A CRASH COURSE IN CRUSH

Sage Weil
Ceph Principal Architect
2016-06-29

OUTLINE

- Ceph
- RADOS
- CRUSH functional placement
- CRUSH hierarchy and failure domains
- CRUSH rules
- CRUSH in practice
- CRUSH internals
- CRUSH tunables
- Summary

CEPH: DISTRIBUTED STORAGE

- Object, block, and file storage in a single cluster
- All components scale horizontally
- No single point of failure
- Hardware agnostic, commodity hardware
- Self-manage whenever possible
- Open source (LGPL)

CEPH COMPONENTS

OBJECT	BLOCK	FILE
RGW A web services gateway for object storage, compatible with S3 and Swift	**RBD** A reliable, fully-distributed block device with cloud platform integration	**CEPHFS** A distributed file system with POSIX semantics and scale-out metadata management

LIBRADOS

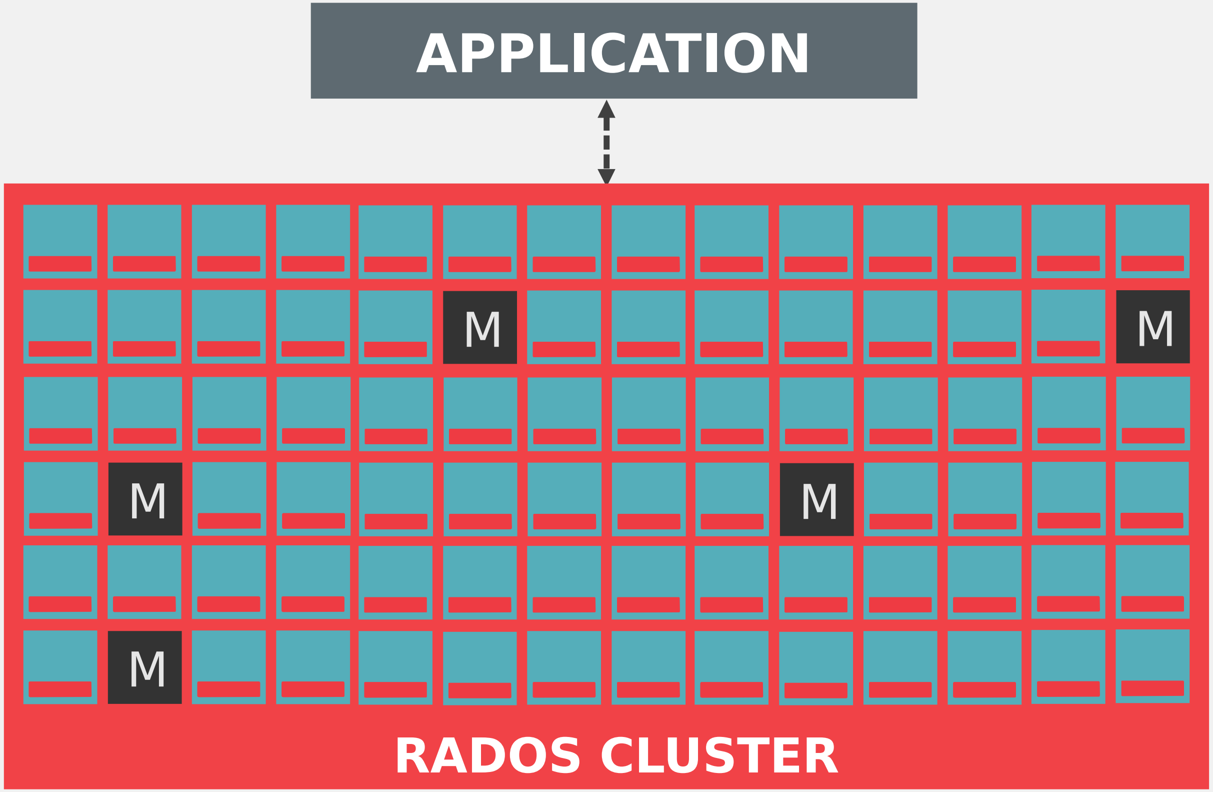
A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

RADOS

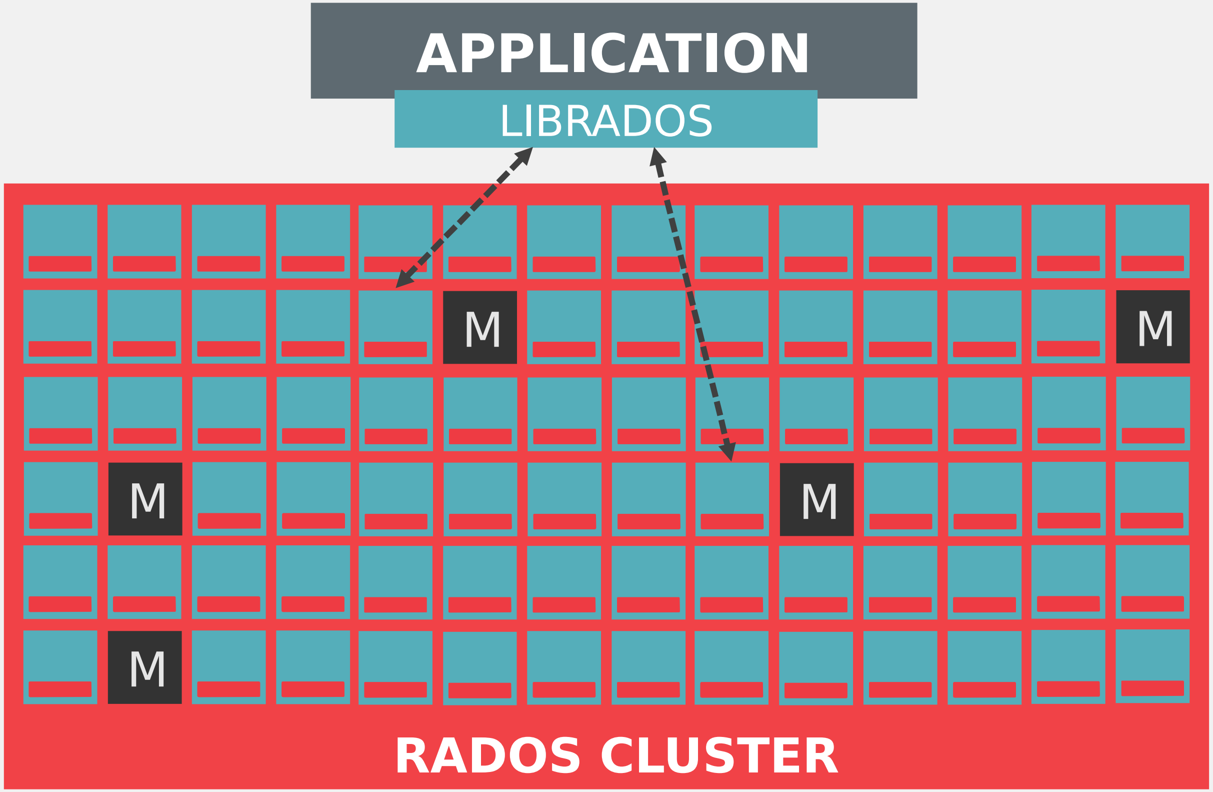
A software-based, reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes and lightweight monitors

RADOS

RADOS CLUSTER



RADOS CLUSTER



CEPH DAEMONS

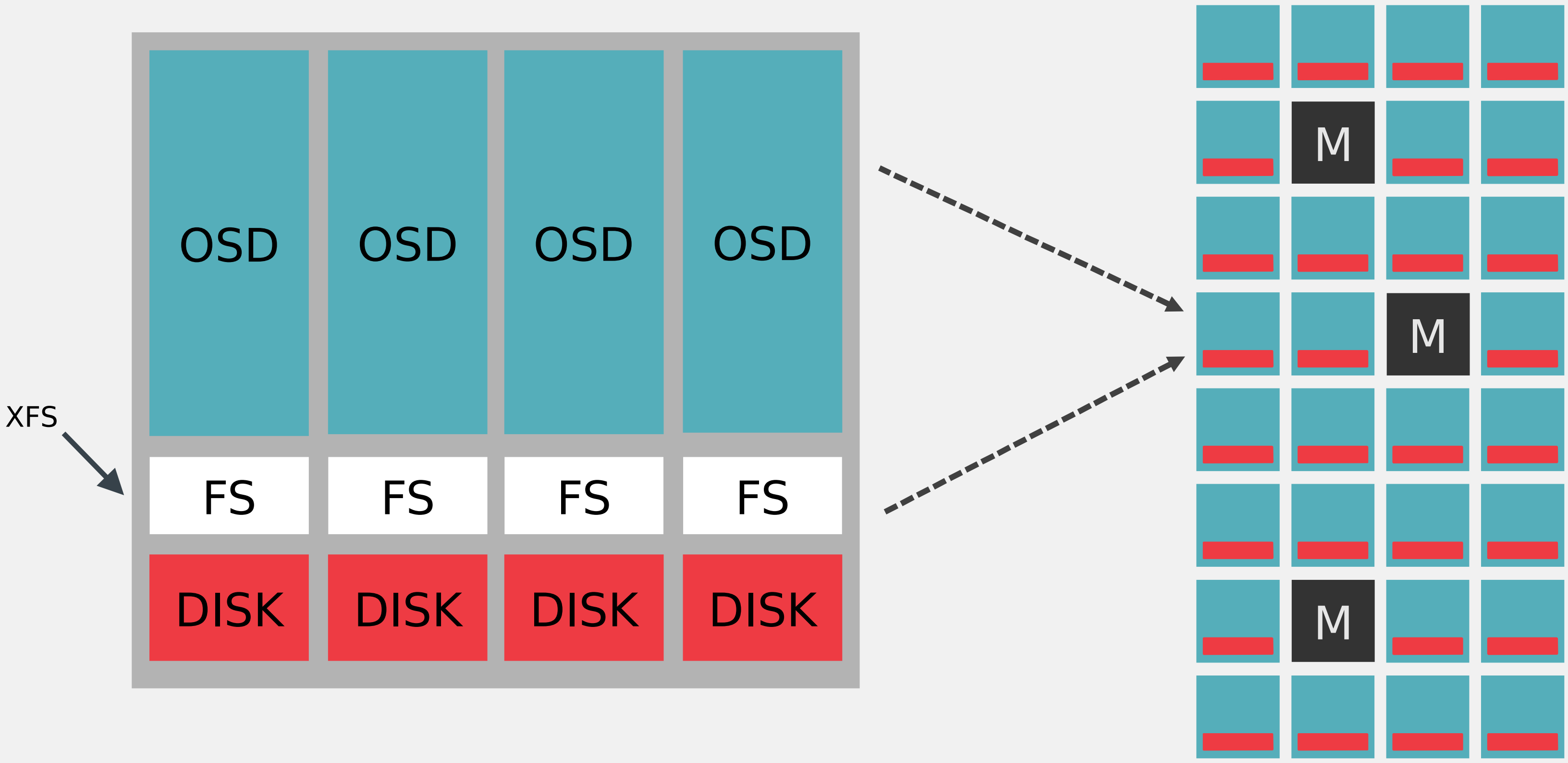
OSD

- 10s to 1000s per cluster
- One per disk (HDD, SSD, NVMe)
- Serve data to clients
- Intelligently peer for replication & recovery

Monitor

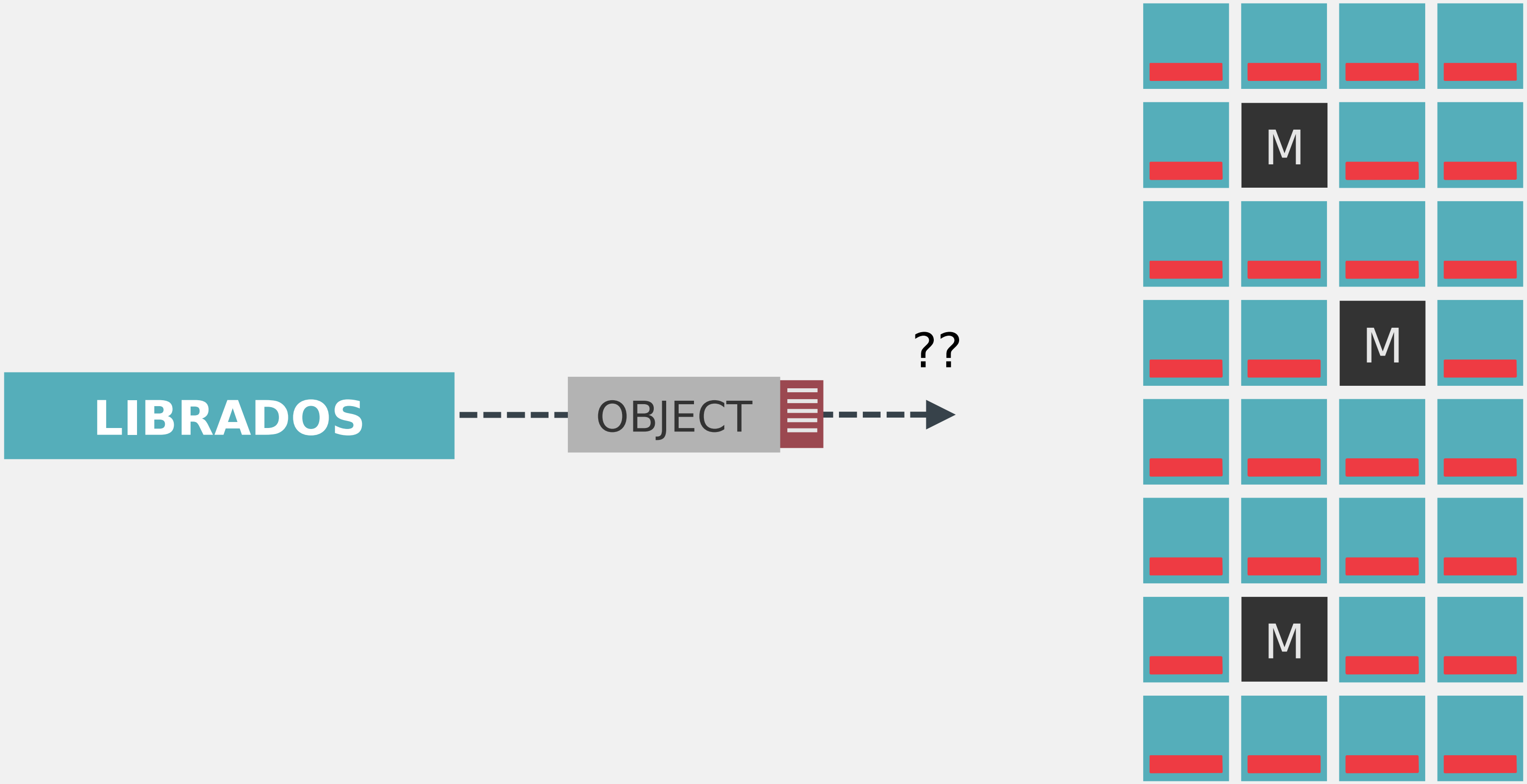
- ~5 per cluster (small odd number)
- Maintain cluster membership and state
- Consensus for decision making
- Not part of data path

MANY OSDS PER HOST

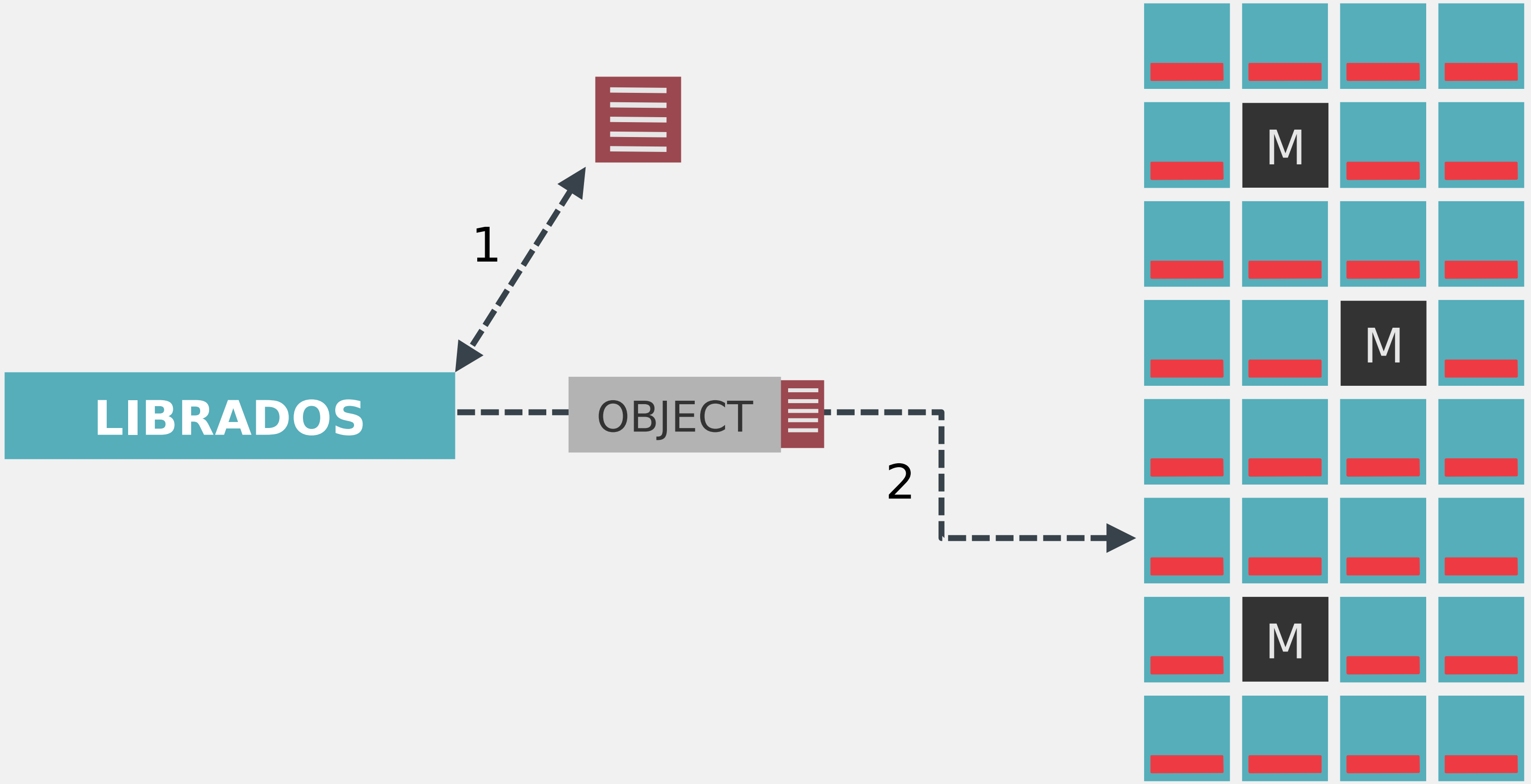


DATA PLACEMENT

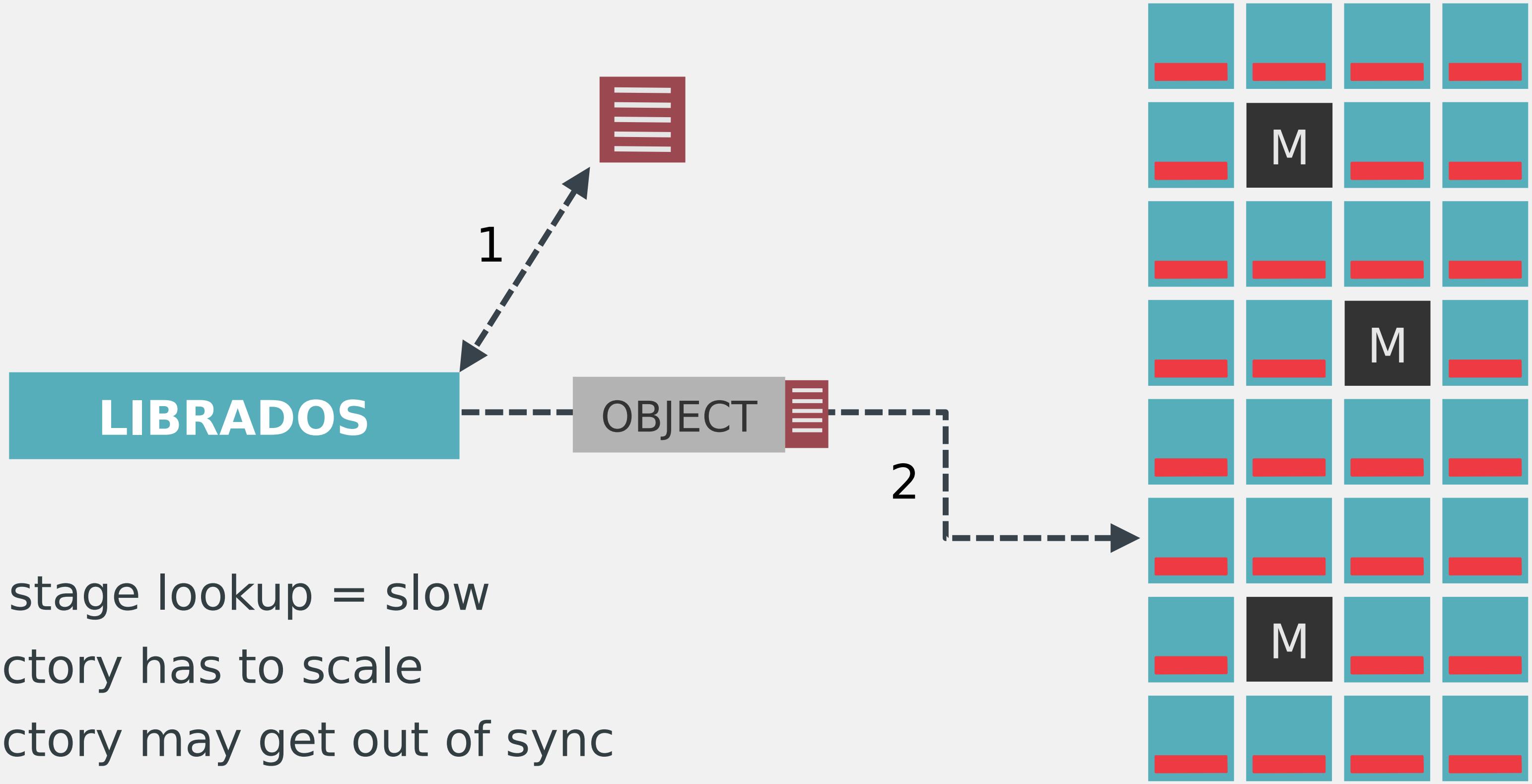
WHERE DO OBJECTS LIVE?



MAINTAIN A DIRECTORY?

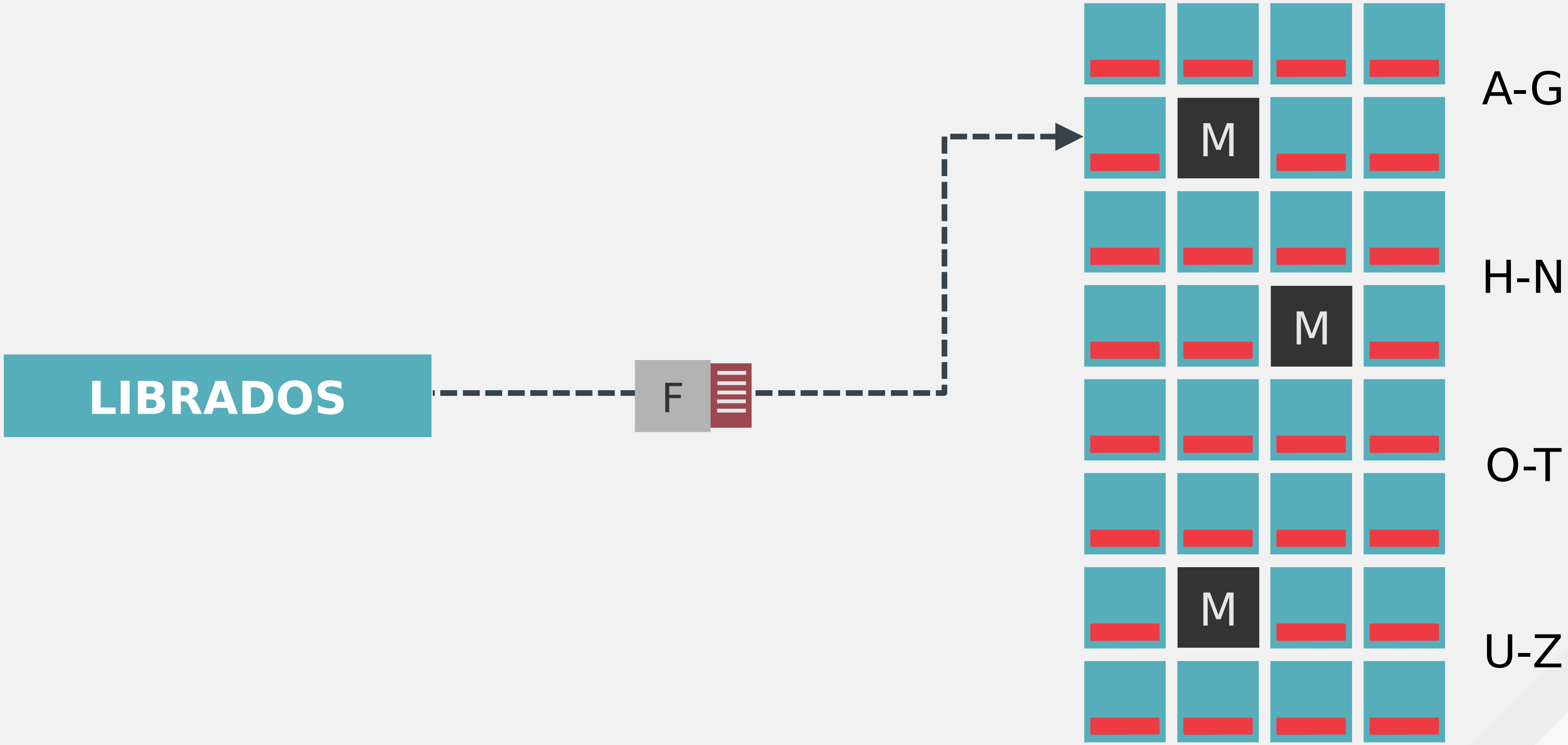


MAINTAIN A DIRECTORY?

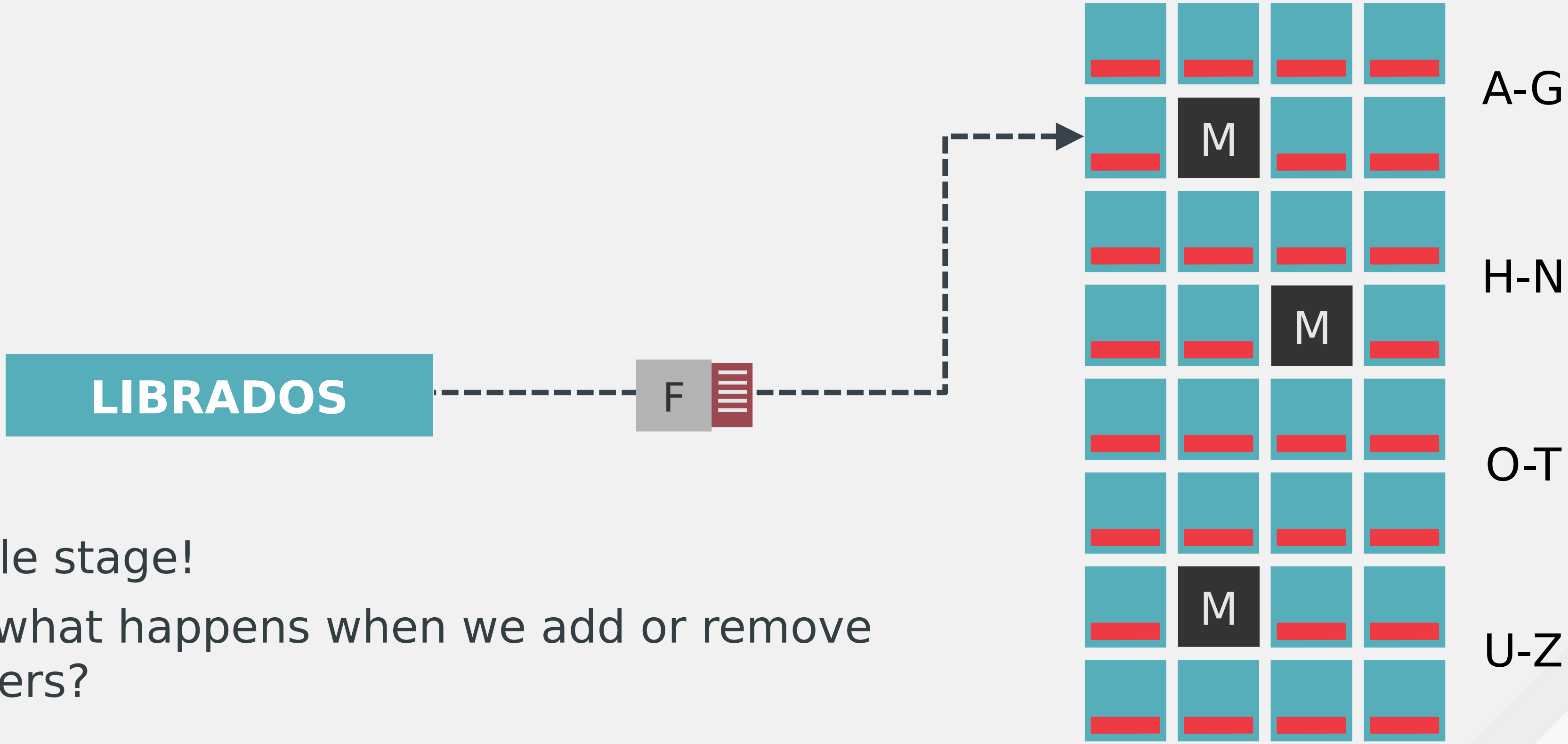


- Two stage lookup = slow
- Directory has to scale
- Directory may get out of sync

CALCULATED PLACEMENT

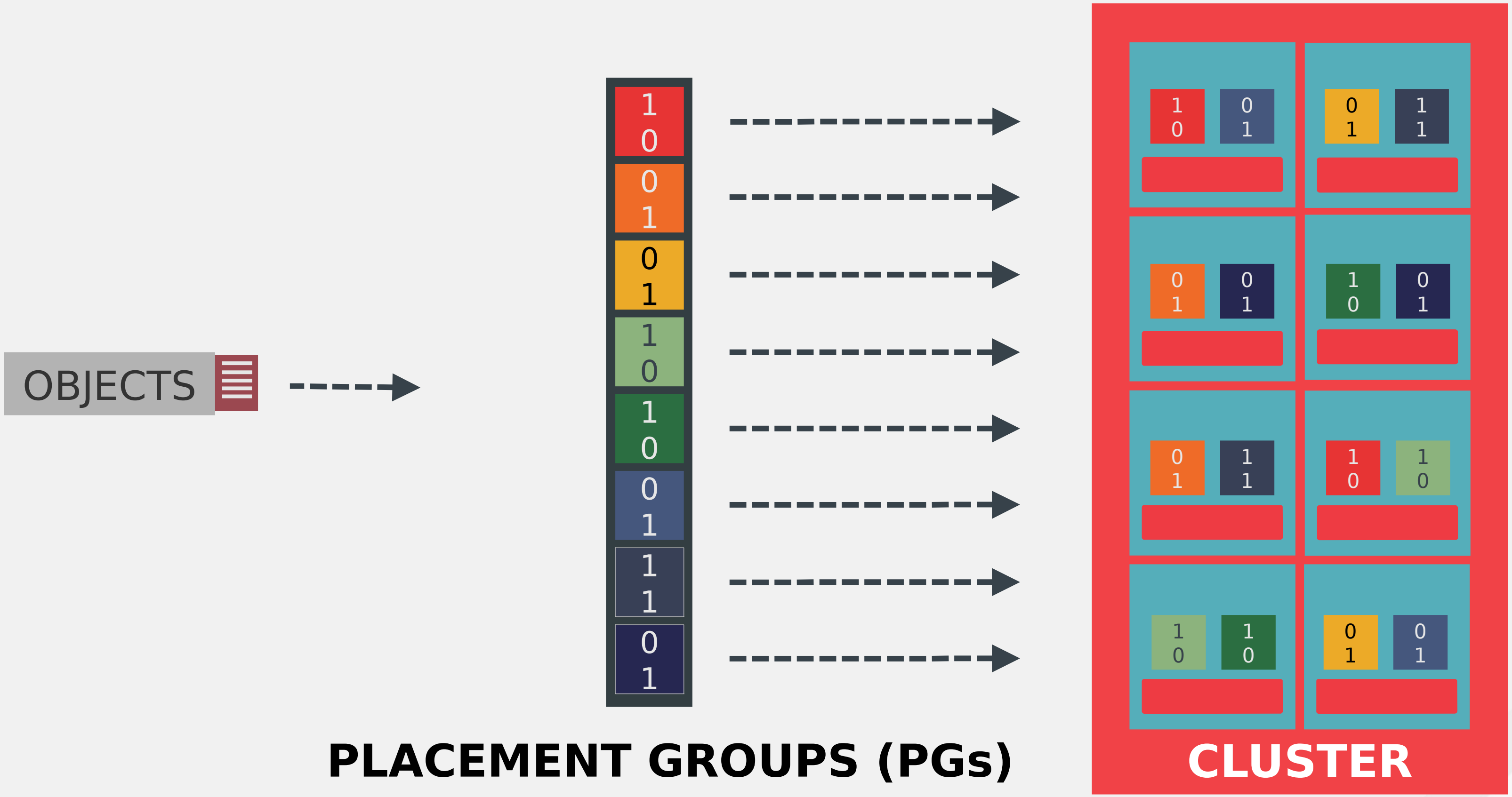


CALCULATED PLACEMENT



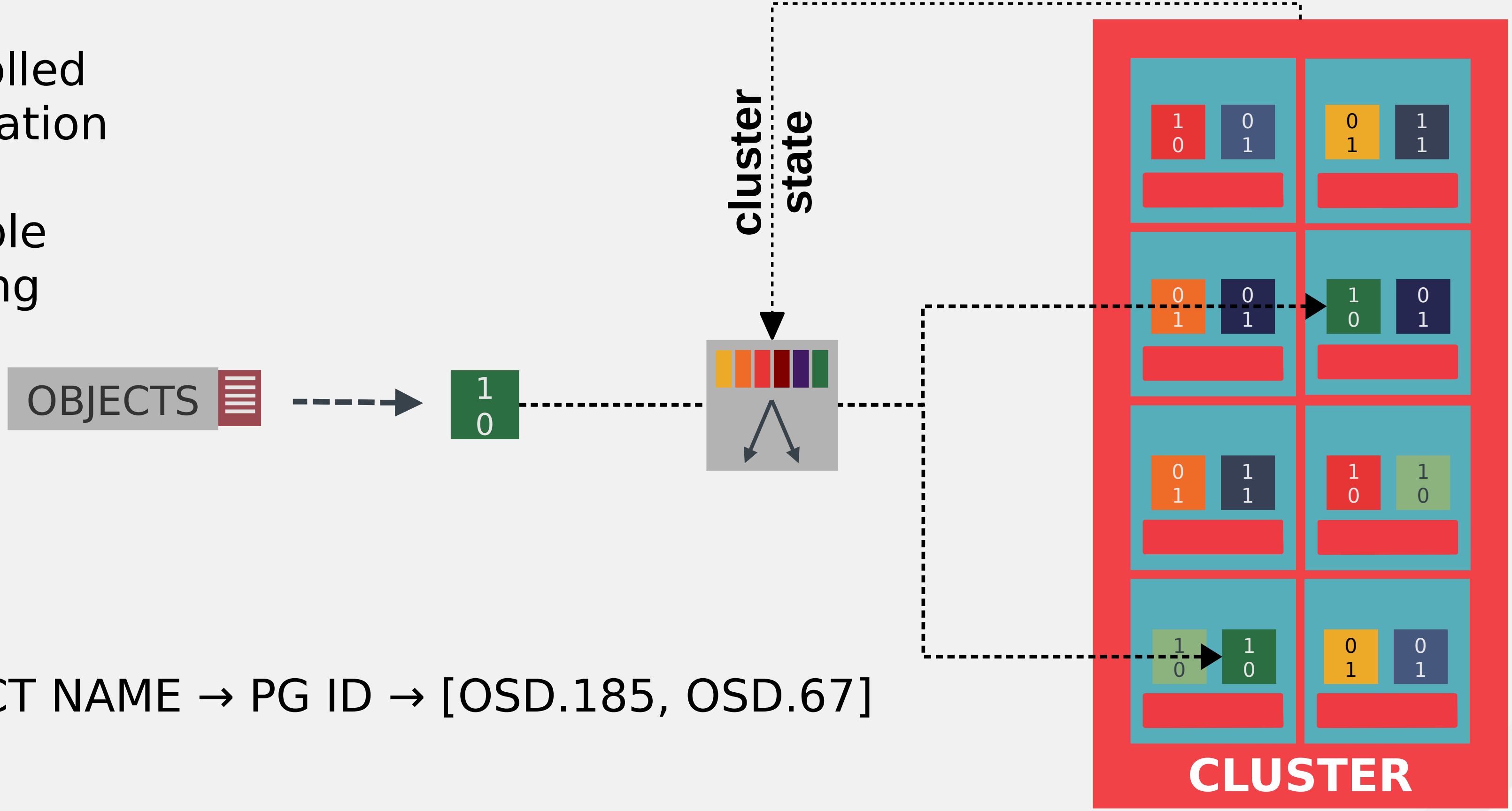
- Single stage!
- But what happens when we add or remove servers?

TWO STEP PLACEMENT



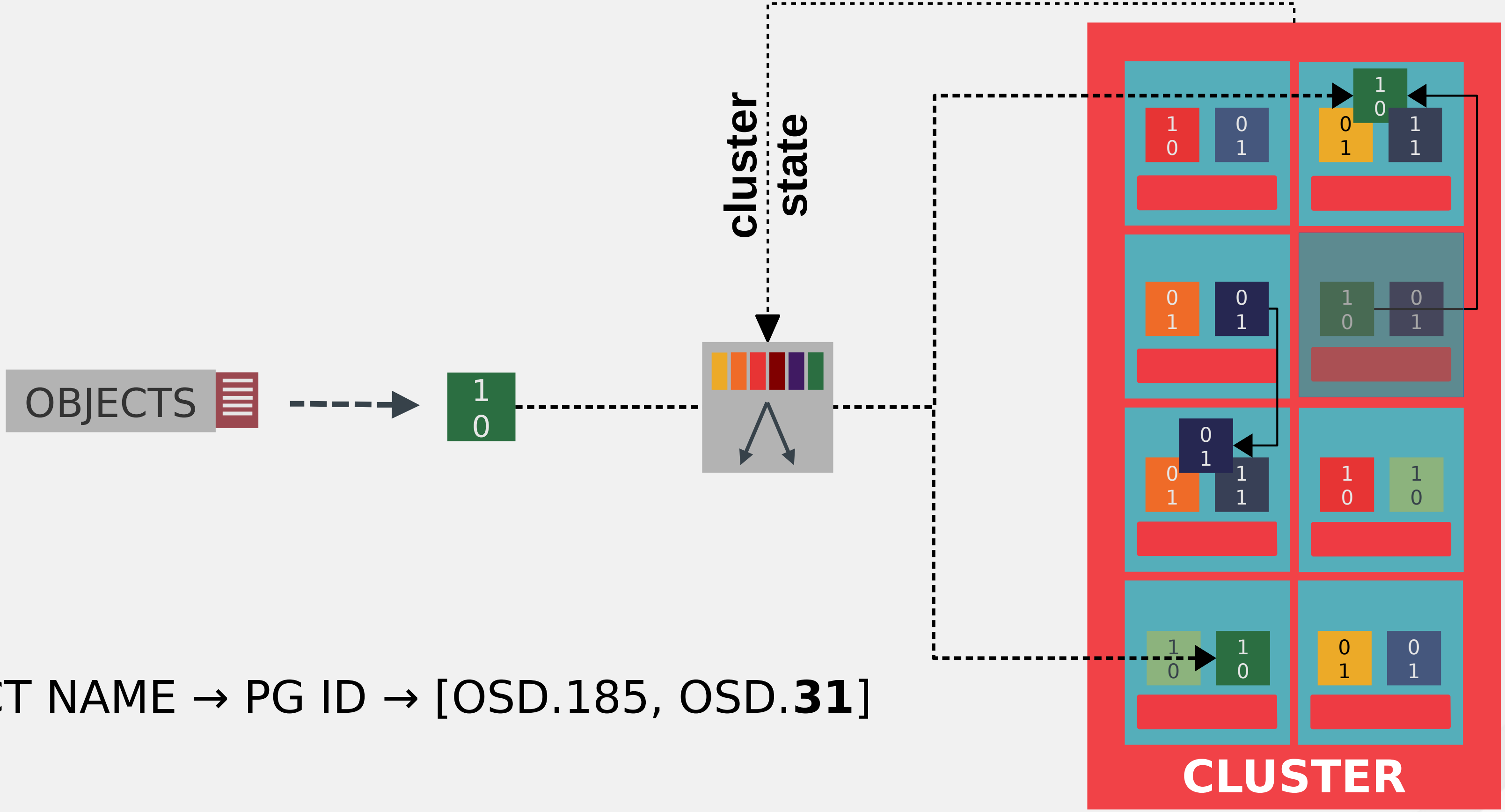
CRUSH

Controlled
Replication
Under
Scalable
Hashing



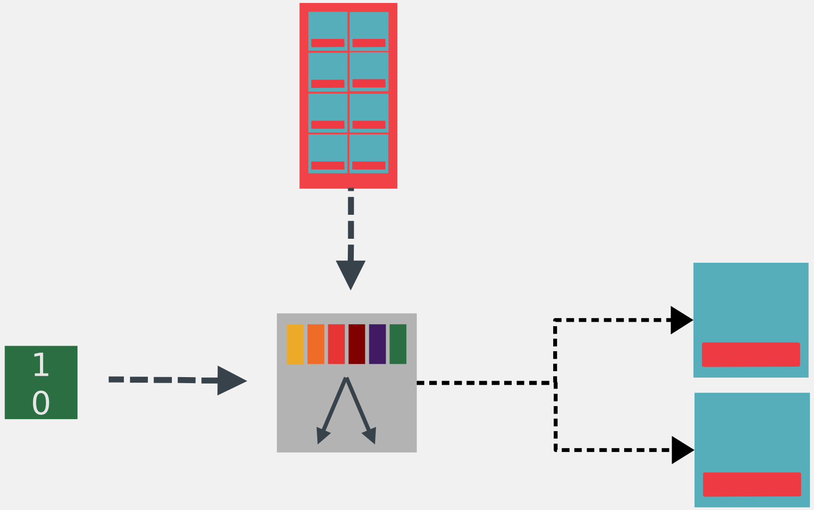
OBJECT NAME → PG ID → [OSD.185, OSD.67]

CRUSH AVOIDS FAILED DEVICES



OBJECT NAME → PG ID → [OSD.185, OSD.**31**]

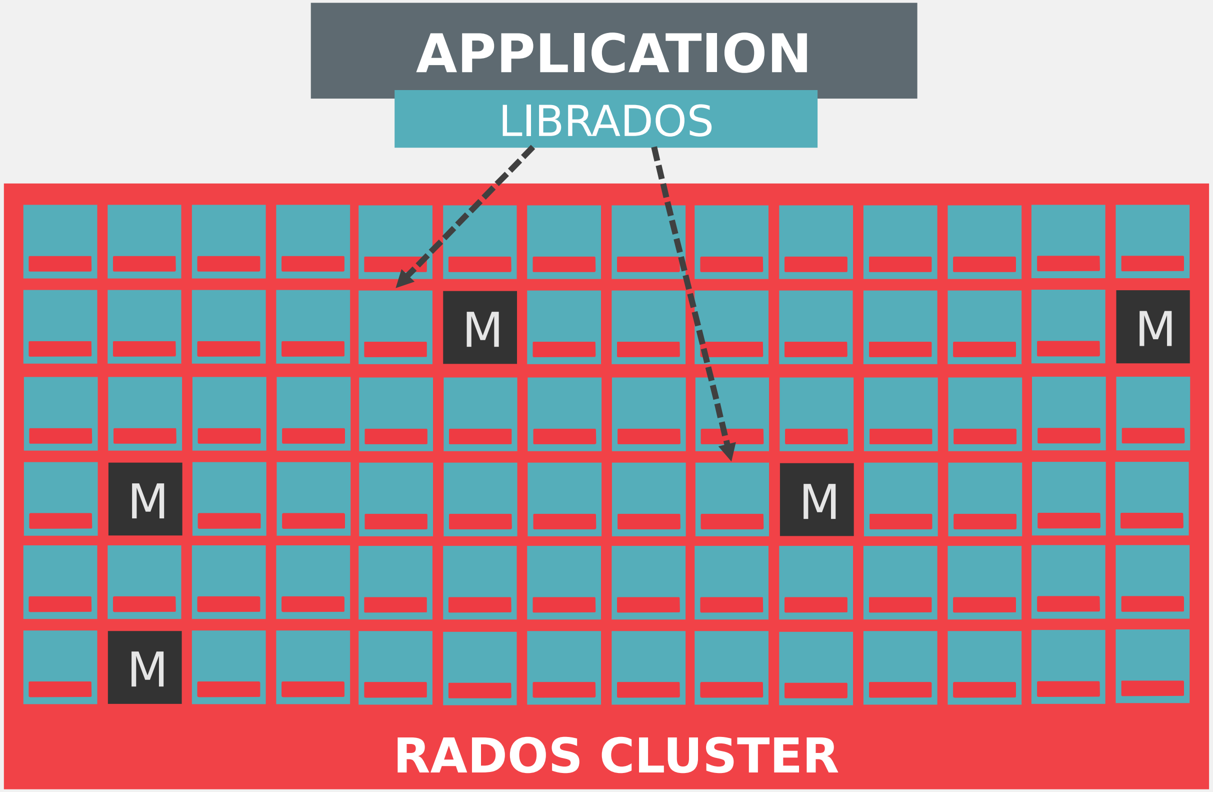
CRUSH PLACEMENT IS A FUNCTION



HASH(**OBJECT NAME**) → **PG ID**

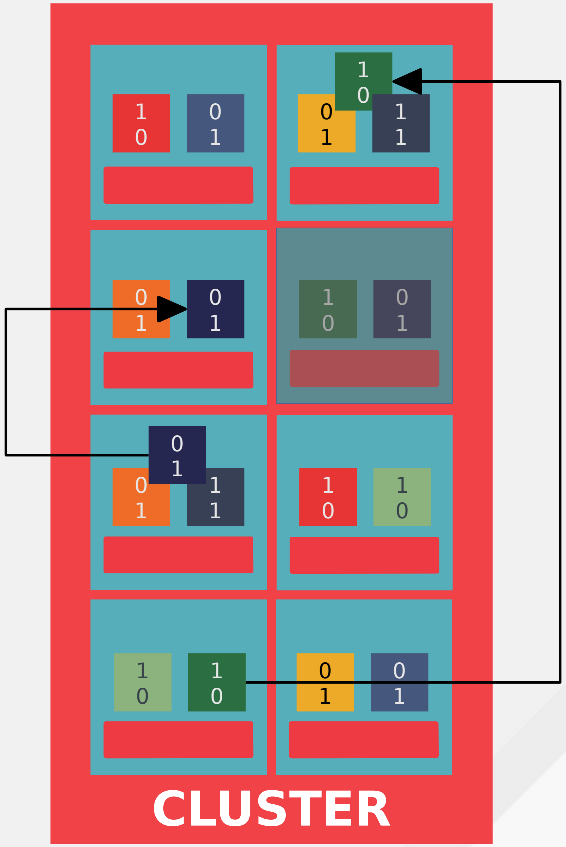
CRUSH(**PG ID**, **CLUSTER TOPOLOGY**) → [OSD.**185**, OSD.**67**]

UNIVERSALLY KNOWN FUNCTION

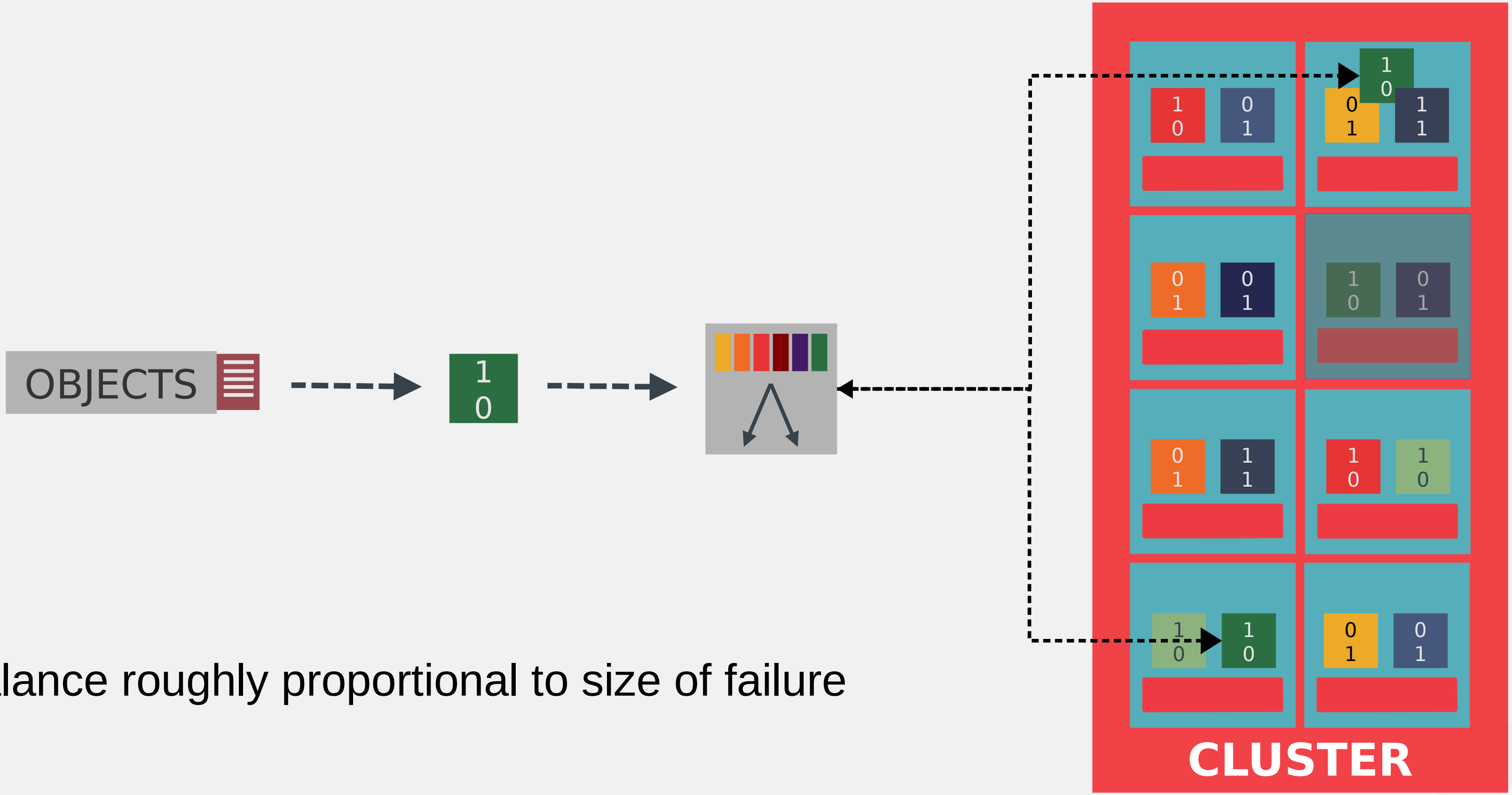


DECLUSTERED PLACEMENT

- Replicas for each device are spread around
- Failure repair is **distributed** and **parallelized**
  - recovery does not bottleneck on a single device



DEVICE FAILURES



Rebalance roughly proportional to size of failure

KEY CRUSH PROPERTIES

- **No storage** – only needs to know the cluster topology
- **Fast** – microseconds, even for very large clusters
- **Stable** – very little data movement when topology changes
- **Reliable** – placement is constrained by failure domains
- **Flexible** – replication, erasure codes, complex placement schemes

CRUSH HIERARCHY

CRUSH MAP

- Hierarchy
  - where storage devices live
  - align with physical infrastructure and other sources of failure
  - device weights
- Rules
  - policy: how to place PGs/objects
    - e.g., how many replicas
- State
  - up/down
  - current network address (IP:port)

- dc-east
  - room-1
  - room-2
    - row-2-a
    - row-2-b
      - rack-2-b-1
        - host-14
          - osd.436
          - osd.437
          - osd.438
        - host-15
          - osd.439
      - rack-2-b-2

FAILURE DOMAINS

- CRUSH generates **n** distinct target devices (OSDs)
  - may be replicas or erasure coding shards
- Separate replicas across failure domains
  - single failure should only compromise one replica
  - size of failure domain depends on cluster size
    - disk
    - host (NIC, RAM, PS)
    - rack (ToR switch, PDU)
    - row (distribution switch, …)
  - based on types in CRUSH hierarchy
- Sources of failure should be aligned
  - per-rack switch *and* PDU *and* physical location

CRUSH RULES

CRUSH RULES

- Policy
  - where to place replicas
  - the failure domain
- Trivial program
  - short sequence of imperative commands
  - flexible, extensible
  - not particularly nice for humans

```
rule flat {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type osd
        step emit
}
```

CRUSH RULES

- firstn = how to do replacement
  - **firstn** for replication
    - [8, 2, 6]
    - [8, 6, 4]
  - **indep** for erasure codes, RAID – when devices store different data
    - [8, 2, 6]
    - [8, 4, 6]
- 0 = how many to choose
  - as many as the caller needs
- type osd
  - what to choose

```
rule flat {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type osd
        step emit
}
```

CRUSH RULES

- first choose **n** hosts
  - [foo, bar, baz]
- then choose 1 osd for each host
  - [433, 877, 160]

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

CRUSH RULES

- first choose **n** hosts
  - [foo, bar, baz]
- then choose 1 osd for each host
  - [433, 877, 160]
- chooseleaf
  - quick method for the common scenario

```
rule better-by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step chooseleaf firstn 0 type host
        step emit
}
```

CRUSH RULES

- Common two stage rule
  - constrain all replicas to a row
  - separate replicas across racks

```
rule by-host-one-rack {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 1 type row
        step chooseleaf firstn 0 type rack
        step emit
}
```

ERASURE CODES

- More results
  - 8 + 4 reed-solomon → 12 devices
- Each object shard is different
  - indep instead of firstn

- Example: grouped placement
  - 4 racks
  - no more than 3 shards per rack

```
rule ec-rack-by-3 {
        ruleset 0
        type replicated
        min_size 1
        max_size 20
        step take root
        step choose indep 4 type rack
        step chooseleaf indep 3 type host
        step emit
}
```

ERASURE CODES - LRC

- Local Reconstruction Code
  - erasure code failure recovery requires more IO than replication
  - single device failures most common
  - we might go from 1.2x → 1.5x storage overhead *if* recovery were faster...
- Example:
  - 10+2+3 LRC code
  - 3 groups of 5 shards
  - single failures recover from 4 nearby shards

```
rule lrc-rack-by-5 {
        ruleset 0
        type replicated
        min_size 1
        max_size 20
        step take root
        step choose indep 3 type rack
        step chooseleaf indep 5 type host
        step emit
}
```

ODD NUMBERS

- Desired **n** is not always a nice multiple
- Example
  - three replicas
  - first two in rack A
  - third in rack
- CRUSH stops when it gets enough results

```
rule two-of-three {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 2 type rack
        step chooseleaf firstn 2 type host
        step emit
}
```

CRUSH IN PRACTICE

CRUSH HIERARCHY

- New OSDs add themselves
  - they know their host
  - ceph config may specify more

```
crush location = rack=a row=b
```

- View tree

```
ceph osd tree
```

- Adjust weights

```
ceph osd crush reweight osd.7 4.0
ceph osd crush add-bucket b rack
ceph osd crush move b root=default
ceph osd crush move mira021 rack=b
```

- Create basic rules

```
ceph osd crush rule create-simple \
    by-rack default rack
```

```
# ceph osd tree
ID  WEIGHT    TYPE NAME          UP/DOWN REWEIGHT
 -1 159.14104 root default
 -3  15.45366     host mira049
  8   0.90999         osd.8           up  1.00000
 12   3.63599         osd.12          up  1.00000
 18   3.63589         osd.18          up  1.00000
 49   3.63589         osd.49          up  1.00000
  7   3.63589         osd.7           up  1.00000
 -4  14.54782     host mira021
 20   0.90999         osd.20          up  1.00000
  5   0.90999         osd.5           up  0.82115
  6   0.90999         osd.6           up  0.66917
 11   0.90999         osd.11          up  1.00000
 17   3.63599         osd.17        down  0.90643
 19   3.63599         osd.19          up  0.98454
 15   3.63589         osd.15        down  1.00000
 -5  10.91183     host mira060
 22   0.90999         osd.22          up  1.00000
 25   0.90999         osd.25          up  0.66556
 26   0.90999         osd.26          up  1.00000
```

CLUSTER EXPANSION

- **Stable mapping**
  - Expansion by 2x: half of all objects will move
  - Expansion by 5%: ~5% of all objects will move
- **Elastic placement**
  - Expansion, failure, contraction – it's all the same
- CRUSH **always** rebalances on cluster expansion or contraction
  - balanced placement → balance load → best performance
  - rebalancing at scale is cheap

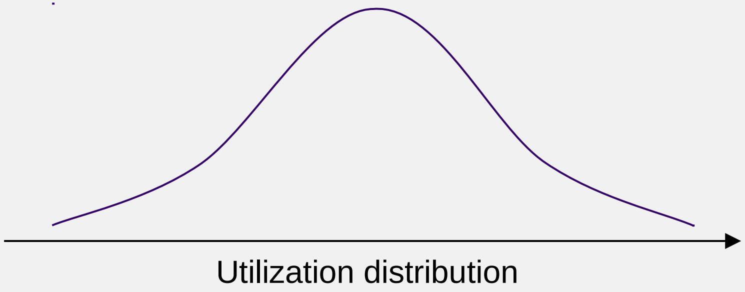
WEIGHTED DEVICES

- OSDs may be different sizes
  - different capacities
  - HDD or SSD
  - clusters expand over time
  - available devices changing constantly
- OSDs get PGs (and thus objects) proportional to their weight
- Standard practice
  - weight = size in TB

```
...
-15   6.37000     host mira019
 97   0.90999         osd.97          up  1.00000          1.00000
 98   0.90999         osd.98          up  0.99860          1.00000
 99   0.90999         osd.99          up  0.94763          1.00000
100   0.90999         osd.100         up  1.00000          1.00000
101   0.90999         osd.101         up  1.00000          1.00000
102   0.90999         osd.102         up  1.00000          1.00000
103   0.90999         osd.103         up  0.92624          1.00000
-17  17.27364     host mira031
111   0.90999         osd.111         up  1.00000          1.00000
112   0.90999         osd.112         up  0.95805          1.00000
 21   3.63599         osd.21          up  0.95280          1.00000
 16   3.63589         osd.16          up  0.92506          1.00000
114   0.90999         osd.114         up  0.83000          1.00000
 58   3.63589         osd.58          up  1.00000          1.00000
 61   3.63589         osd.61          up  1.00000          1.00000
...
```

DATA IMBALANCE

- CRUSH placement is pseudo-random
  - behaves like a random process
  - "**uniform** distribution" in the statistical sense of the word
- Utilizations follow a **normal** distribution
  - more PGs → tighter distribution
  - bigger cluster → more outliers
  - high outlier → overfull OSD



REWEIGHTING

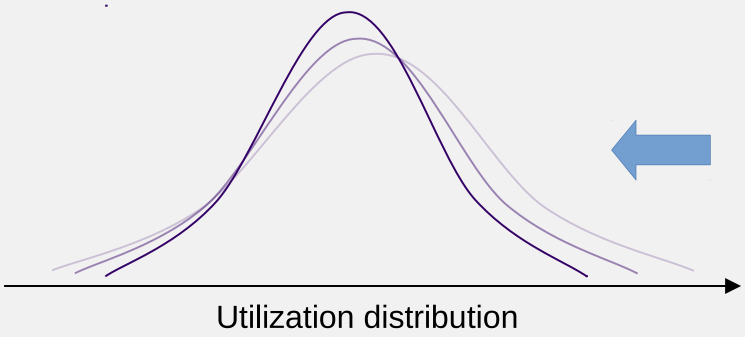
- OSDs get data proportional to their weight
- Unless they have failed...
  - ...they get no data
  - CRUSH does internal “retry” if it encounters a failed device

- Reweight treats failure as non-binary
  - 1 = this device is fine
  - 0 = always reject it
  - .9 = reject it 10% of the time

REWEIGHT-BY-UTILIZATION

- Find OSDs with highest utilization
  - reweight proportional to their distance from average
- Find OSDs with lowest utilization
  - if they were previously reweighted down, reweight back up

- Run periodically, automatically
- Make small, regular adjustments to data balance

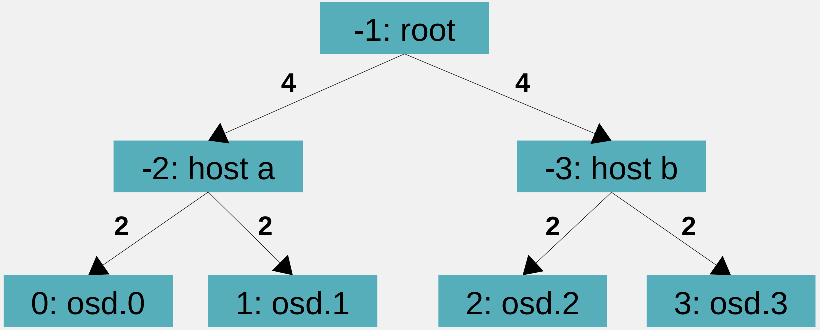


INTERNALS

HOW DOES IT WORK?

- follow rule steps
- pseudo-random weighted descent of the hierarchy
- retry if we have to reject a choice
  - device is failed
  - device is already part of the result set

HOW DOES IT WORK?



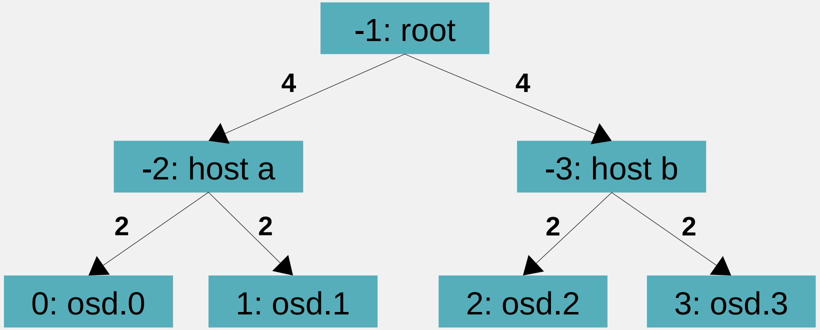
- Weighted tree
- Each node has a unique id

- While CRUSH is executing, it has a “working value” vector → → →

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[ ]
```

HOW DOES IT WORK?

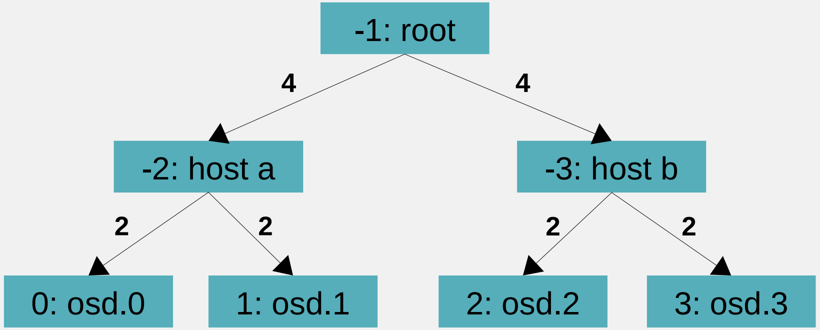


- take root

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
```

HOW DOES IT WORK?

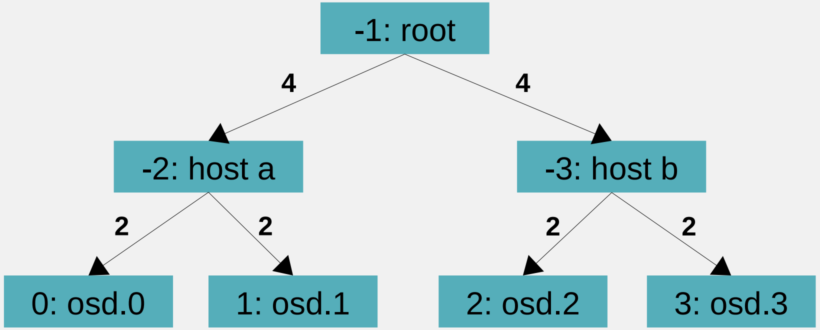


- choose firstn 0 type host
  - nrep=2
- descend from [-1] with
  - x=<whatever>
  - r=0
  - hash(-1, x, 0)
    → -3

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
[-3]
```

HOW DOES IT WORK?



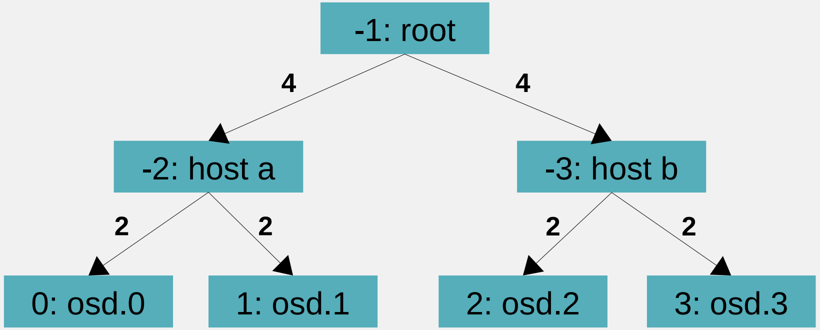
- choose firstn 0 type host
  - nrep=2
- descend from [-1] with
  - x=<whatever>
  - r=**1**
  - hash(-1, x, 1)

    → -3 → dup, reject

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
[-3]
```

HOW DOES IT WORK?



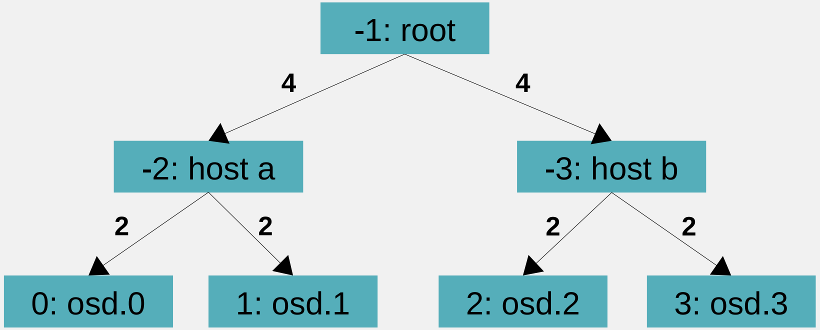
- choose firstn 0 type host
  - nrep=2
- descend from [-1] with
  - x=<whatever>
  - r=2
  - hash(-1, x, 2)

    → -2

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
[-3,-2]
```

HOW DOES IT WORK?



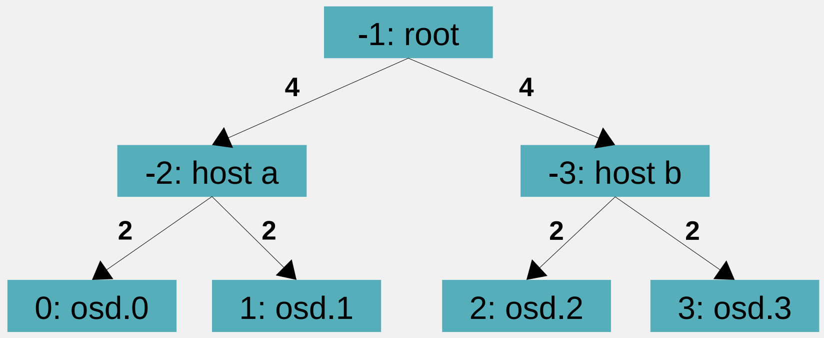
- choose firstn 1 type osd
  - nrep=1
- descend from [**-3**,-2] with
  - x=<whatever>
  - r=0
  - hash(-3, x, 0)

    → 2

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
[-3,-2]
[2]
```

HOW DOES IT WORK?



- choose firstn 1 type osd
  - nrep=1
- descend from [-3,**-2**] with
  - x=<whatever>
  - r=1
  - hash(-2, x, 1)

    → 1

```
rule by-host {
        ruleset 0
        type replicated
        min_size 1
        max_size 10
        step take root
        step choose firstn 0 type host
        step choose firstn 1 type osd
        step emit
}
```

```
[-1]
[-3,-2]
[2,1]
```

BUCKET/NODE TYPES

- Many algorithms for selecting a child
  - every internal tree node has a type
  - tradeoff between time/computation and rebalancing behavior
  - can mix types within a tree
- uniform
  - hash(nodeid, x, r) % num_children
  - fixed O(1) time
  - adding child shuffles everything
- straw2
  - hash(nodeid, x, r, child) for every child
  - scale based on child weight
  - pick the biggest value
  - O(n) time
- adding or removing child
  - only moves values to or from that child
  - still fast enough for small n

TUNABLES

CHANGING CRUSH BEHAVIOR

- We discover improvements to the algorithm all the time
  - straw → straw2
  - better behavior with retries
  - …
- Clients and servers must run identical versions
  - everyone has to agree on the results

- All behavior changes are conditional
  - tunables control which variation of algorithm to use
  - deploy new code across whole cluster
  - only enable new behavior when all clients and servers have upgraded
  - once enabled, prevent older clients/servers from joining the cluster

TUNABLE PROFILES

- Ceph sets tunable “profiles” named by the release that first supports them
  - argonaut – original legacy behavior
  - bobtail
    - choose_local_tries = 0, choose_local_fallback_tries = 0
    - choose_total_tries = 50
    - chooseleaf_descend_once = 1
  - firefly
    - chooseleaf_vary_r = 1
  - hammer
    - straw2
  - jewel
    - chooseleaf_stable = 1

SUMMARY

CRUSH TAKEAWAYS

- CRUSH placement is functional
  - we **calculate** where to find data—no need to store a big index
  - those calculations are **fast**
- Data distribution is **stable**
  - just enough data data is migrated to restore balance
- Placement is **reliable**
  - CRUSH separated replicas across failure domains
- Placement is **flexible**
  - CRUSH rules control how replicas or erasure code shards are separated
- Placement is **elastic**
  - we can add or remove storage and placement rules are respected

THANK YOU

Sage Weil
sage@redhat.com
@liewegas

http://ceph.com/
http://redhat.com/storage

RED HAT
SUMMIT

LEARN. NETWORK.
EXPERIENCE OPEN SOURCE.