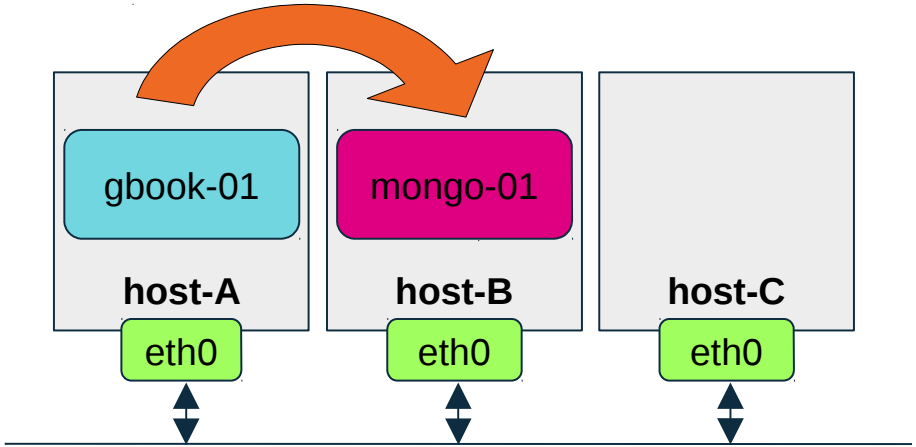
Key-value store

- Rely on a key-value store (etcd, consul, zookeeper)
- Producer register itself: IP, port #
- Orchestration engine handles this data to the consumer
- At run time either:
  - Change your application to read data straight from the k/v
  - Rely on some helper that exposes the values via environment file or configuration file

Handing changes & multiple choices

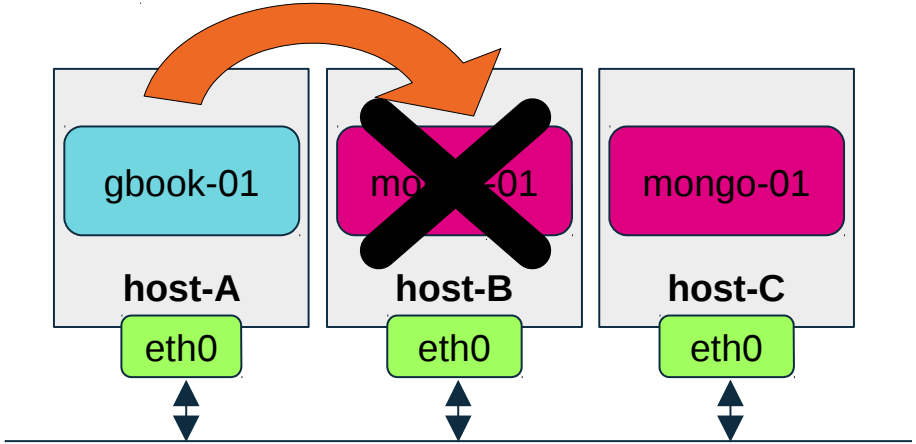
Challenge #3: react to changes

“gbook” is already connected to “mongo”



Challenge #3: react to changes

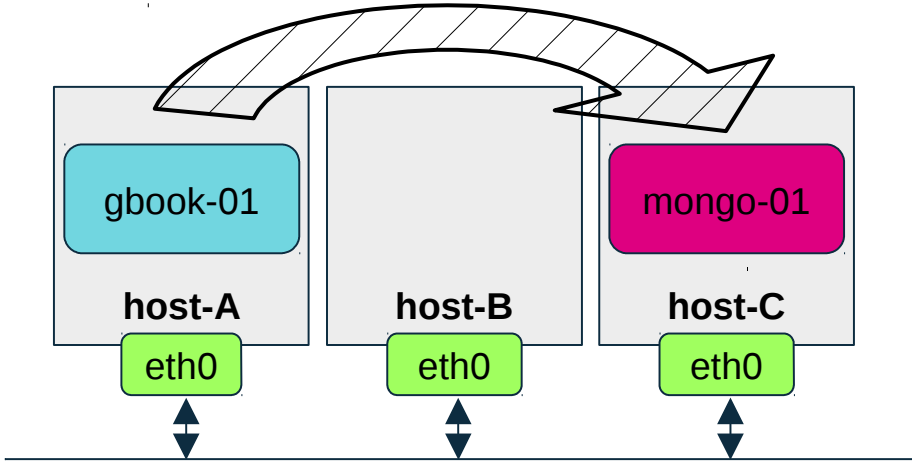
“mongo” is moved to another host → different IP



“gbook” points to to the old location → it’s broken

Challenge #3: react to changes

The link has to be reestablished

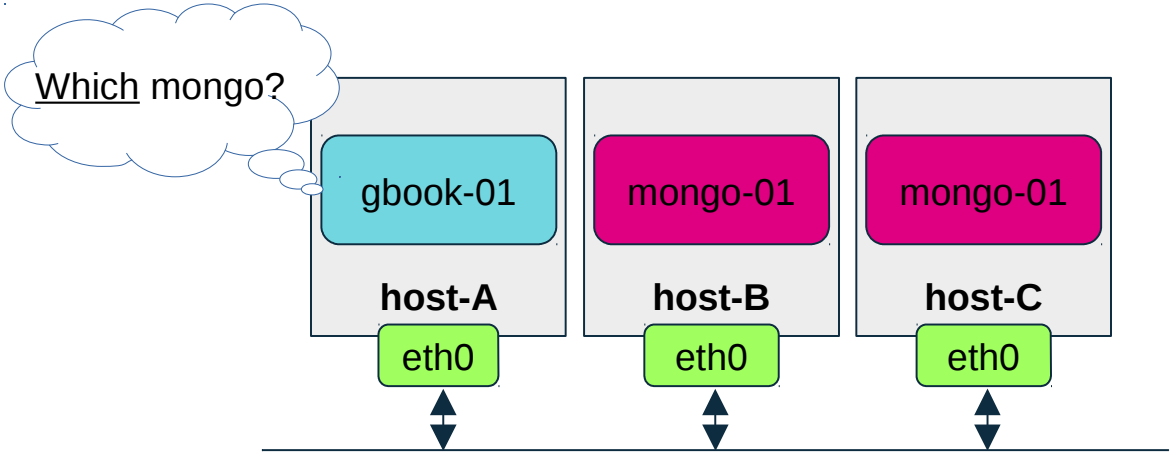


Containers can be moved at any time:

- The producer can be moved to a different host
- The consumer should keep working

Challenge #4: multiple choices

Multiple instances of the “mongo” image



Workloads can be scaled:

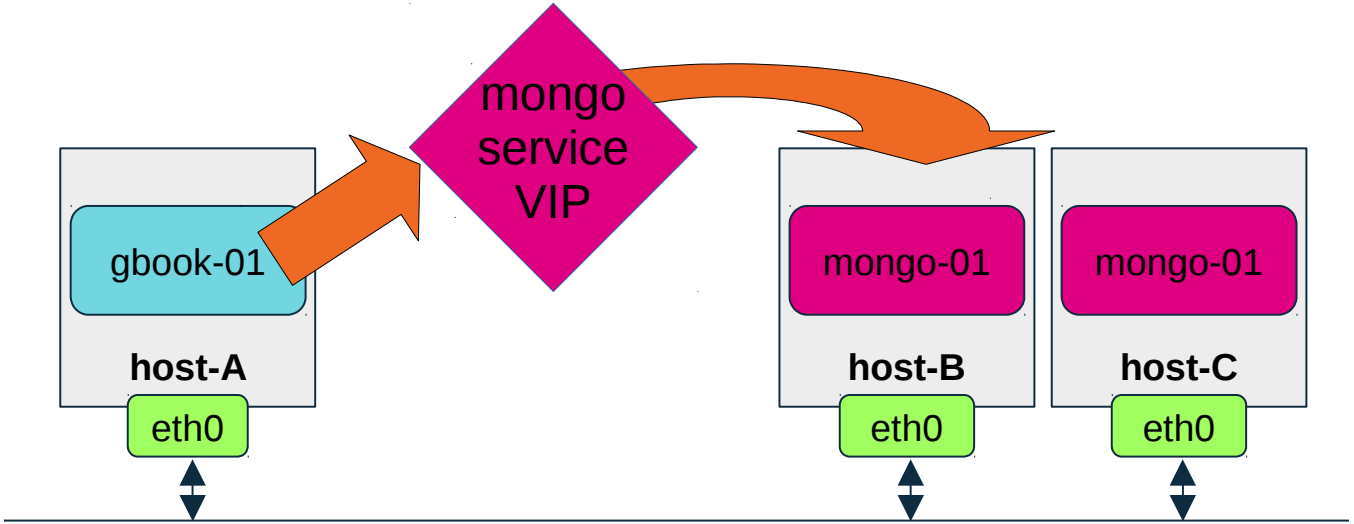
- More instances of the same producer
- How to choose between all of them?

DIY solution

- Use a load balancer
- Point all the consumers to a load balancer
- Expose the producer(s) using the load balancer
- Configure the load balancer to react to changes

→ More moving parts

Kubernetes services



- Service gets a unique and stable Virtual IP Address
- VIP always points to one of the service containers
- Consumers are pointed to the VIP
- Can run in parallel to DNS for legacy applications

Ingress traffic

Challenge #5: publish applications

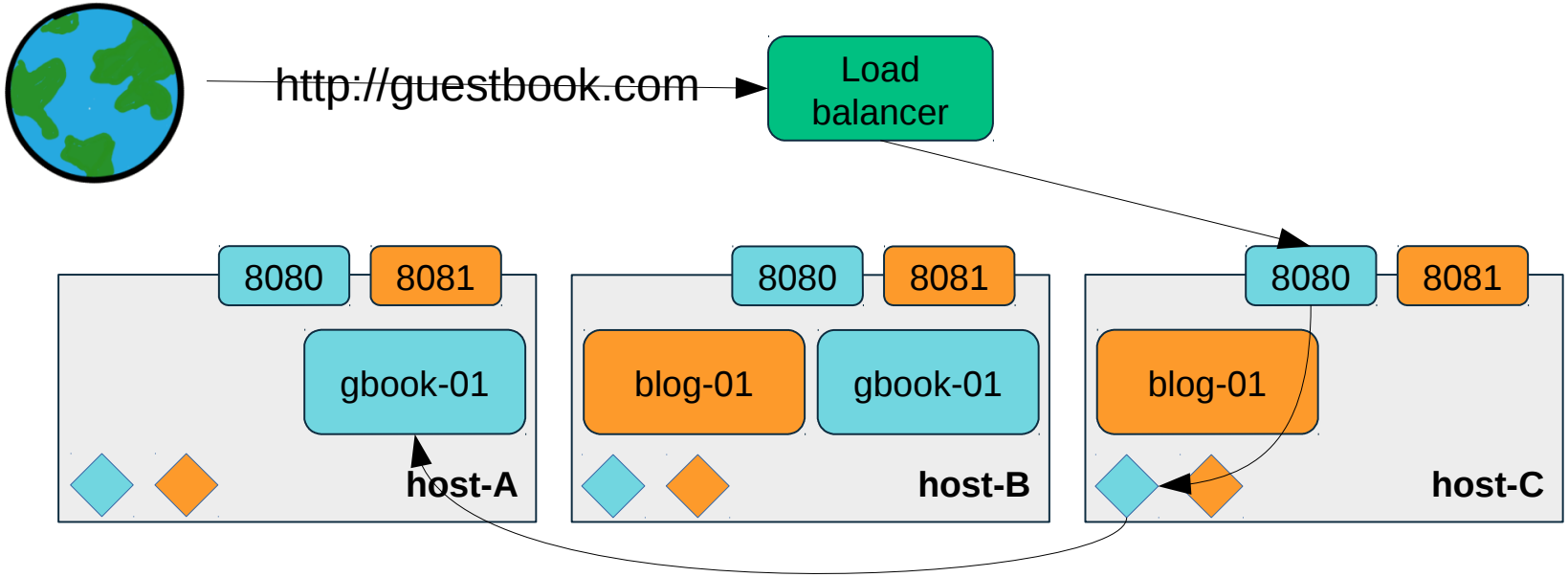
- Your production application is running inside of a container cluster
- How to route customers’ requests to these containers?
- How to react to changes (containers moved, scaling,…)?

Kubernetes’ approach

Services can be of three different types:

- **ClusterIP:** virtual IP reachable only by containers inside of the cluster
- **NodePort**: *“ClusterIP”* + the service is exposed on **all** the nodes of the cluster on a specific port → `<NodeIP>:<NodePort>`
- **LoadBalancer:** *“NodePort”* + k8s allocates a load balancer using the underlying cloud provider. Then it configures it and it keep it up-to-date

Ingress traffic flow



- Load balancer picks a container host
- Traffic is handled by the internal service
- Works even when the node chosen by the load balancer is not running the container

Data persistence

Challenge #6: stateful containers

- Not all applications can offload data somewhere else
- Some data should survive container death/relocation

Kubernetes Volumes

- Built into kubernetes
- They are like resources → scheduler is aware of them
- Support different backends via dedicated drivers:
  - NFS
  - Ceph
  - Cinder
  - ...

Demo

Questions?


SUSE
We adapt. You succeed.