Detect, Decide, Fence: Resolving Netsplits in 3-AZ Stretch Cluster

Kamoltat (Junior) Sirivadhna, IBM

Agenda:

- 2AZ (stretch mode) → 3AZ
- Problem with split brain in 3-AZ Stretch Cluster
- Detecting Netsplit
- Resolving Netsplit
  - Clique Detection (Bron Kerbosch)
  - Heuristics in deciding which bucket to fence.
  - Fencing of OSDs
  - Lifting the Fence
  - Different Scenarios and Fencing Decisions.
- Q&A

2AZ (stretch mode) → 3AZ

- 2 Availability Zones (2AZ) stretch mode ✅ (Since v16.1.0 Pacific)
  - Goal:
    - Expand failure domain from host level to zone level.
    - Survive 1 total zone failure
    - Automatically resolve and survive network partition.
- 3 Availability Zones (3AZ) ✅ (Since v19.2.1 squid)
  - Goal:
    - Support more zones = higher availability from 2 -> 3 zones
    - Survive 1 zone failure + 1 host failure
    - Automatically resolve and survive network partition. ⚠️
      - Limitation
        - For 2AZ the tie-breaker monitor takes care of this
        - We cannot have tie-breaker monitor in 3AZ!

Problem: Split Brain during Netsplit in a 3-AZ

1. Network failure between DC1 and DC2

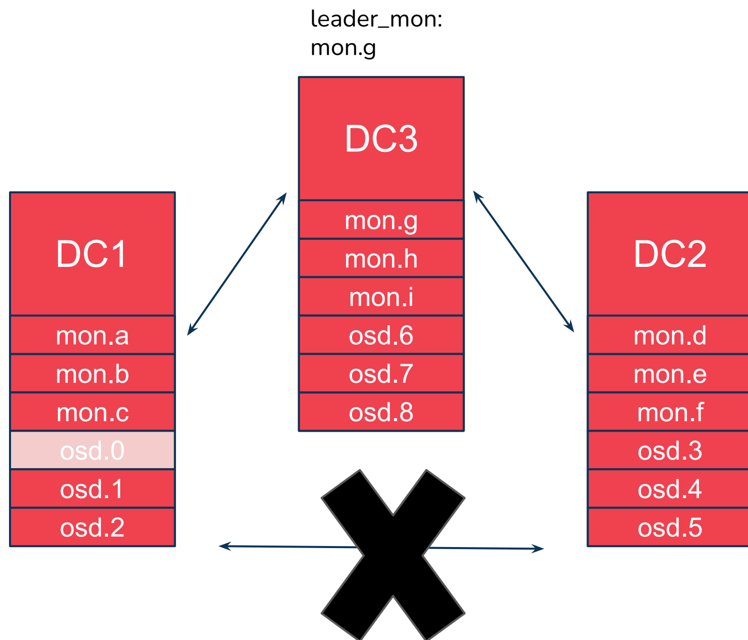
2. osd.3 and osd.4 reports heartbeat failure of osd.0 to the leader_mon (through forward op)

3. Leader monitor marks osd.0 as down in the OSDMap since there are enough peers that reported osd.0 down.

4. However, osd.0 is able to reach the leader monitor and osd.0 is marked back up.

PG keep re-peering and becomes inactive, I/O is stopped on the affected PGs.

✗ I/O



size = 6
min_size = 3
quorum {a,b,c,d,e,f,g,i}
pg 1.1 = {0,1,3,4,6,7}
leader_mon = mon.g

We need to pick a winning side to continue serving I/O.

Solution:

- Detecting Netsplit ✅ (Since v19.2.4)
- Auto-Resolving Netsplit (Targeting next release)
  - Maximal Clique Detection (Bron Kerbosch)
  - Use user-preference or heuristics in deciding which zone to fence.
  - Fencing of OSDs
  - Lifting the Fence

Detecting Netsplit

Ceph is aware of network partitions in both monitor level and zone level.

dc1 : {a, b, c}   dc2: {d, e, f}   dc3: {h, i, j}

Scenario 1:

mon.a ——x—— mon.d   **MON_NETSPLIT:"Netsplit detected between mon.a and mon.d"**

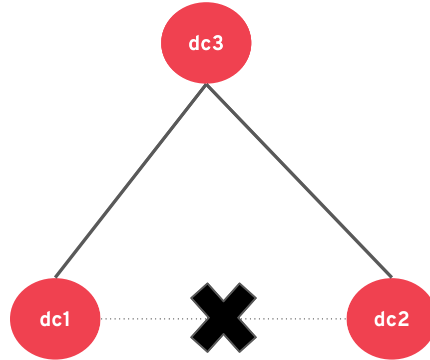
Scenario 2:

mon.a ——x—— mon.d

mon.b ——x—— mon.e   **MON_NETSPLIT: "Netsplit detected between dc1 and dc2"**

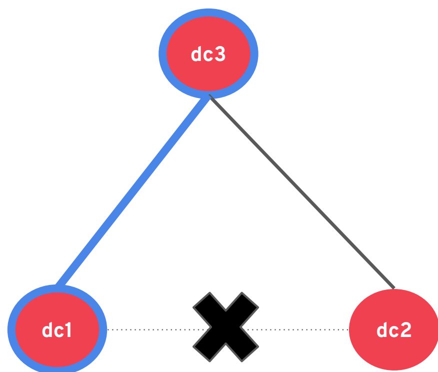
mon.c ——x—— mon.f

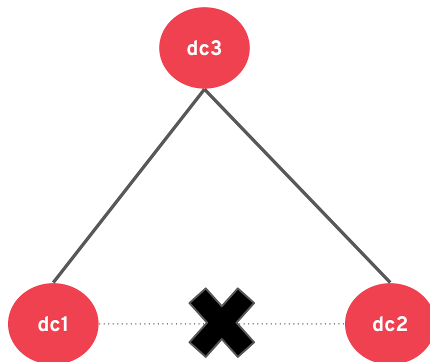
How do we identify connected DCs after a netsplit?



How do we identify connected DCs after a netsplit?

{dc1, dc3}





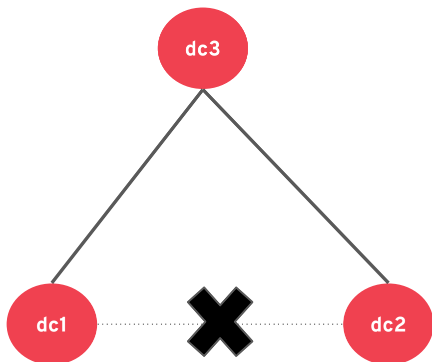
How do we identify connected DCs after a netsplit?

We need a search algorithm to find the largest connected sets of DCs!

Bron Kerbosch (BK algorithm)

BK will find all maximal cliques by:

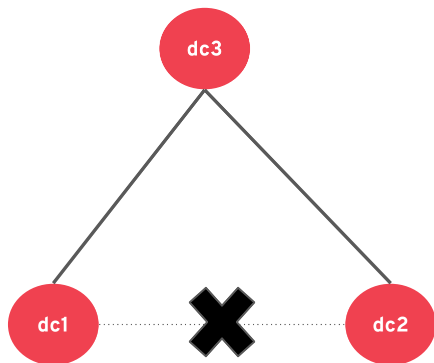
Clique: a group of vertices in which **every vertex is directly connected to every other vertex.**



1. Iterating through each candidate set. e.g., {dc1, dc2, dc3}
2. Adding each candidate's neighbor to form a clique.
3. Memorizing which candidate we have visited to avoid duplication
4. Reporting the clique is a maximal clique when there are no candidates left and that we are not a subset of a larger clique.

Runtime: *O(3^(n/3))*. Since n = zones, the search space is small in practice.

Bron Kerbosch (BK algorithm)




```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0      if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0         for each v in candidate_set:
2.1             clique_copy = clique
2.2             clique_copy.push_back(v)
2.3             neighbors = adj_list.at(v)
2.4             candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5             exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6             bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7             candidate_set.erase(v)
2.8             exclude_set.insert(v)
```

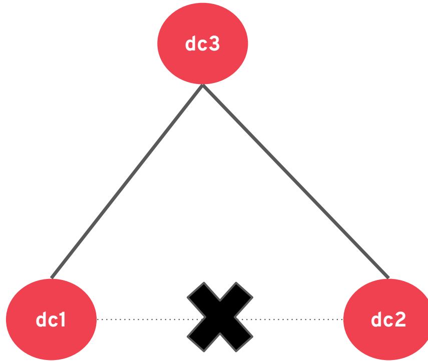
clique - set of cliques we are building

candidate_set - set of potential vertices we can add to the clique

exclude_set - set of vertices we have explored.

clique = {}
candidate_set = {dc1, dc2, dc3}
exclude_set = {}

Bron Kerbosch (BK algorithm)

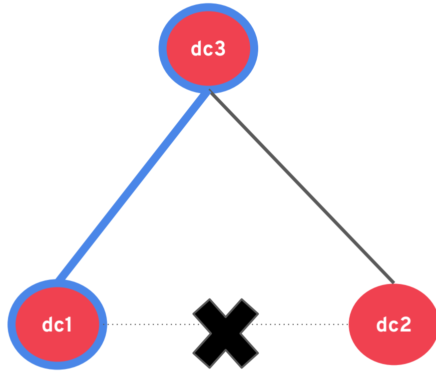



```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0      if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0         for each v in candidate_set:
2.1            clique_copy = clique
2.2            clique_copy.push_back(v)
2.3            neighbors = adj_list.at(v)
2.4            candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5            exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6            bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7            candidate_set.erase(v)
2.8            exclude_set.insert(v)
```

clique = {dc1}
candidate_set = {dc1, dc2, dc3}
exclude_set = {}

dc1

Bron Kerbosch (BK algorithm)




```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0     if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0        for each v in candidate_set:
2.1           clique_copy = clique
2.2           clique_copy.push_back(v)
2.3           neighbors = adj_list.at(v)
2.4           candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5           exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6           bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7           candidate_set.erase(v)
2.8           exclude_set.insert(v)
```

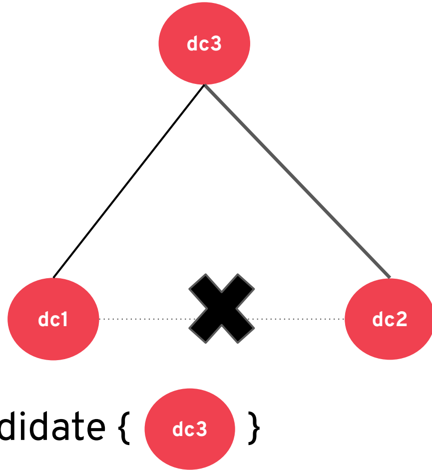
dc1's only neighbor is dc3!

clique = {dc1}
candidate_set= {dc1, dc2, dc3} ∩ {dc3} = **dc3**
exclude_set = {}

dc1

Intersect of candidate_set and dc1's neighbor = dc3!

Bron Kerbosch (BK algorithm)

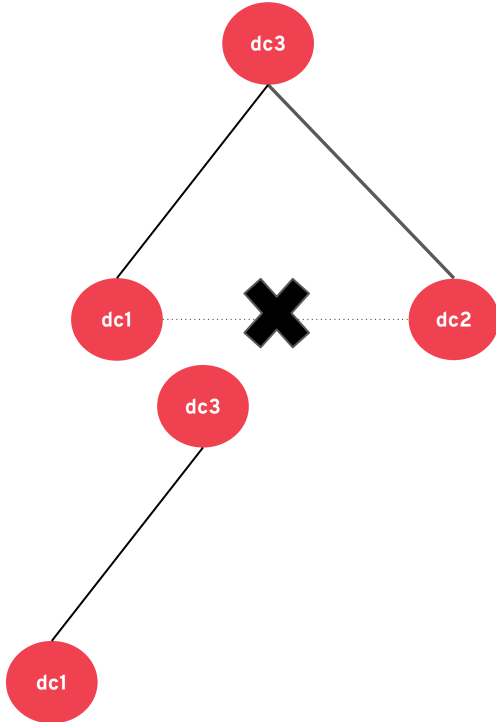



```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0      if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0         for each v in candidate_set:
2.1            clique_copy = clique
2.2            clique_copy.push_back(v)
2.3            neighbors = adj_list.at(v)
2.4            candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5            exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6            bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7            candidate_set.erase(v)
2.8            exclude_set.insert(v)
```

clique = {dc1}
candidate_set = {dc3}
exclude_set = {}

1. Added dc3 into candidate set
2. Recursively call BK algorithm!

Bron Kerbosch (BK algorithm)




```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0    if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0       for each v in candidate_set:
2.1          clique_copy = clique
2.2          clique_copy.push_back(v)
2.3          neighbors = adj_list.at(v)
2.4          candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5          exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6          bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7          candidate_set.erase(v)
2.8          exclude_set.insert(v)
```

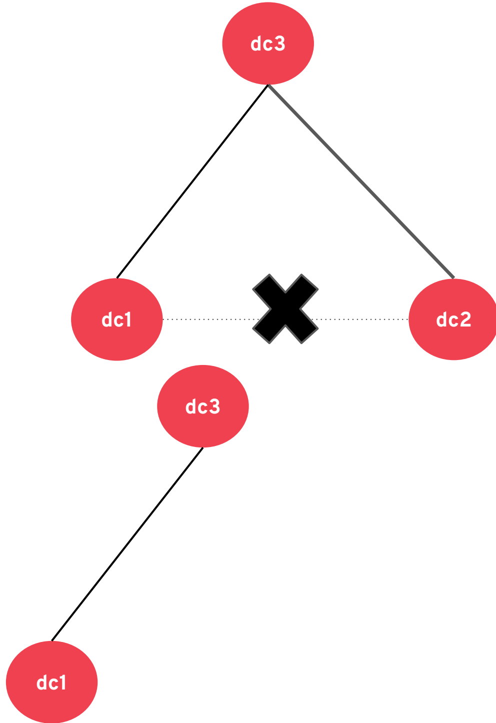
clique = {dc1, dc3}

candidate_set ={dc3} ⋂ {dc1, dc2} = {}

exclude_set = {}

1. Added dc3 into clique set = {dc1, dc3}
2. Find the intersection between dc3 (the candidate) and neighbors of dc3 = {}
3. Recursively call BK algorithm with clique = {dc1, dc3}, candidate_set = {}, exclude_set = {}

Bron Kerbosch (BK algorithm)




```
bronKerbosch(clique, candidate_set, exclude_set):
1. 0      if (candidate_set and exclude_set both empty) return // We found maximal clique
2.0        for each v in candidate_set:
2.1           clique_copy = clique
2.2           clique_copy.push_back(v)
2.3           neighbors = adj_list.at(v)
2.4           candidate_set_copy = intersect_sets(candidates_set, neighbors)
2.5           exclude_set_copy = intersect_sets(excluded_set,   neighbors)

2.6           bronKerbosch(clique_copy, candidate_set_copy, exlcude_set_copy)

2.7           candidate_set.erase(v)
2.8           exclude_set.insert(v)
```

clique = {dc1, dc3}
candidate_set = {}
exclude_set = {}

Hit base case: candidate_set = {} and exlcude_set = {}

Report {dc1, dc3} is a maximal clique!

Bron Kerbosch (BK algorithm)

clique = {}
candidate_set = {dc1, dc2, dc3}
exclude_set = {}

clique = {dc1}
candidate_set = {dc3}
exclude_set = {}

clique = {d2}
candidate_set = {dc3}
exclude_set = {}

clique = {dc3}
candidate_set = {dc3}
exclude_set = {dc1, dc2}

clique = {dc1, dc3}
candidate_set = {}
exclude_set = {}

clique = {dc2, dc3}
candidate_set = {}
exclude_set = {}

clique = {dc1, dc3}
candidate_set = {}
exclude_set = {dc1, dc2}

dc3
dc1
dc2
dc1
dc3
dc2

Maximal Clique = [{dc1, dc3} , {dc2, dc3}]

But there are **two** largest maximal!

Option 1: User Preference (netsplit_zone_preferences)

Set zone preferences (persisted in MonMap):

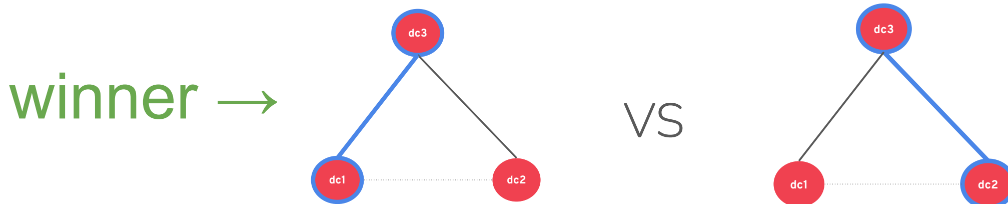
```
ceph mon set netsplit_zone_preferences dc1 dc2 dc3
```

Sum preference rankings per zone

- **dc1 = 3 points (highest)**
- **dc2 = 2 points**
- **dc3 = 1 point**

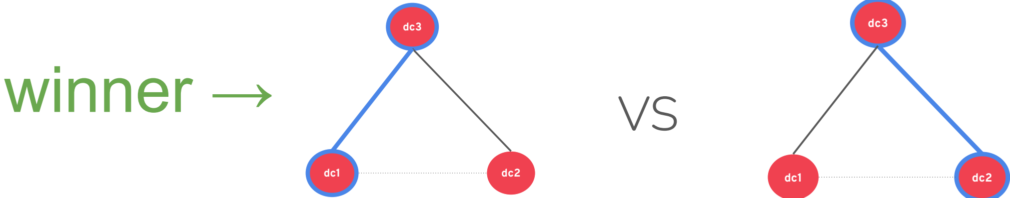
Partition with highest total score wins

Example result: Partition A (dc1+dc3) = 4 points > Partition B (dc2+dc3) = 3 points



Option 2: Automation (Heuristic)

Heuristic	{dc1, dc3}	{dc2, dc3}	Result 🏆
1. Total Effective OSD Weight (osd_crush_weight × osdmap_weight)	**120.0**	**120.0**	**Tie → next**
2. Total Number of Up Monitors	**6** (dc1: 3, dc3: 3)	**6** (dc2: 3, dc3: 3)	**Tie → next**
3. Total Monitor Connection Scores	**5.92**	**5.87**	**{dc1, dc3} wins**
Tiebreaker (if needed)	Lexicographic → `{dc1, dc3}`		Did not reached.



Pre-fencing procedures

After identifying the winning side of the partition:

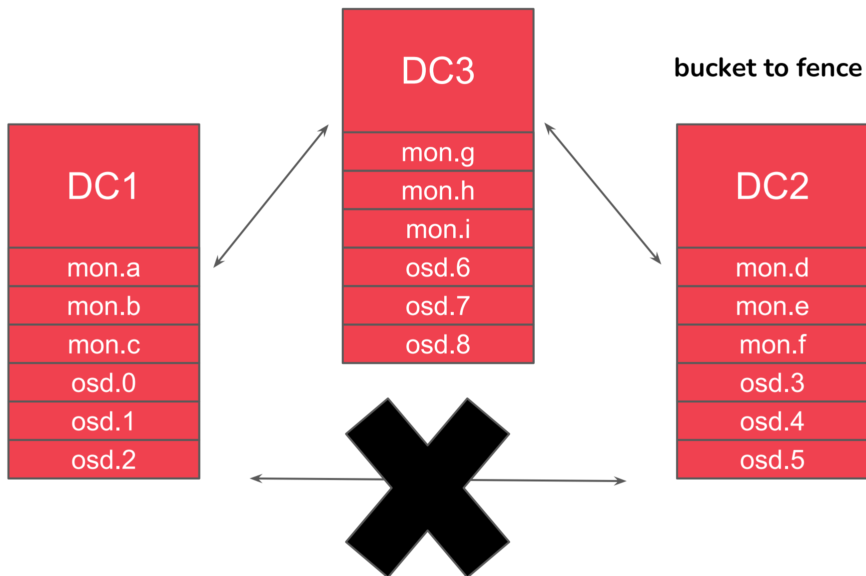
1. **Work out the side that needs fencing (fenced_buckets).**
    i. Example: {dc1, dc2, dc3} - {dc1, dc3}; we fence **all OSDs** in {dc2}.
2. **Persist this in OSDMap**

3. **Set flags: norecover & nobackfill globally (unset after fencing is applied successfully)**
    a. Remapped PGs don't immediately start recovery/backfill churn

Fencing of the OSDs

- **Monitors Ignore failure reports from fenced OSDs**
  Drop any **MOSD_FAILURE** messages from or about fenced OSDs in the monitor,
  to avoid split brain behavior.
- **Exclude fenced OSDs from PG when choosing acting set**
  Remove fenced OSDs, when the primary PG is choosing candidates for its acting_set (see choose_acting in PeeringState.cc). This will trigger remapping, and some PGs will be undersized.
- **netsplit_zone_preferences cannot be modified while fencing.**

Fencing of the OSDs



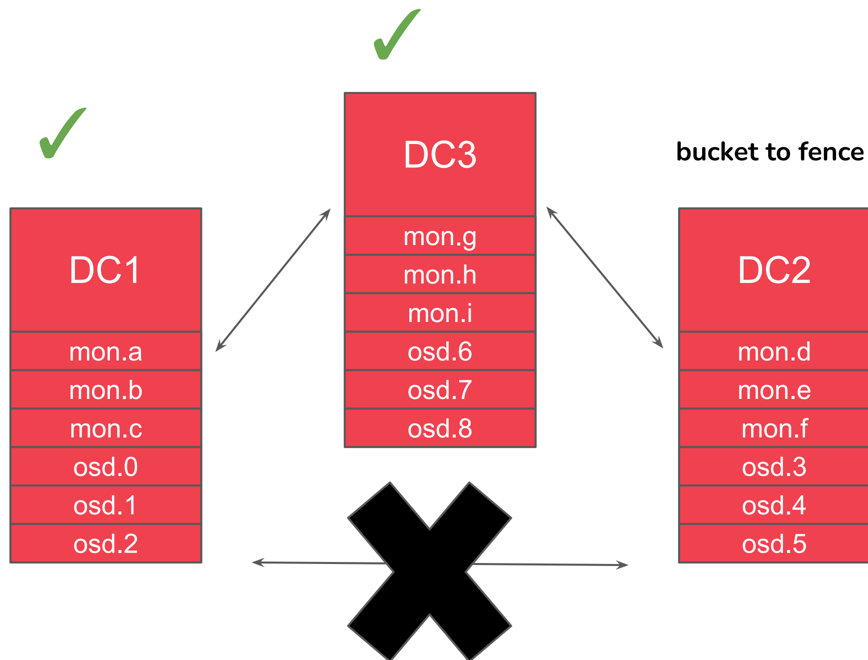
bucket to fence

1. Netsplit detected between dc1 and dc2

2. Bron Kerbosch outputs the maximum cliques: [{DC1, DC3}, {DC2, DC3}].

3. Heuristic Calculation determines that we should choose {DC1, DC3} to be the surviving site and fence DC2.

Fencing of the OSDs



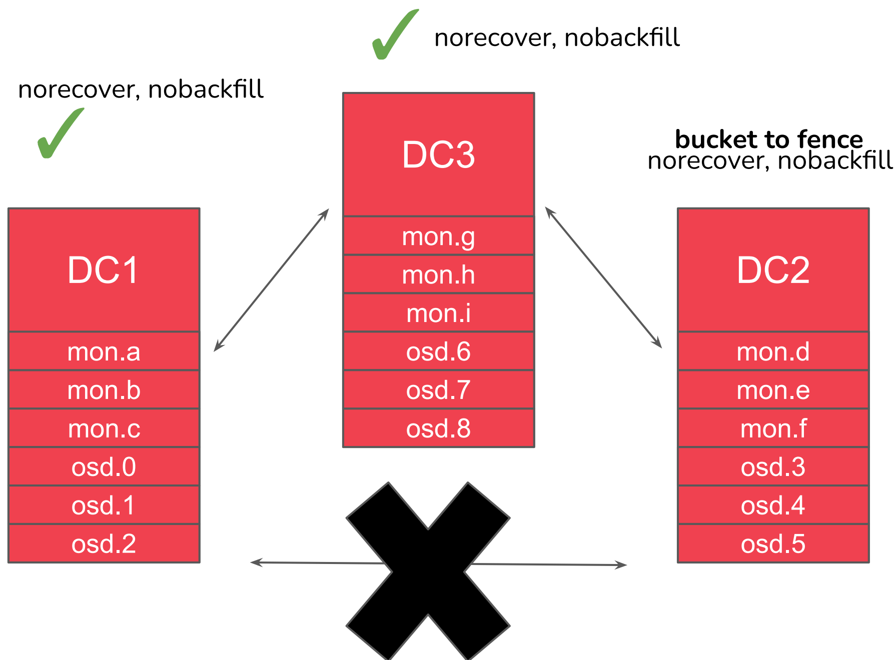
1. Netsplit detected between dc1 and dc2

2. Bron Kerbosch outputs the maximum cliques: [{DC1, DC3}, {DC2, DC3}].

3. Heuristic Calculation determines that we should choose {DC1, DC3} to be the surviving site and fence DC2.

4. Pre-safety checks passed

Fencing of the OSDs



1. Netsplit detected between dc1 and dc2

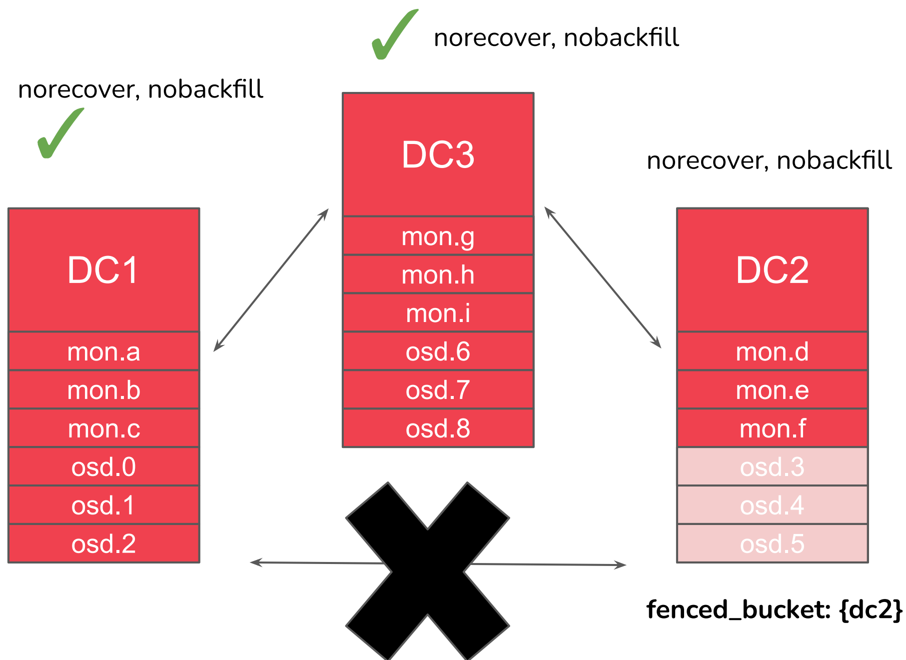
2. Bron Kerbosch outputs the maximum cliques: [{DC1, DC3}, {DC2, DC3}].

3. Heuristic Calculation determines that we should choose {DC1, DC3} to be the surviving site and fence DC2.

4. Pre-safety checks passed

5. Set norecover, nobackfill globally.

Fencing of the OSDs



1. Netsplit detected between dc1 and dc2

2. Bron Kerbosch outputs the maximum cliques: [{DC1, DC3}, {DC2, DC3}].

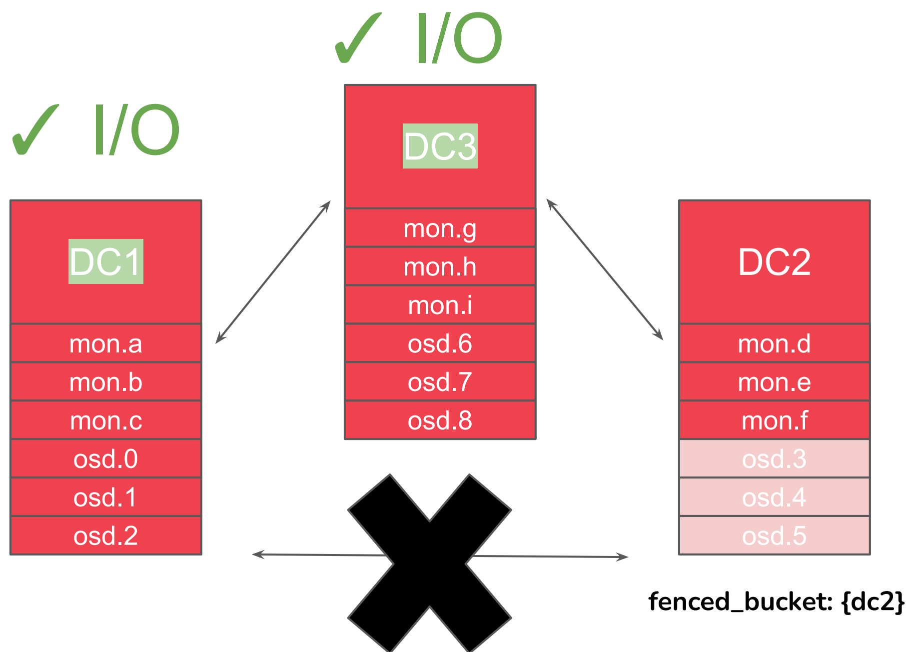
3. Heuristic Calculation determines that we should choose {DC1, DC3} to be the surviving site and fence DC2.

4. Pre-safety checks passed

5. Set norecover, nobackfill globally.

6. Apply the fence -> fenced_buckets.insert({DC2})-> commit to OSDMap.

Fencing of the OSDs



1. Netsplit detected between dc1 and dc2

2. Bron Kerbosch outputs the maximum cliques: [{DC1, DC3}, {DC2, DC3}].

3. Heuristic Calculation determines that we should choose {DC1, DC3} to be the surviving site.

4. Pre-safety checks passed

5. Set norecover, nobackfill globally.

6. Apply the fence -> fenced_buckets.insert({DC2})-> commit to OSDMap.

7. Unset norecover, nobackfill globally.

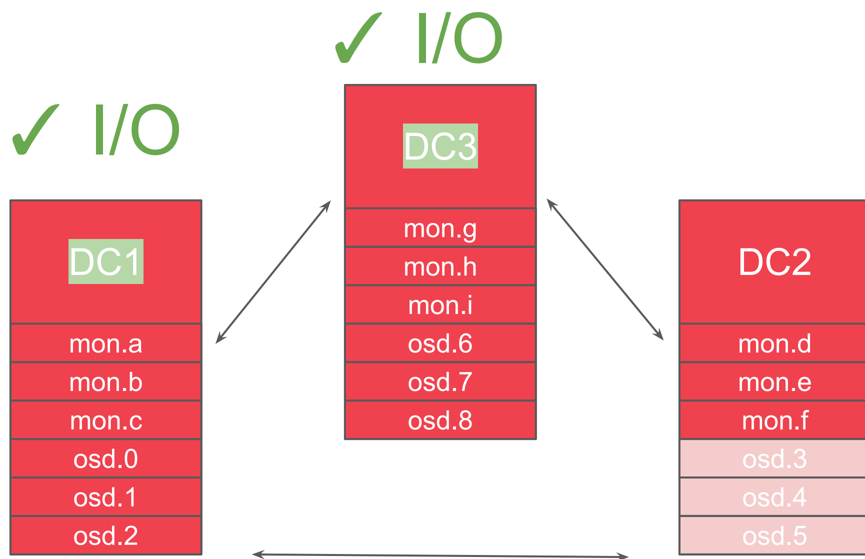
OSDs in DC2 are fenced!
All, PGs are now active and can accept IO with some PGs being in active+undersized, active+degraded.

Lifting the Fence

We have to ensure that we are not lifting the fence too quickly

- **Lift occurs only after the cluster no longer experiencing netsplit**
  - All data centers reconnect, no location-level netsplit detected.
  - Condition holds for ≥ **lift_fencing_threshold** (default: 90s).
- **Prevents "fence flapping" during transient netsplit.**
- **Once lifted, PGs can safely re-peer and rebalance.**

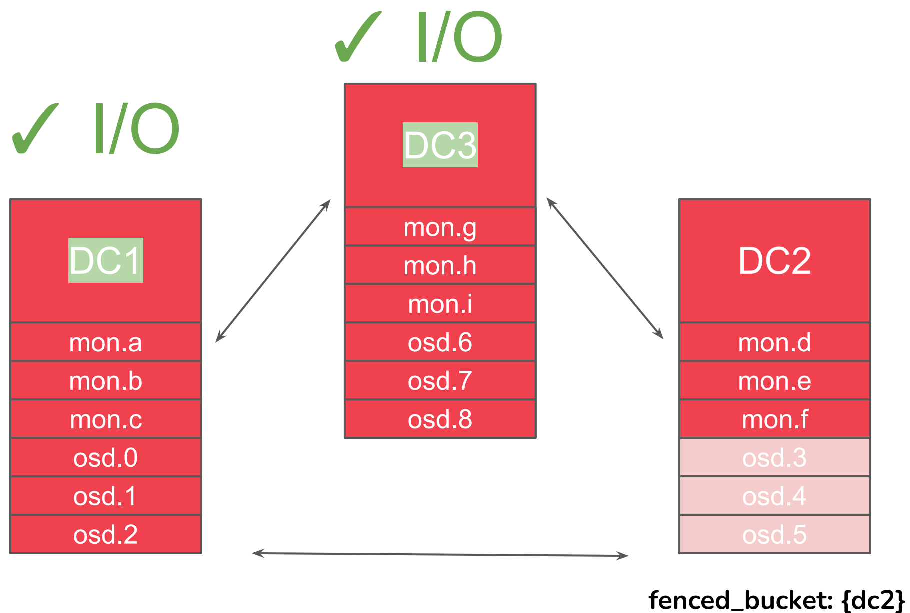
Lifting the Fence



1. Netsplit no longer detected between dc1 and dc2

fenced_bucket: {dc2}

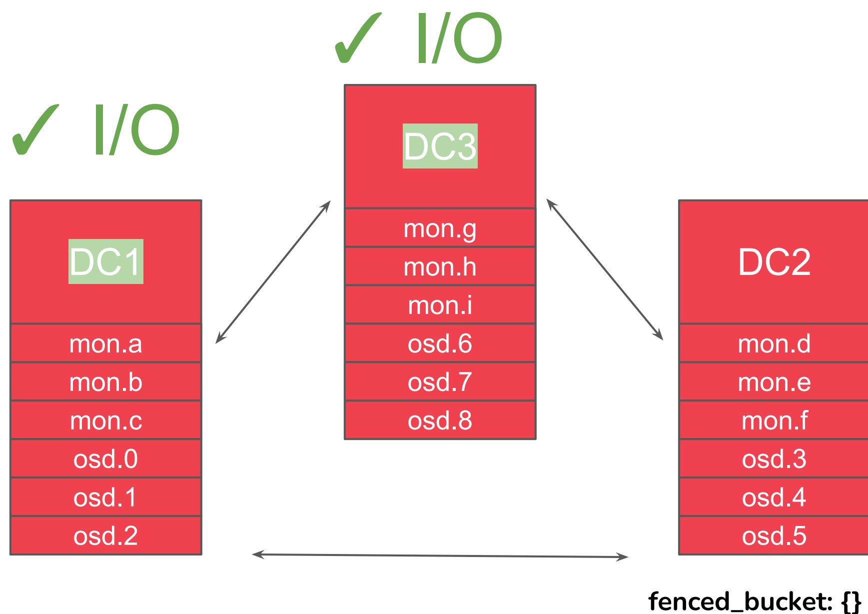
Lifting the Fence



fenced_bucket: {dc2}

1. Netsplit no longer detected between dc1 and dc2
2. Condition holds for ≥ lift_fencing_threshold (default: 90s)

Lifting the Fence



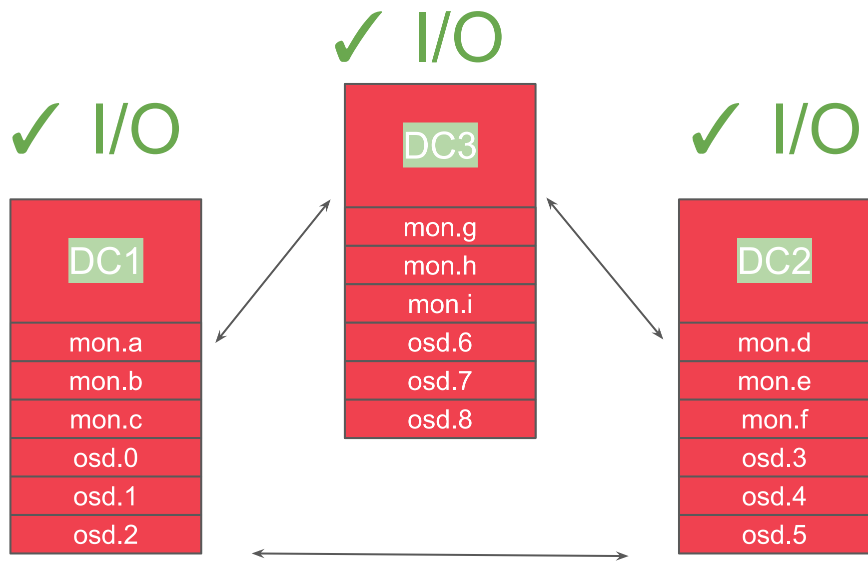
fenced_bucket: {}

1. Netsplit no longer detected between dc1 and dc2

2. Condition holds for ≥ lift_fencing_threshold (default: 90s)

3. Lift fencing -> fenced_bucket.clear() -> commit to OSDMap

Lifting the Fence



1. Netsplit no longer detected between dc1 and dc2

2. Condition holds for ≥ lift_fencing_threshold (default: 90s)

3. Lift fencing -> fenced_bucket.clear() -> commit to OSDMap

4. OSDs in DC2 rejoins the cluster and all PGs slowly become active+clean.

fenced_bucket: {}

Different Scenarios and Fencing Decisions.

Scenario 1: Netsplit detected, no fence yet

- **Apply fence** (After pre-check passed).

Scenario 2: Currently fencing, Heuristics -> same fenced bucket

- **Hold fence** (update metadata only)

Scenario 3: Currently fencing, Heuristics -> different buckets: EDGE CASE!

- **Hold fence** don’t switch since risk > reward, worst case we end up with less optimal side of the cluster but it can still serve IO.

Scenario 5: No netsplit < 90s (lift_fencing_threshold)

- **Hold fence** (don’t lift to avoid flapping)

Scenario 6: No netsplit > 90s (lift_fencing_threshold)

- **Lift fence**

Acknowledgements

- Bill
- Greg
- Neha
- Radoslaw
- Sam

Thank You!
Q&A