Aalto University
School of Electrical
Engineering

Systems of multiple (micro)services

7.3.2019

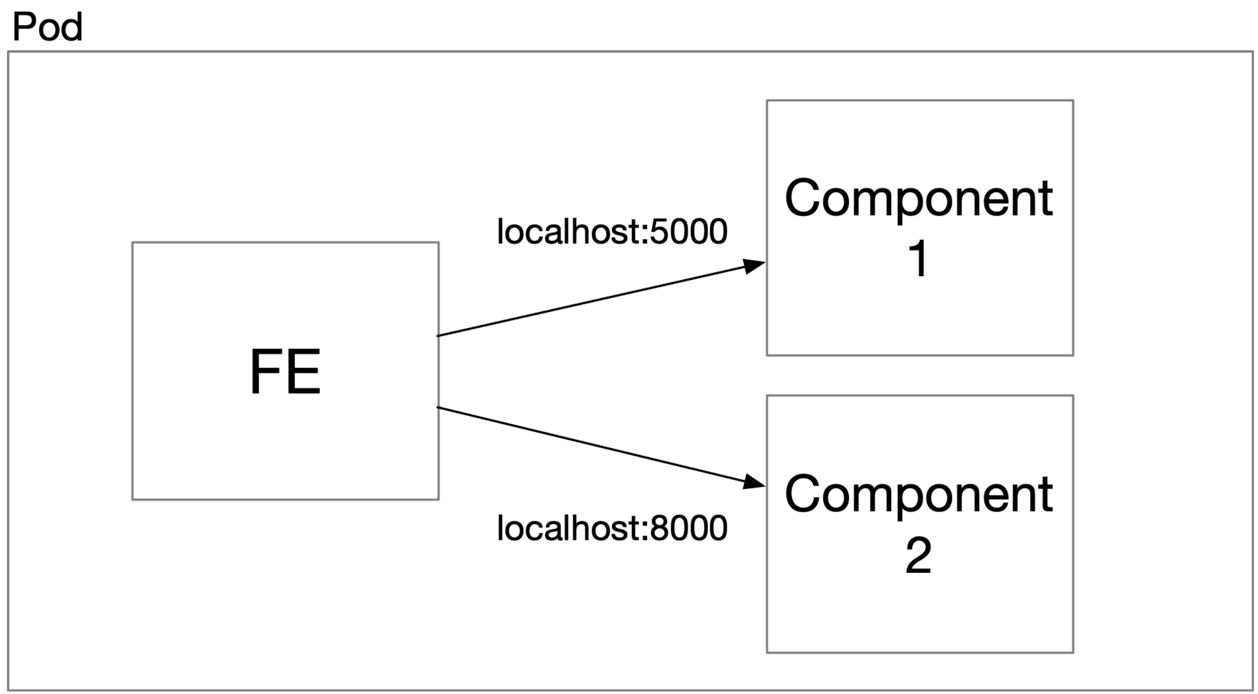
Santeri Paavolainen

What we've covered already

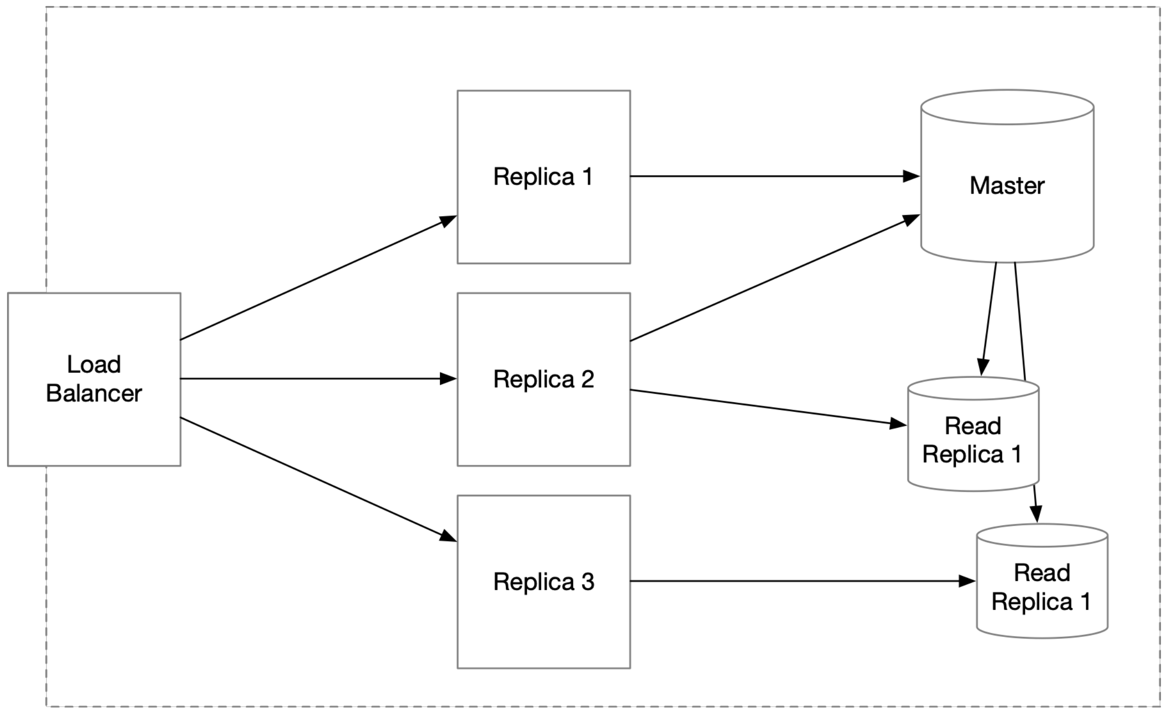
- **Microservices architecture**
  - Boundary, service definition, interfaces, SLA, …
- **Distributed services**
  - Network communication models and protocols (somewhat)
  - Load balancing, asynchronous processing patterns etc.

But think about all the plumbing?

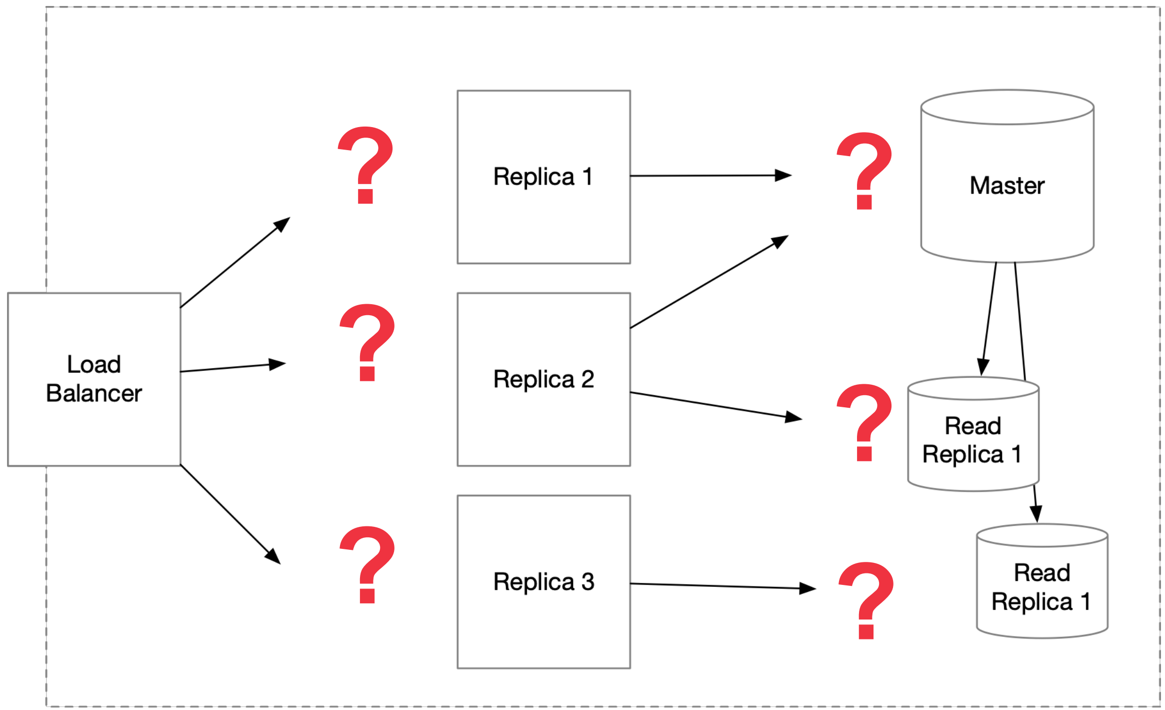
Single pod, multiple containers



Independent components



Independent components



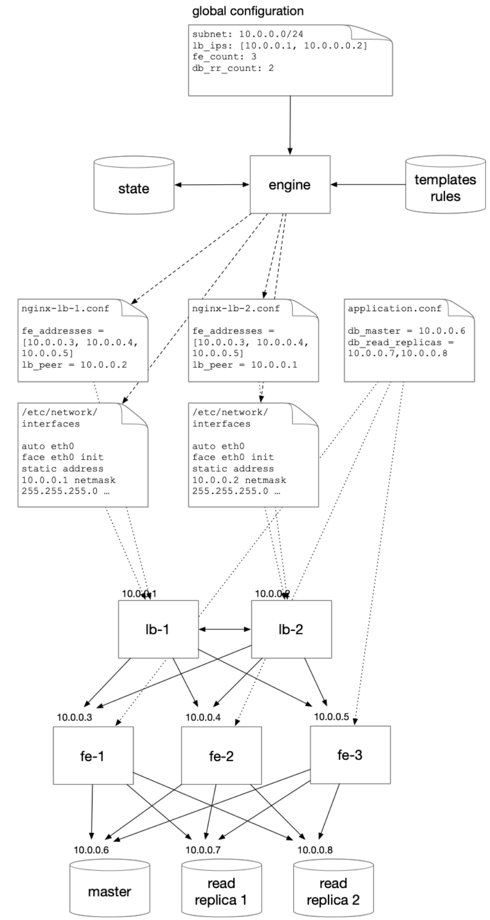
Service Discovery

Service discovery

- **Idea: Map a <u>name</u> to an service <u>address</u>**
- **Questions**
  - Static or dynamic set of hosts?
  - Continuous discovery or done only once?
  - What kind of address? (IPv4, URL, queue name, …)
  - What kind of name? (structured, unstructured, notation)
  - Resiliency and fault-tolerance? (at client, during discovery, by service)
- **Service injection, environment, hosts, DNS, discovery services, directory services**

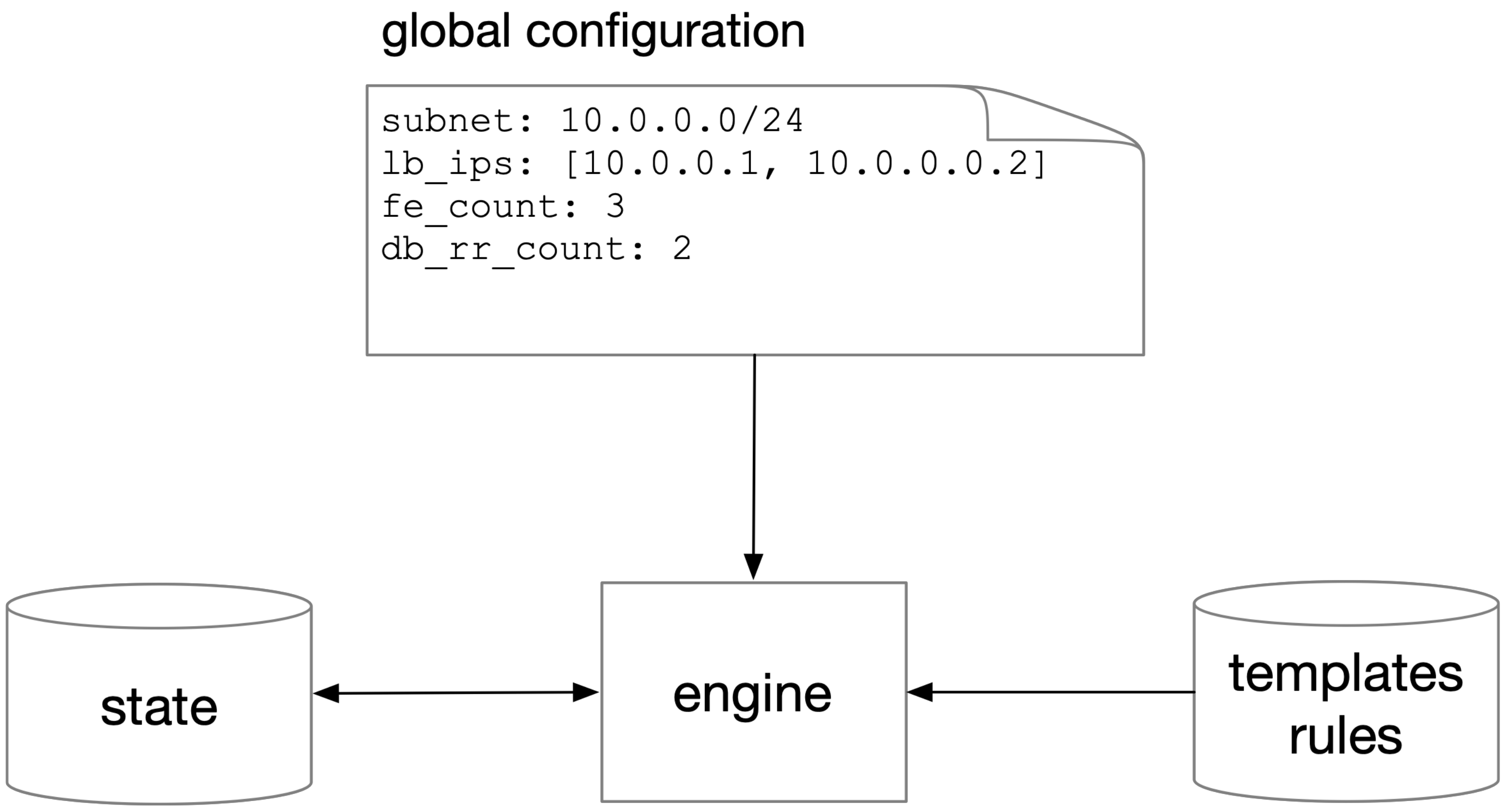
Service injection

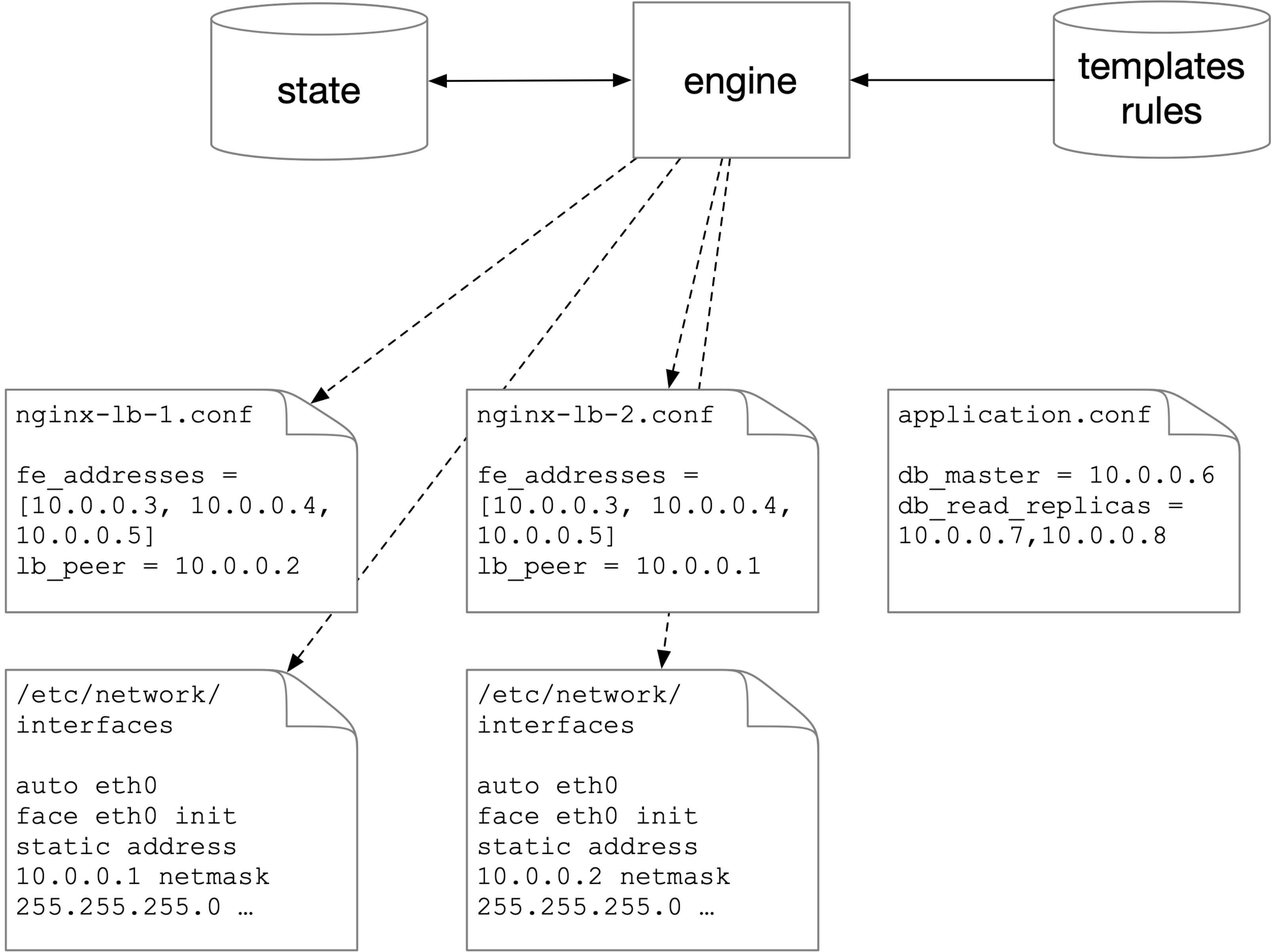
- **Extremely static case of discovery**
- **<u>Everything</u> is known at deployment**
  - IP addresses
  - Number of nodes in cluster
- **Methods**
  - Configuration templates
  - Environmental variables



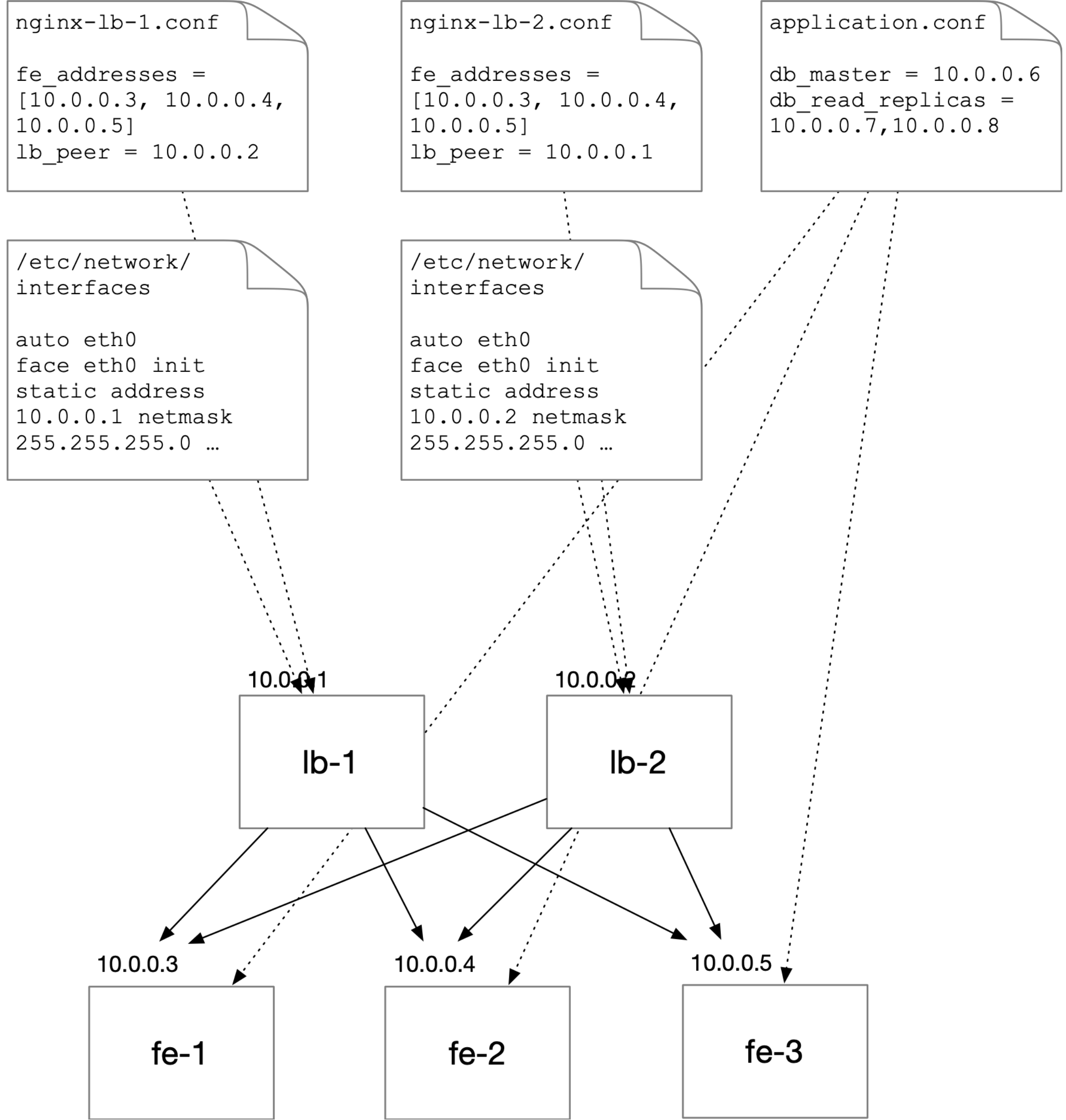
global configuration

```
subnet: 10.0.0.0/24
lb_ips: [10.0.0.1, 10.0.0.0.2]
fe_count: 3
db_rr_count: 2
```

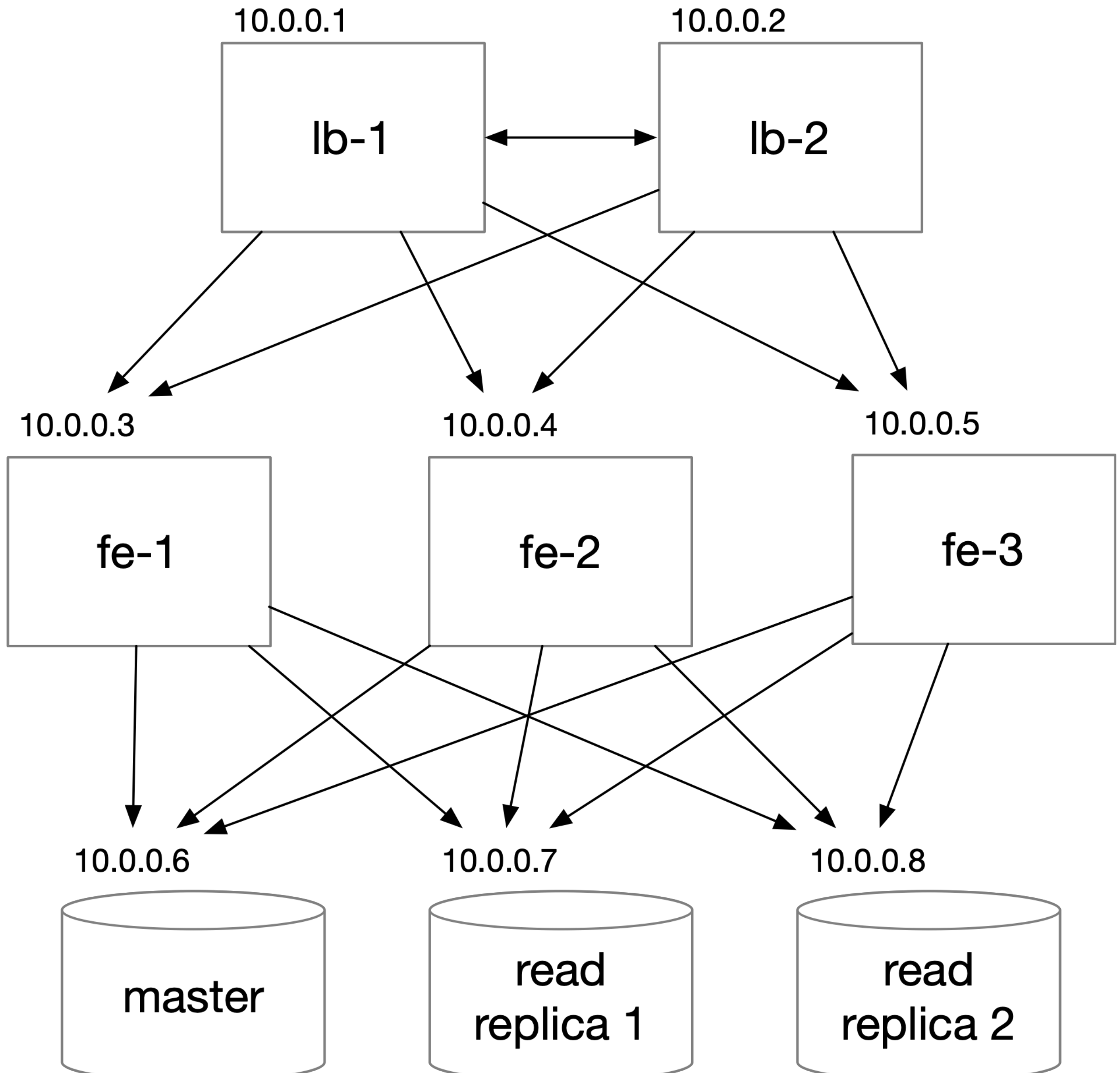

global configuration
subnet: 10.0.0.0/24
lb_ips: [10.0.0.1, 10.0.0.0.2]
fe_count: 3
db_rr_count: 2
state
engine
templates
rules



state
engine
templates
rules
nginx-lb-1.conf
fe_addresses =
[10.0.0.3, 10.0.0.4,
10.0.0.5]
lb_peer = 10.0.0.2
nginx-lb-2.conf
fe_addresses =
[10.0.0.3, 10.0.0.4,
10.0.0.5]
lb_peer = 10.0.0.1
application.conf
db_master = 10.0.0.6
db_read_replicas =
10.0.0.7,10.0.0.8
/etc/network/
interfaces
auto eth0
face eth0 init
static address
10.0.0.1 netmask
255.255.255.0 …
/etc/network/
interfaces
auto eth0
face eth0 init
static address
10.0.0.2 netmask
255.255.255.0 …



nginx-lb-1.conf
fe_addresses =
[10.0.0.3, 10.0.0.4,
10.0.0.5]
lb_peer = 10.0.0.2
nginx-lb-2.conf
fe_addresses =
[10.0.0.3, 10.0.0.4,
10.0.0.5]
lb_peer = 10.0.0.1
application.conf
db_master = 10.0.0.6
db_read_replicas =
10.0.0.7,10.0.0.8
/etc/network/
interfaces
auto eth0
face eth0 init
static address
10.0.0.1 netmask
255.255.255.0 …
/etc/network/
interfaces
auto eth0
face eth0 init
static address
10.0.0.2 netmask
255.255.255.0 …
10.0.0.1
lb-1
10.0.0.2
lb-2
10.0.0.3
fe-1
10.0.0.4
fe-2
10.0.0.5
fe-3


10.0.0.1
lb-1
10.0.0.2
lb-2
10.0.0.3
fe-1
10.0.0.4
fe-2
10.0.0.5
fe-3
10.0.0.6
master
10.0.0.7
read replica 1
10.0.0.8
read replica 2

Recap: Service injection

Not worth the hassle

(OTOH, most of the tools useful for infrastructure management, but not for service discovery unless static)

- **Very simple and straightforward**
- **Common tools:**
  - Fabric, Chef, Puppet, Terraform, AWS CloudFormation
  - Just plain manual configuration also type of "service injection" (human doing the work)
- **Problems**
  - Manual and dynamic changes (some tools work better than others)
  - Failures have to be handled at application level (no automatic reconfiguration)

Environmental variables

Avoid for discovery

- **Docker Compose (obsoleted), Kubernetes (not recommended) etc. have this**
  - Remember `docker run -e VAR=VALUE` ?
- **Almost like service injection**
  - Can change when service restarted (with rolling restarts starts resembling dynamic discovery)
- **Docker Compose:**
  ```
  FE_1_PORT_80_TCP_ADDR=10.0.0.3
  DB_MASTER_PORT_5432_TCP_ADDR=10.0.0.6
  ```
- **Kubernetes:**
  ```
  DB_MASTER_SERVICE_HOST=10.0.0.6
  DB_MASTER_SERVICE_PORT=5432
  ```
- **Use these in configuration file (via environment) or command line**

Host-based discovery

- **Idea: Distributed services over network**
  ➔ DNS built-in to almost everywhere ➔ why not use it?
- **Host-based discovery**
  - /etc/hosts (static = old, since dynamic mounts or rewriting)
  - Local DNS resolver
  - Cluster DNS
  - Integrated service discovery service with DNS

Host-based discovery variants

- **Static singular records**
  - ```
    fe_addresses = [fe-1, fe-2, fe-3]
    lb_peer = lb-2
    ```
- **Multiple records**
  - ```
    fe_addresses = fe
    lb_peer = lb-2
    ```
  - DNS
    ```
    fe.local. IN A 10.0.0.3
    fe.local. IN A 10.0.0.4
    fe.local. IN A 10.0.0.5
    ```
  - Either randomized ordering on DNS response or client-side support for multiple records
- **Search domain (often .local)**
  - Can have structure too
  - *svc.ns*.svc.cluster.local (Kubernetes)
- **Cloud infrastructure services use CNAME indirection**
  - RR load balancing at DNS level (see next page)

Example: CNAME forwarding to changing A records in AWS S3 (RR LB)

```
$ dig energysim.kooma.net
;energysim.kooma.net.         IN    A
energysim.kooma.net.    1340  IN    CNAME energysim.kooma.net.s3-website-eu-west-1.amazonaws.com.
energysim.kooma.net.s3-website-eu-west-1.amazonaws.com.     12 IN CNAME s3-website-eu-west-1.amazonaws.com.
s3-website-eu-west-1.amazonaws.com. 2 IN A      54.231.134.140
```

```
$ dig energysim.kooma.net
;energysim.kooma.net.         IN    A
energysim.kooma.net.    1333  IN    CNAME energysim.kooma.net.s3-website-eu-west-1.amazonaws.com.
energysim.kooma.net.s3-website-eu-west-1.amazonaws.com.     5 IN CNAME s3-website-eu-west-1.amazonaws.com.
s3-website-eu-west-1.amazonaws.com. 5 IN A      52.218.104.108
```

More on host-based discovery via DNS

- **Single vs. multiple records vs. RR LB vs. intelligent DNS**
  - Plus region affinity, fallbacks etc. (can go really deep)
- **Namespaces**
  - Split by organization or function?
- **Split DNS**
  - Different names on internal and external (or more) sides
    - *service.local inside ➔ IN A 1.2.3.4, 2.3.4.5, 3.4.5.6*
service.local outside ➔ IN CNAME external-lb…

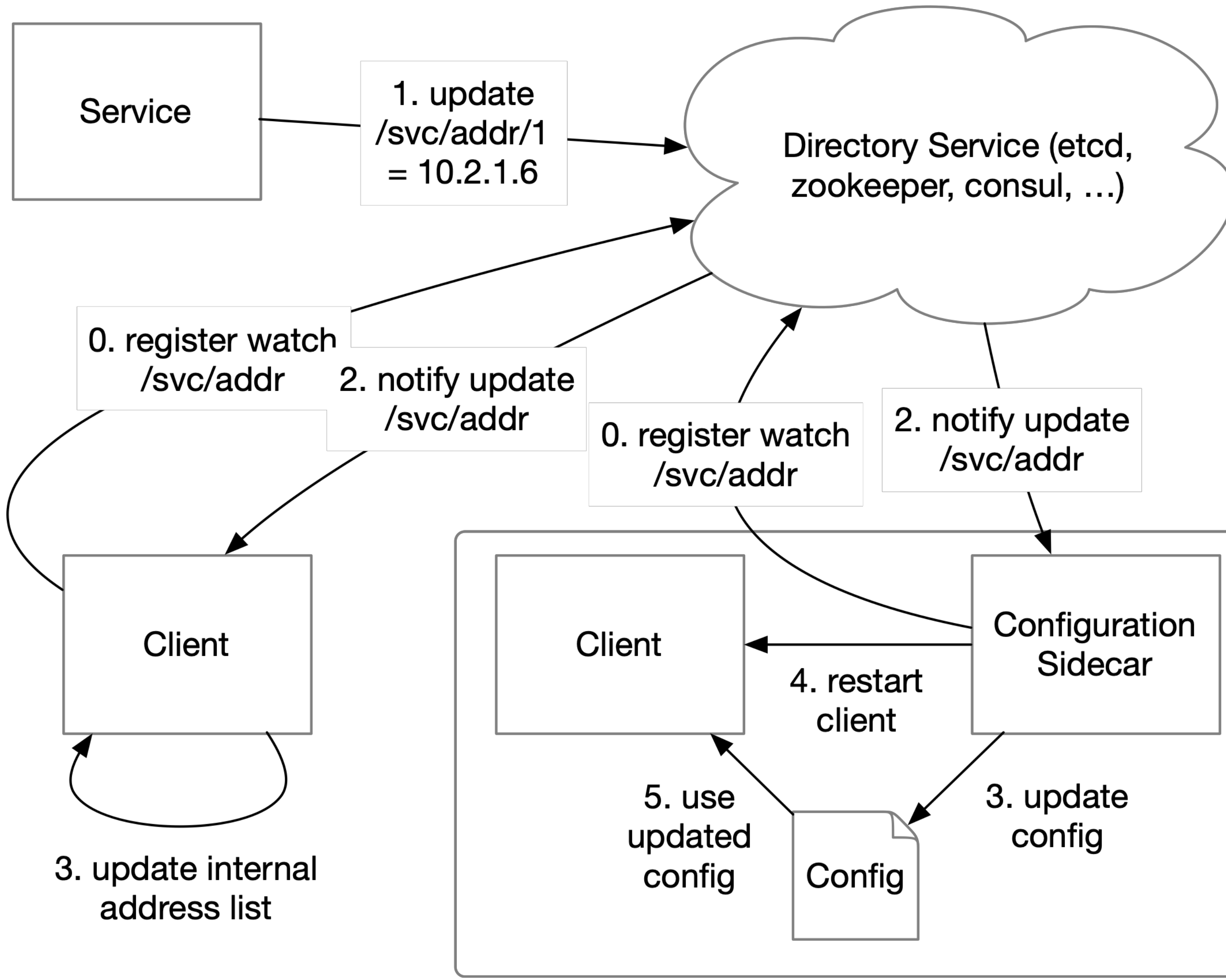
Kubernetes: host-based nginx reverse proxy with 2 services

- **nginx as reverse proxy**
  - / → "hello world" app
  - /static/ → static file server
- **Three services**
  - 2 x internal (no public IP) but visible to nginx
  - 1 x external (public IP) to nginx
  - Public → nginx → (hello world app | static)
- **Name resolution**
  - kube-dns most likely… (may be CoreDNS too)

More service discovery patterns

- **Host-based via DNS easy to use**
  - Might require client-side understanding (multiple records)
  - Difficult to generalize to other uses (queue names etc.)
  - (Ports not so much a problem with private IPs and port remapping)

➔ **Generalized directory services**
  - etcd, ZooKeeper, Consul …
  - <u>Requires</u> client-side support: external configurator (sidecar?) or internal to application (integrate service client)
  - Users have complete control over key and value semantics



Service
1. update
/svc/addr/1
= 10.2.1.6
Directory Service (etcd,
zookeeper, consul, …)
0. register watch
/svc/addr
2. notify update
/svc/addr
0. register watch
/svc/addr
2. notify update
/svc/addr
Client
3. update internal
address list
Client
Configuration
Sidecar
4. restart
client
5. use
updated
config
3. update
config
Config

Summary

- **Name-based service discovery a practical requirement in microservice architectures**
  - Dynamic environment
- **Host-based resolution (DNS) very useful**
  - Usually built-in to many container orchestration environments (Kubernetes) → easy to use
  - Can be adapted to more advanced use cases (RR IN A LB etc.)
- **Sometimes generalized directory services needed**
  - DNS not a good match for non-host resolution (queue names etc.)

Failures

Overview

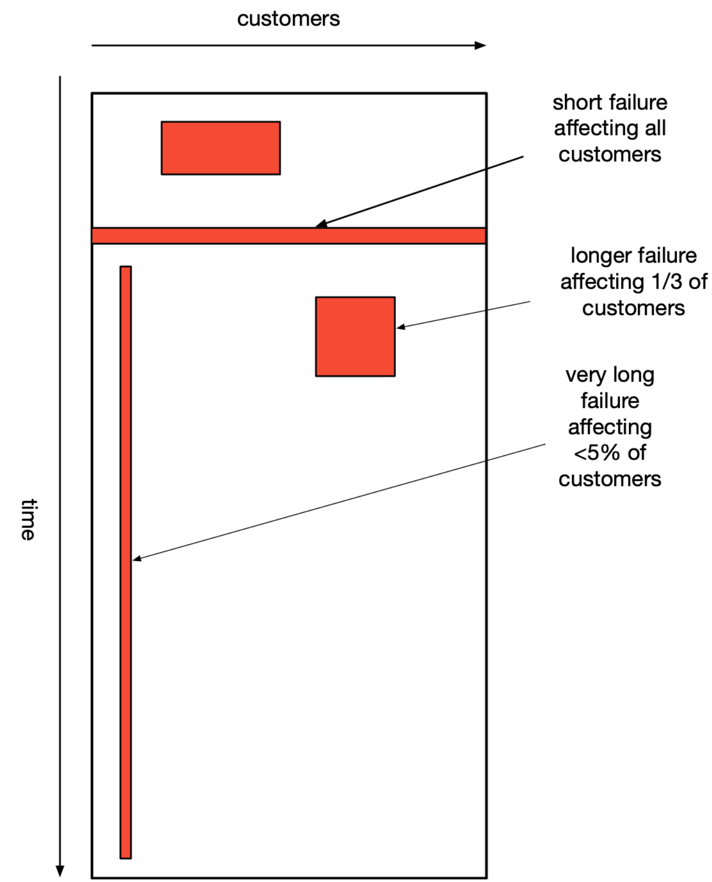
- **Already established that in a distributed system**
  - Services may fail at any time
  - Network may fail at any time
  - Services may delay response arbitrarily
  - Network may hang up arbitrarily
  - Services may produce unexpected responses for any request
  - (Client may fail at any time too, but let's ignore that for now)
- **What can we do about that?**

Concepts

- **“Blast radius”**
  - How large effect?
- **Failure types**
  - Can we roughly categorize different failure types?
- **Recovery vs. remediation (triage)**
  - Short-term vs. long-term responses
- **Responses**
  - How to handle?

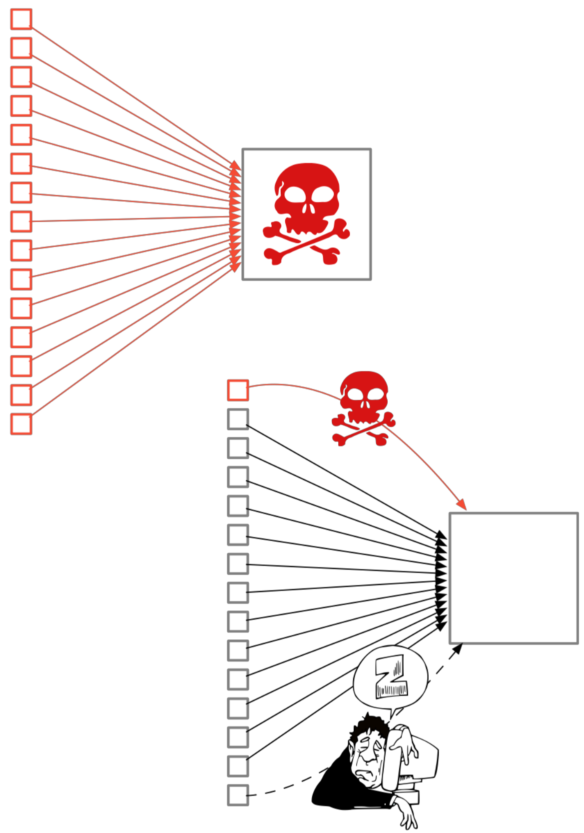
Blast radius

- **"Blast radius"**
  - How large (spatial) and long (temporal) is the effect of a failure?
  - Whole system (all clients, all requests)? Some % of system (clients and requests on a single server or container)? Single request (a client)?
- **Failure types (roughly)**
  - Server dies (application server crashes, server crashes)
  - Request fails (connection terminated, 500 Server Error, incorrect response, corrupted response)
  - Request hangs (response not completed)



Failure types

- **Failure types (roughly)**
  - Server dies (application server crashes, server crashes)
  - Request fails (connection terminated, 500 Server Error, incorrect response, corrupted response)
  - Request hangs (response not completed)
- **May occur in combination**
  - Server dies → may look as a hang to client
  - Server dies → may result in 500 from proxy or a connection termination
  - Request hangs → eventual error response from timeout in proxy



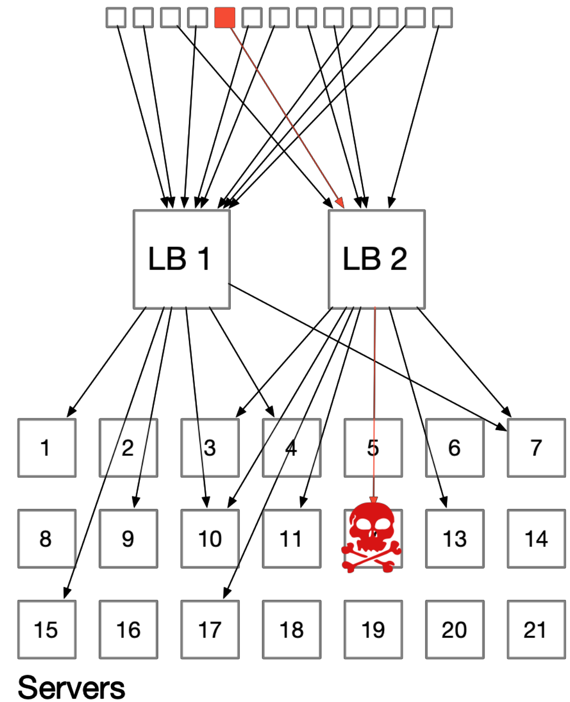
Remediation and recovery

- **Remediation**
  - Immediate response
  - "What can be done now?"
- **Recovery**
  - Long-term responses
  - If this is not a transient problem, can we reduce the likelihood of that happening again?
    - *Automated responses vs. manual responses*

- **Remediation responses**
  - Ignore (e.g. propagate failure)
  - Retry (transparently / at client)
  - Do something other
  - Remove server / container from load balancer / cluster
- **Recovery responses**
  - Wait
  - Increase capacity (more replicas)
  - Alert humans (manual intervention)
  - Systemic fix (aka debug)

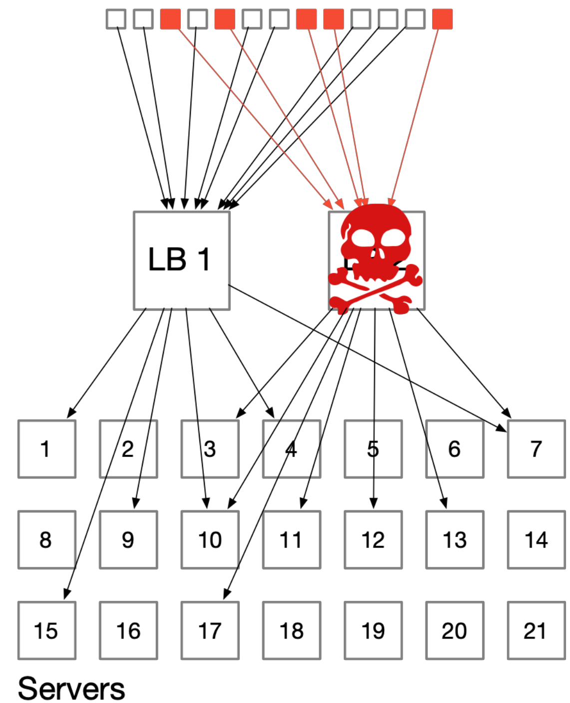
Replicas failing behind LB

- **Stateful**
  - Persistent state → trouble
    - *Has to wait until restarted*
    - *Pray data is not corrupted*
    - *Long recovery time*
  - Cached state → slowdown
    - *Recompute or retrieve cached state on rebalanced nodes*
    - *Potential global service degradation*
- **Generally 1 / N effect**
  - 1 / 21 = ~5%
  - Assumes random placement
- **Health checks on nodes**



Replicas behind failing LB

- **Hidden assumption:**
  LBs are more reliable than services they are fronting
  - What if not valid?
- **1 / N blast radius**
  - N = 2 not unheard of
  - 50% worst case
- **Cascading failures**
  - Remaining LBs must be able to handle increased traffic
  - If N = 2 → 100% increase



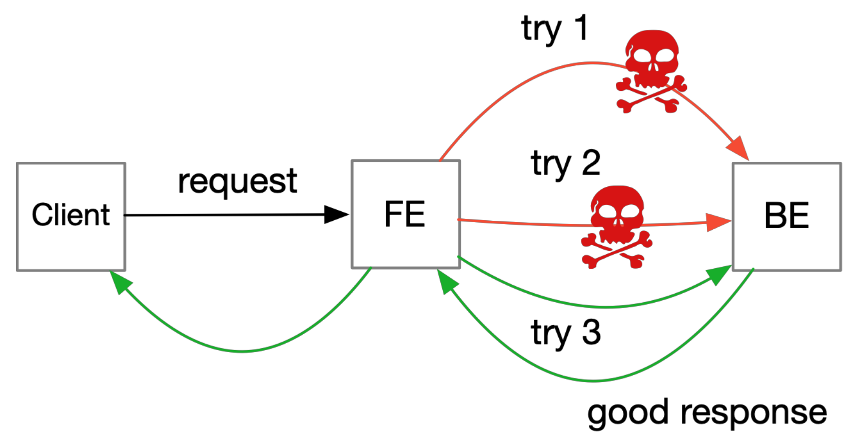
Failing services

- **The overall service may be up**
  - E.g. LB health checks pass (or no LB, and service used via RR IN A record set)
  - Transient or client-specific failure → no rationale for heavy-handed global response
  - May be limited to specific APIs or parameters
- **Timeouts or 5xx failures**

- **Remediation possibilities**
  - Transparent retry
  - Client-side retry
  - Fallbacks
  - Circuit breakers
- **Recovery**
  - If transient, maybe just wait?
  - If can trace to deployment, maybe roll back?
  - If CPU load, the cluster should be autoscaled anyway (so wait)

Transparent retry

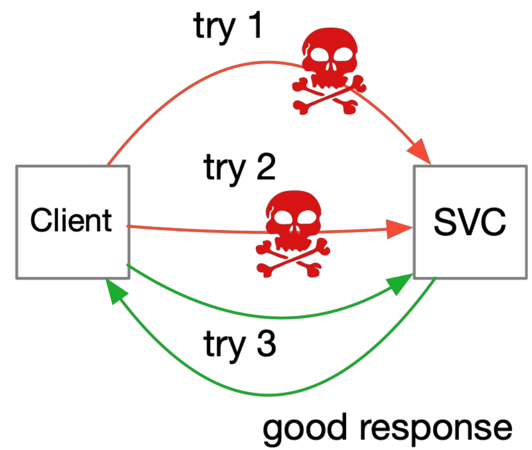
- **Incorporate retry logic at intermediary**
  - LB, reverse proxy, etc.
- **Most useful for transients**
- **Really only for idempotent requests**
  - GET or HEAD
- **Timeouts**
  - N tries with timeout T for each, max N * T seconds (N = 3, T = 10 → 30 seconds before definite failure)
- **Also useful for backoffs (429, 503)**
- **Does not help if clients aggressive**
  - RELOAD if >5s second page load

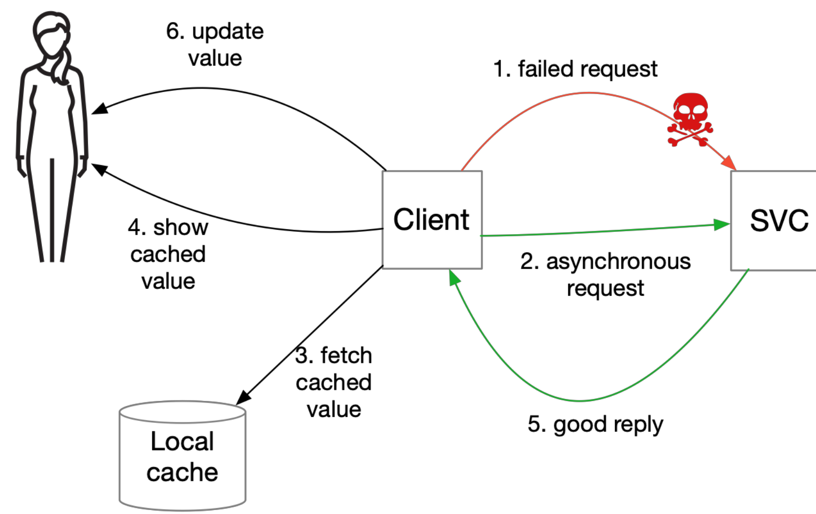


Note: BE may be multiple replicas, with different requests going to different nodes

Client-side retry

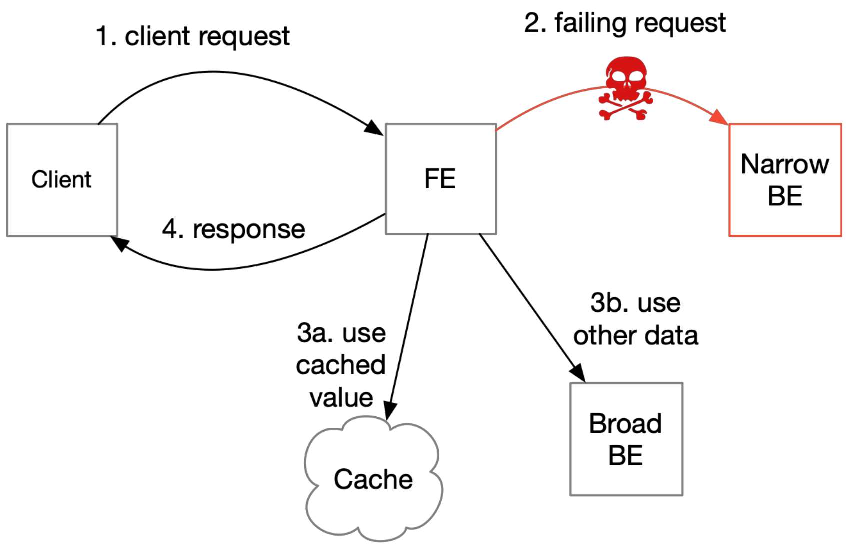
- **Return failures immediately to client**
  - Timeouts controlled by client
- **Client decides what to do**
  - Retry?
  - Use other service?
  - Use cached value?
  - Use cached value, but call asynchronously and update if successful?
  - Report error?





Fallbacks

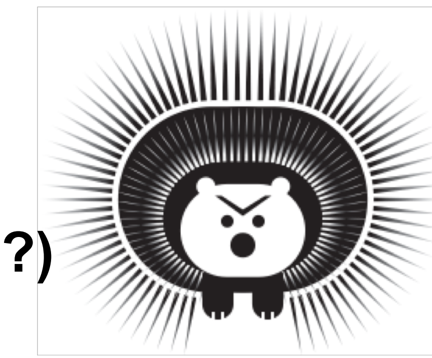
- **Aspect of resilient computing**
  - Adaptive activities and responses based on environmental conditions
- **Use cached or other data**
  - "Old value" often better than "no value"
  - Broader data may be applicable too
    - *Per-user recommendations → general recommendations*
    - *Finland feed → Europe aggregate feed*
- **Applicable for all**
  - Services using other services, transparent proxies and client-side logic

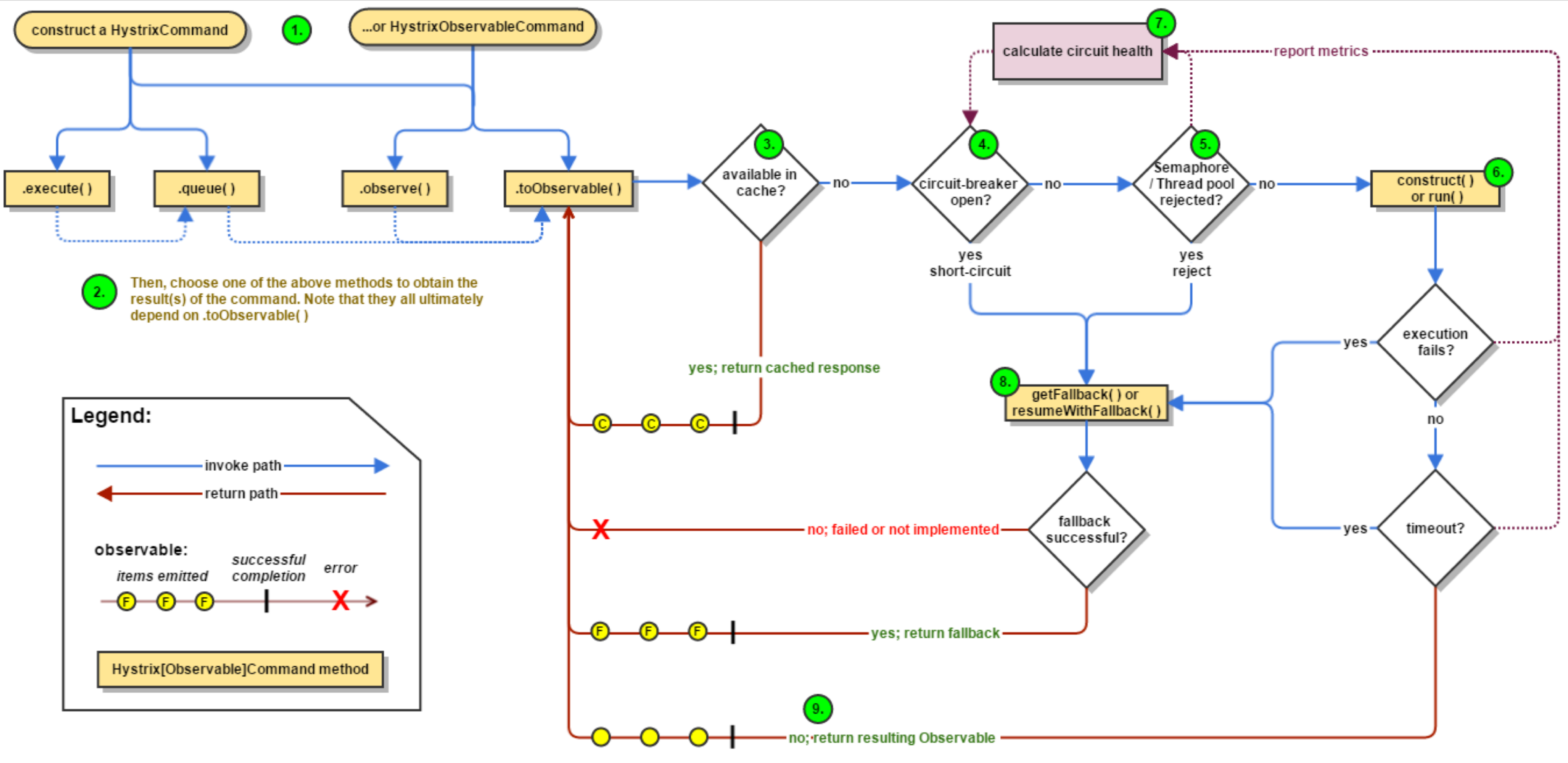


We can do better!

Circuit breakers (software fuses)

- **Distributed system = <u>many parties</u>**
  - Share information about failures
  - Clients can react to failing services before using them
- **Circuit breaker**
  - Trip on failures
  - Use fallback if tripped
  - Some fuse reset policy
- **Hystrix! (originating from Netflix, where else?)**
  - <u>Circuit breaker</u> design pattern





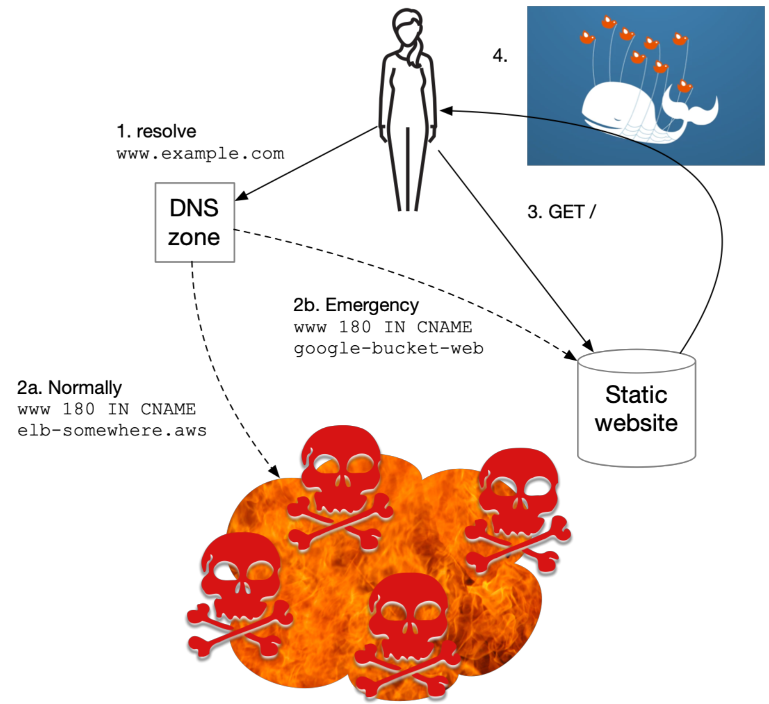
Source: Hystrix / Netflix

Summary

Failure	Blast radius	Remediation	Recovery
Node dies	All clients on the node	Load balance to another node	Reboot or launch new
Request fails	Single request	Retry Fallback	(Debug)
Slow request processing	Single request	**<u>Timeout!</u>** Then retry or fallback	Increase processing capacity (Debug)
Service failure	Variable	Circuit breaker	Wait Increase capacity

Finally …

- **You can not hide all failures from clients**
  - If client is intelligent, it can often do something else
  - Transparent retry? Eventually retries fail → use cached value – what if no cached value? FE fails during sending response? Etc. etc.
- **“World is burning” scenario**
  - Trying desperately not to show “Unreachable site” page
  - Fail page hosted <u>elsewhere</u> ready, redirect at DNS level



Summary of summaries

- **Plumbing services together: Service Discovery**
  - Host name based
  - Directory services (client-side support)

- **What to do when plumbing leaks: Remediation and recovery**
  - Retries
  - Fallbacks
  - Circuit breakers