Orchestrator High Availability tutorial

Shlomi Noach
GitHub

PerconaLive 2018

About me

@github/database-infrastructure

Author of **orchestrator**, **gh-ost**, **freno**, **ccql** and others.

Blog at http://openark.org

@ShlomiNoach

Agenda

- Introduction to **orchestrator**
- Basic configuration
- Reliable detection considerations
- Successful failover considerations
- **orchestrator** failovers
- Failover meta
- **orchestrator/raft** HA
- Master discovery approaches

GitHub

Largest open source hosting

67M repositories, 24M users

Critical path in build flows

Best octocat T-Shirts and stickers

MySQL at GitHub

Stores all the metadata: users, repositories, commits, comments, issues, pull requests, ...

Serves web, API and auth traffic

MySQL 5.7, semi-sync replication, RBR, cross DC

~15 TB of MySQL tables

~150 production servers, ~15 clusters

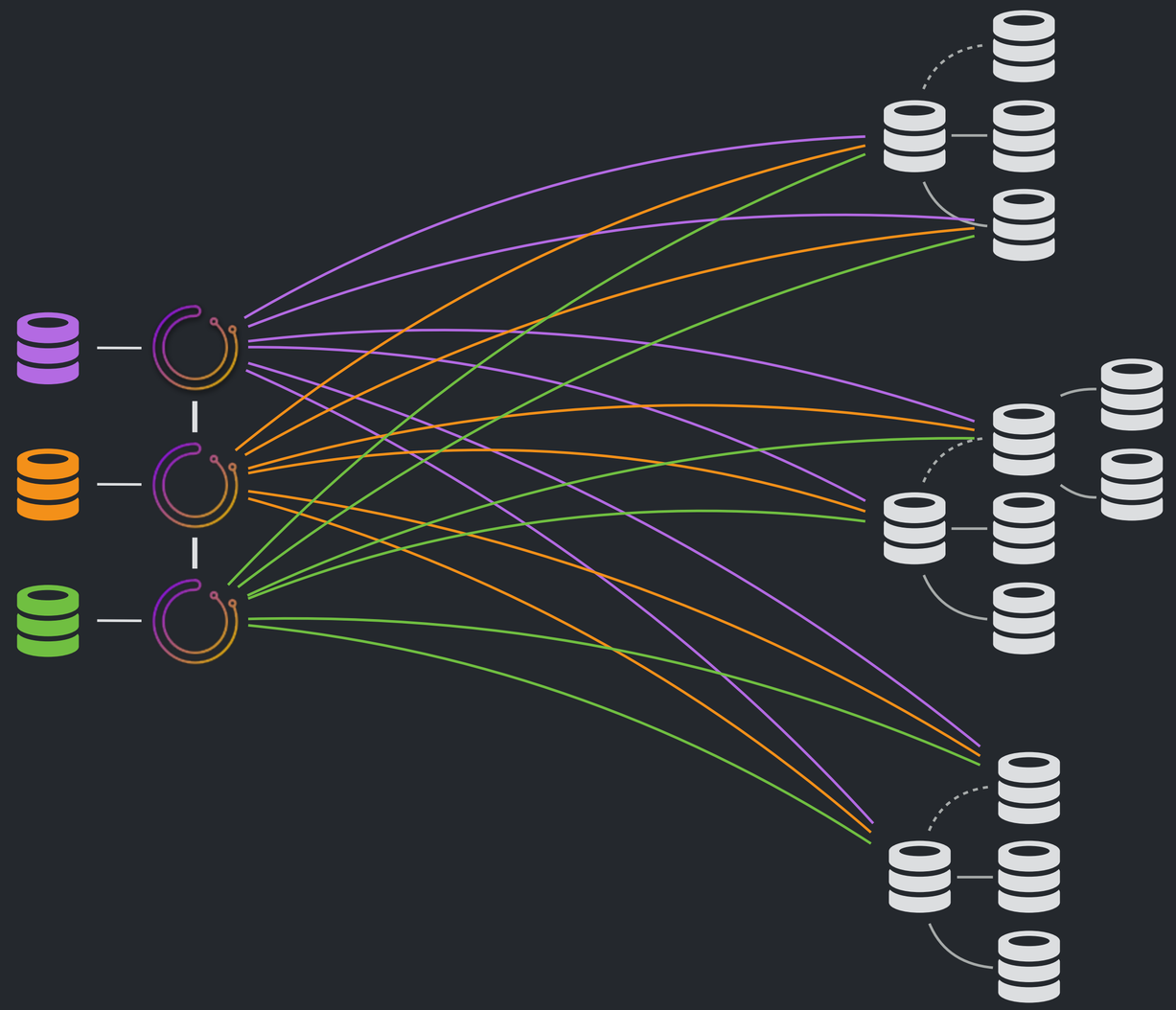
Availability is critical

orchestrator, meta

Adopted, maintained & supported by GitHub,
github.com/github/orchestrator

Previously at Outbrain and Booking.com

Orchestrator is free and open source, released under the Apache 2.0 license
github.com/github/orchestrator/releases

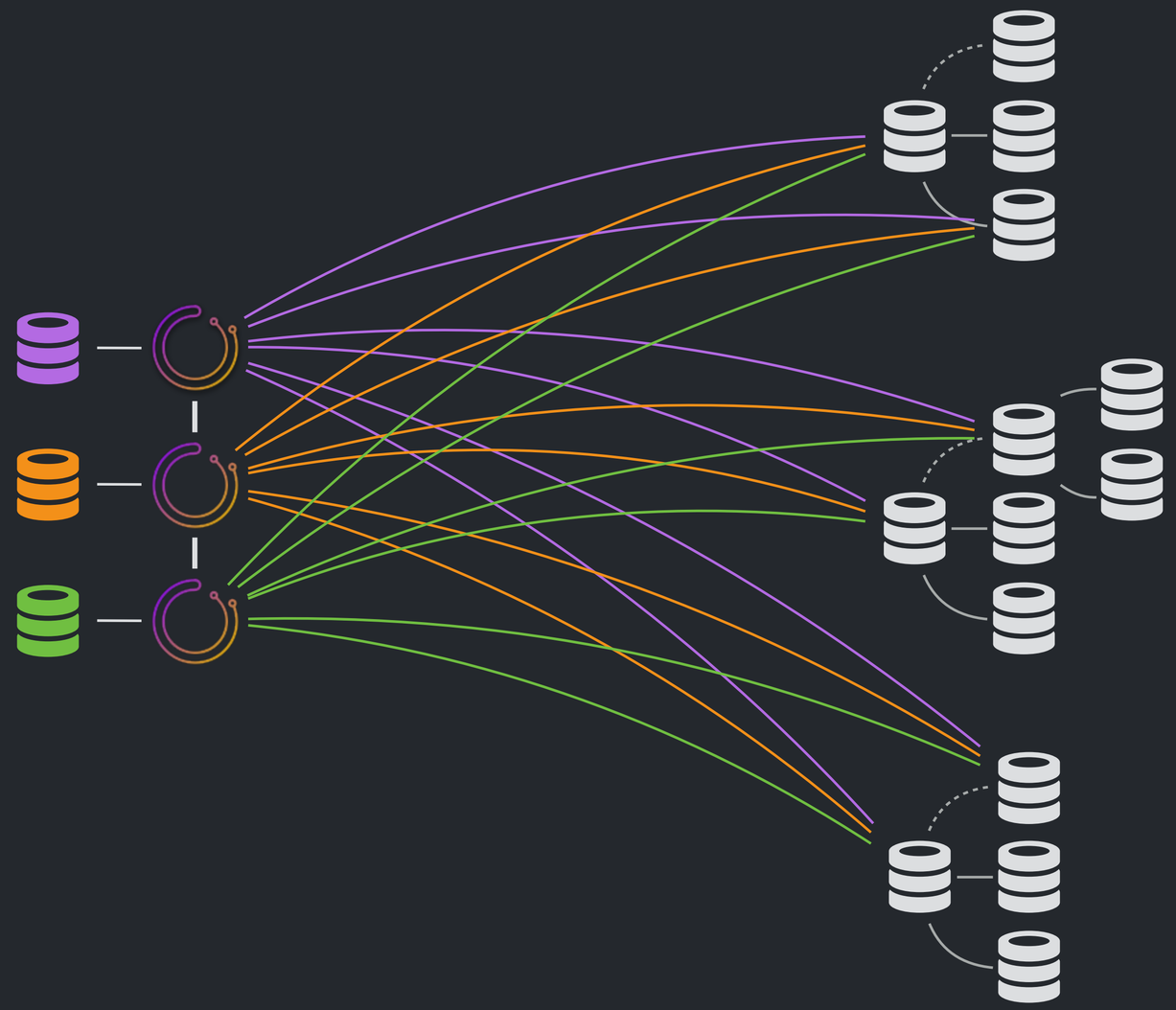


orchestrator

Discovery
Probe, read instances, build topology graph, attributes, queries

Refactoring
Relocate replicas, manipulate, detach, reorganize

Recovery
Analyze, detect crash scenarios, structure warnings, failovers, promotions, acknowledgements, flap control, downtime, hooks



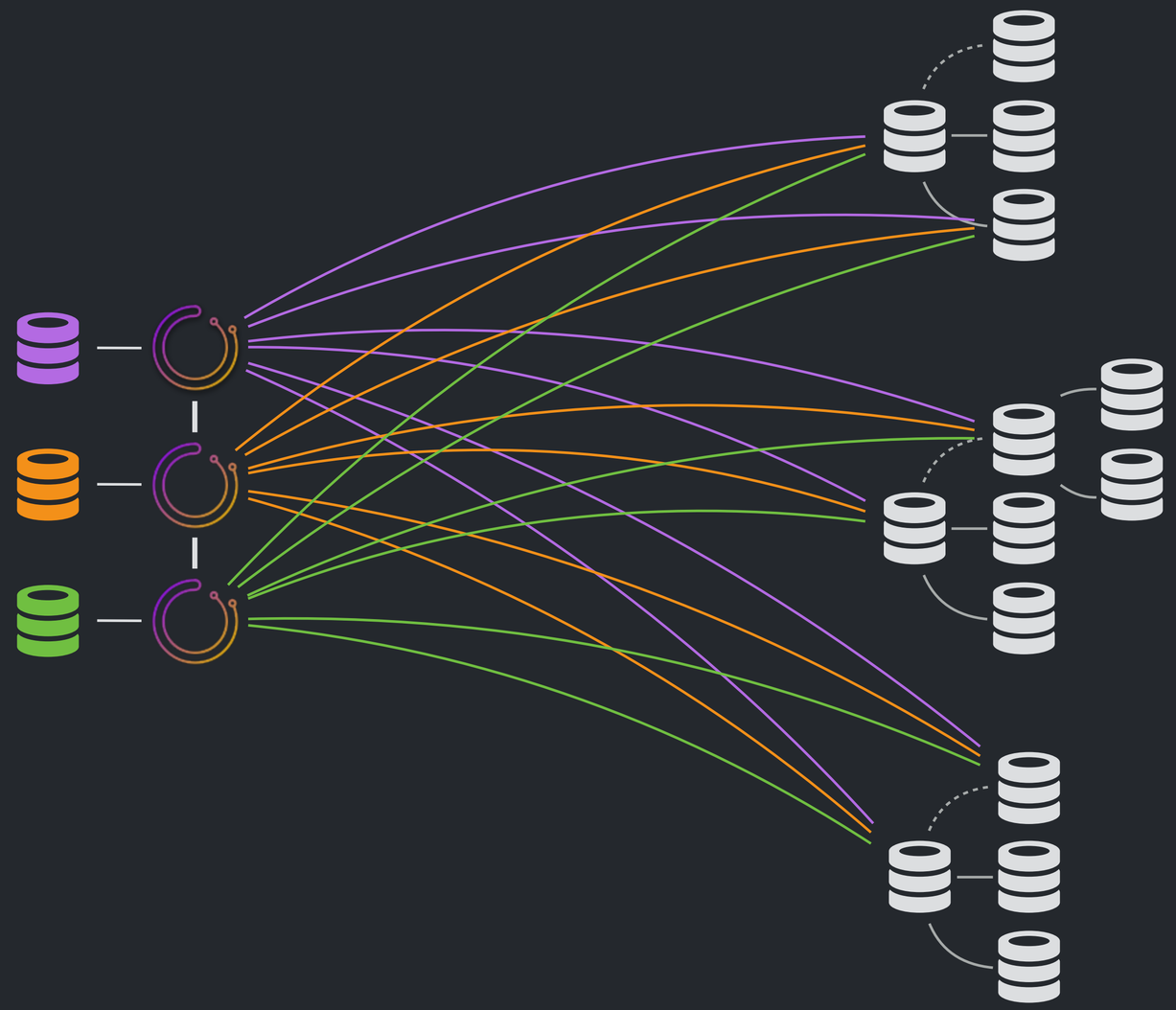
orchestrator/raft

A highly available **orchestrator** setup

Self healing

Cross DC

Mitigates DC partitioning



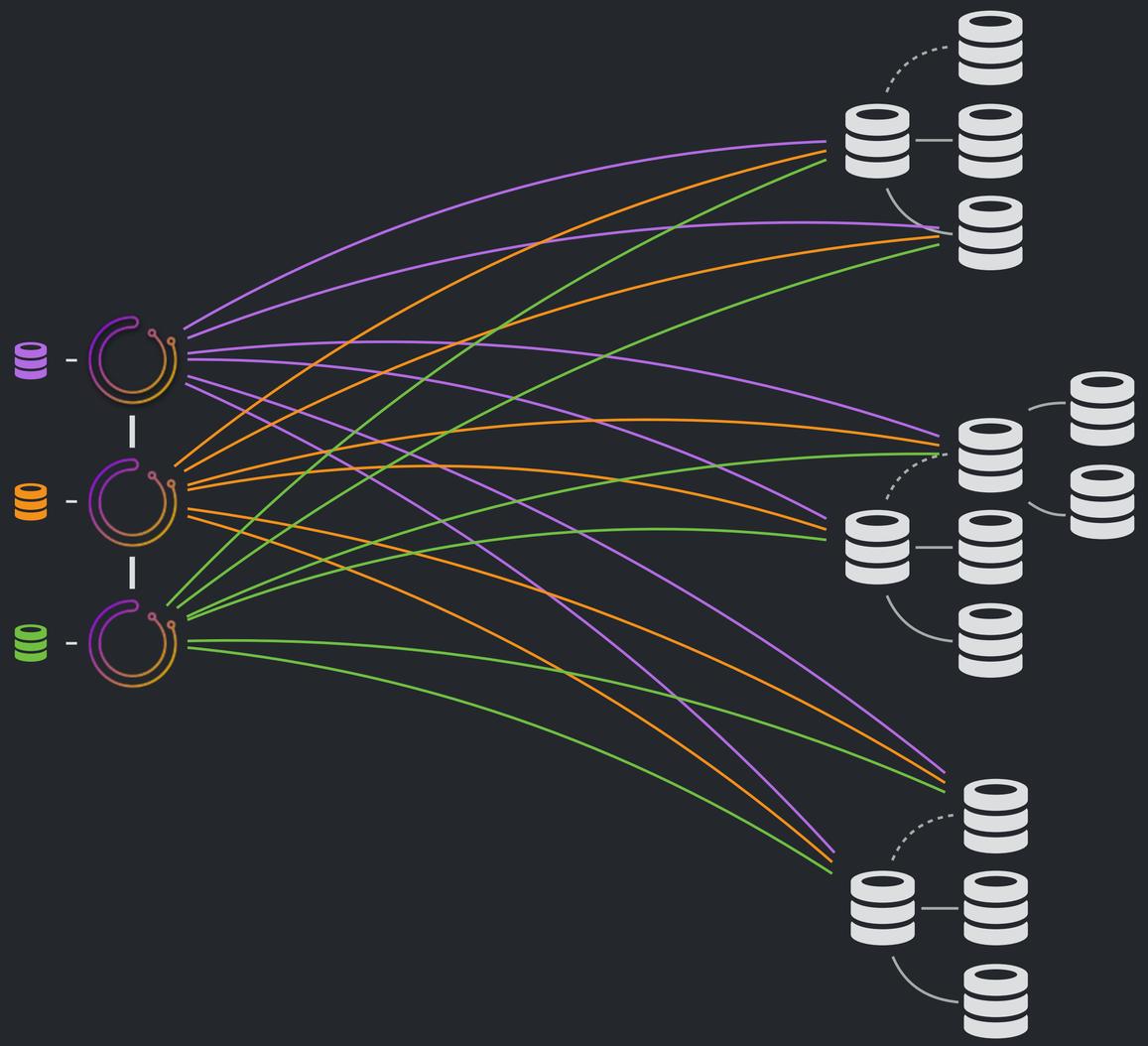
orchestrator/raft/sqlite

Self contained **orchestrator** setup

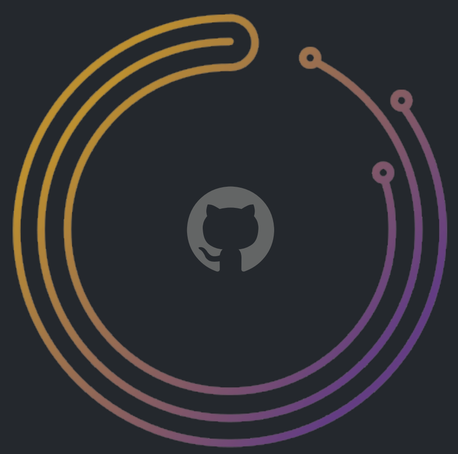
No MySQL backend

Lightweight deployment

Kubernetes friendly



orchestrator @ GitHub



orchestrator/raft deployed on 3 DCs

Automated failover for masters and intermediate masters

Chatops integration

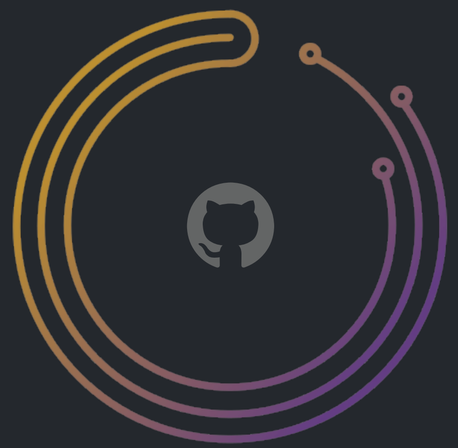
Recently instated a orchestrator/consul/proxy setup for HA and master discovery

Setting up

Configuration for:

- Backend
- Probing/discovering MySQL topologies

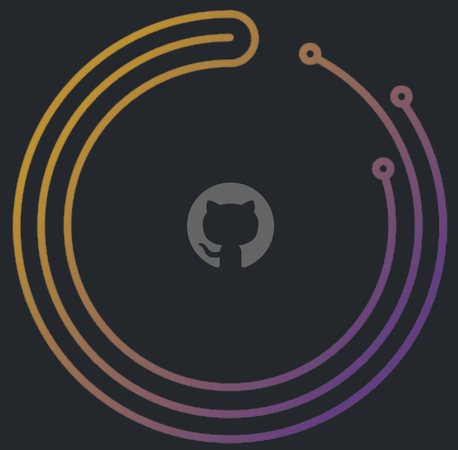
Basic configuration



```
"Debug": true,
"ListenAddress": ":3000",
```

https://github.com/github/orchestrator/blob/master/docs/configuration-backend.md

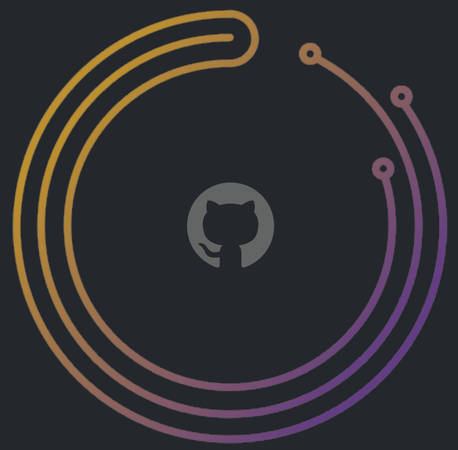
Basic configuration, SQLite



```
"BackendDB": "sqlite",
"SQLite3DataFile": "/var/lib/orchestrator/
orchestrator.db",
```

https://github.com/github/orchestrator/blob/master/docs/configuration-backend.md

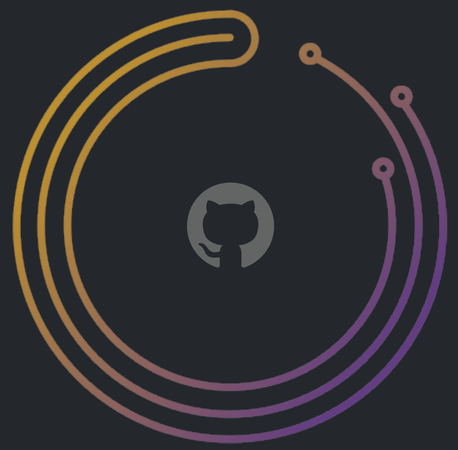
Basic configuration, MySQL



```
"MySQLOrchestratorHost": "127.0.0.1",
"MySQLOrchestratorPort": 3306,
"MySQLOrchestratorDatabase": "orchestrator",

"MySQLTopologyCredentialsConfigFile":
  “/etc/mysql/my.orchestrator.cnf“,
```

https://github.com/github/orchestrator/blob/master/docs/configuration-backend.md

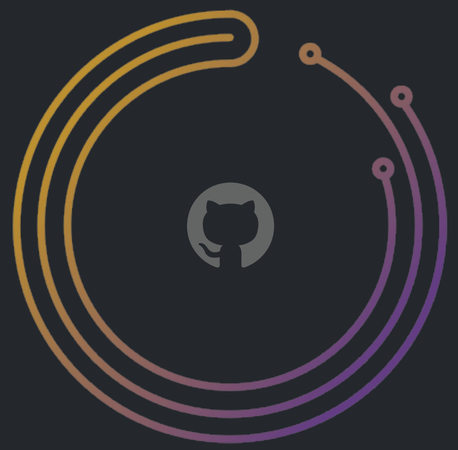
Discovery configuration, local



```
"MySQLTopologyUser": "orc_client_user",
"MySQLTopologyPassword": "123456",

"DiscoverByShowSlaveHosts": true,
"InstancePollSeconds": 5,

“HostnameResolveMethod": "default",
"MySQLHostnameResolveMethod": "@@report_host",
```

https://github.com/github/orchestrator/blob/master/docs/configuration-discovery-basic.md
https://github.com/github/orchestrator/blob/master/docs/configuration-discovery-resolve.md

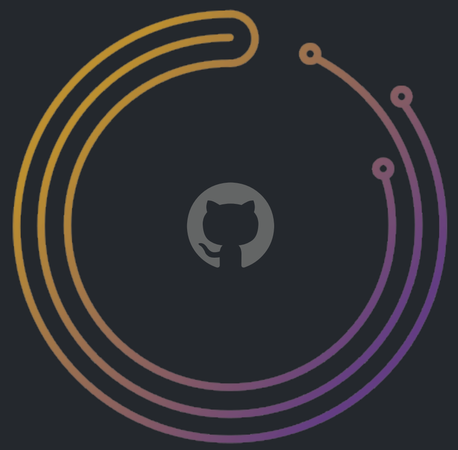
Discovery configuration, prod



```
“MySQLTopologyCredentialsConfigFile": “/etc/mysql/
my.orchestrator-backend.cnf”,

"DiscoverByShowSlaveHosts": false,
"InstancePollSeconds": 5,

“HostnameResolveMethod": "default",
"MySQLHostnameResolveMethod": "@@hostname",
```

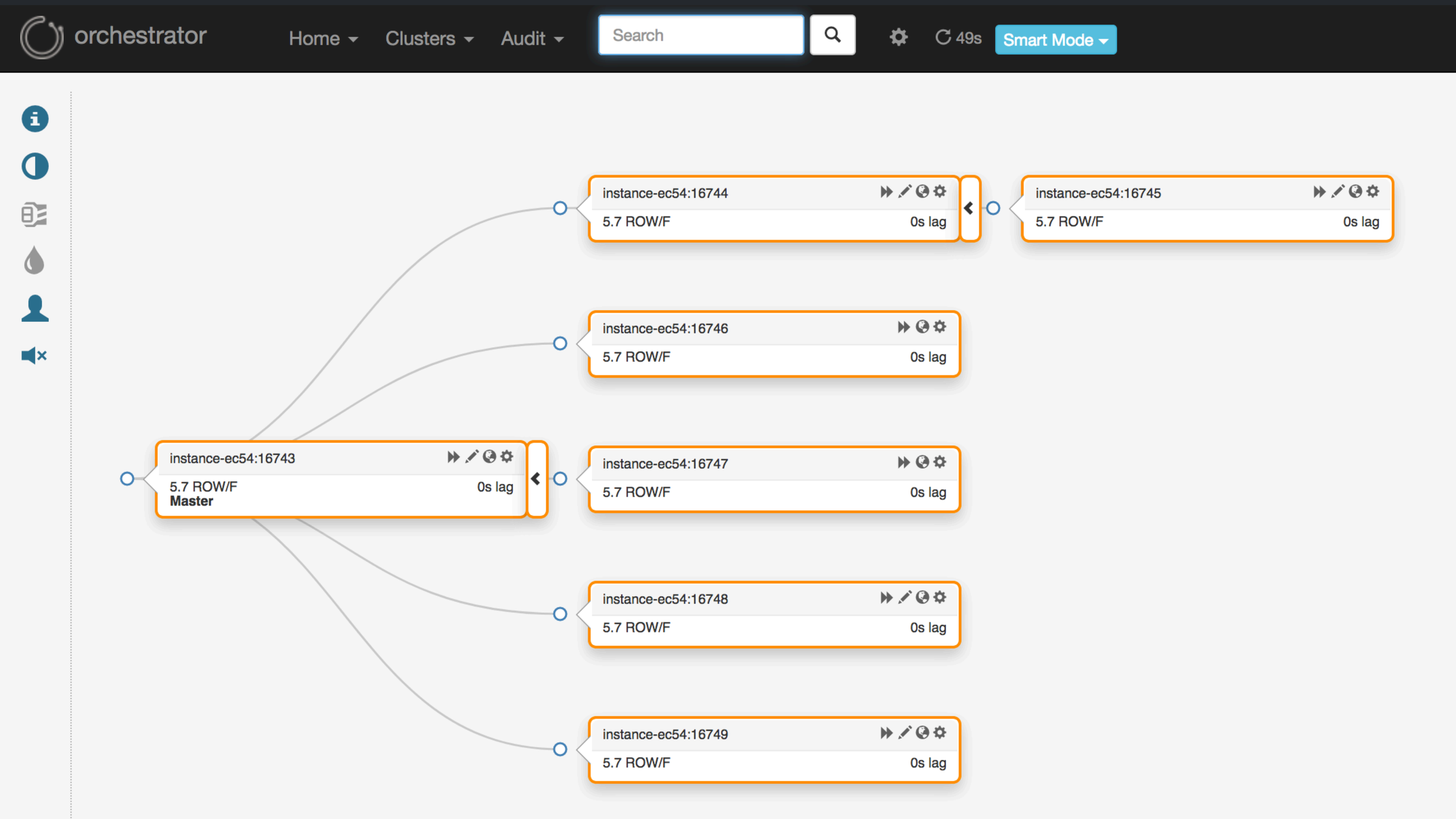
https://github.com/github/orchestrator/blob/master/docs/configuration-discovery-basic.md
https://github.com/github/orchestrator/blob/master/docs/configuration-discovery-resolve.md

Discovery/probe configuration

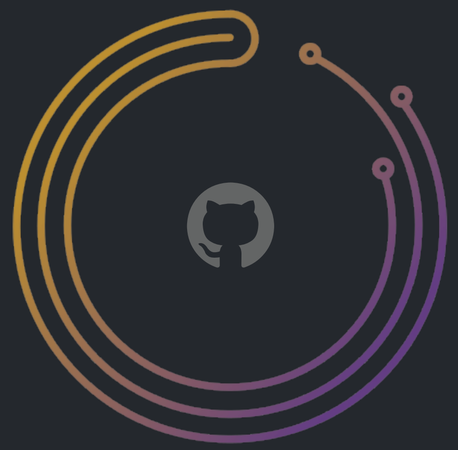


```
"ReplicationLagQuery": "select
  absolute_lag from meta.heartbeat_view",

"DetectClusterAliasQuery": "select
  ifnull(max(cluster_name), '') as cluster_alias
  from meta.cluster where anchor=1",

"DetectDataCenterQuery": "select
  substring_index(
    substring_index(@@hostname, '-',3),
  '-', -1) as dc",
```

https://github.com/github/orchestrator/blob/master/docs/configuration-discovery-classifying.md


orchestrator
Home
Clusters
Audit
Search
49s
Smart Mode
instance-ec54:16744
5.7 ROW/F
0s lag
instance-ec54:16745
5.7 ROW/F
0s lag
instance-ec54:16746
5.7 ROW/F
0s lag
instance-ec54:16743
5.7 ROW/F
Master
0s lag
instance-ec54:16747
5.7 ROW/F
0s lag
instance-ec54:16748
5.7 ROW/F
0s lag
instance-ec54:16749
5.7 ROW/F
0s lag

Detection & recovery primer



What's so complicated about detection & recovery?

How is orchestrator different than other solutions?

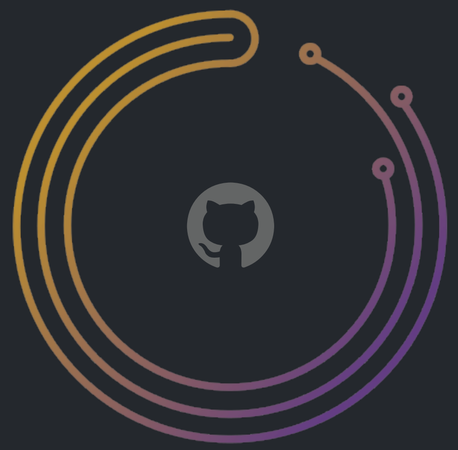
What makes a reliable detection?

What makes a successful recovery?

Which parts of the recovery does orchestrator own?

What about the parts it doesn't own?

Detection



Runs at all times

Some tools: dead master detection

Common failover tools only observe per-server health.

If the master cannot be reached, it is considered to be dead.

To avoid false positives, some introduce repetitive checks + intervals.

e.g. check every 5 seconds and if seen dead for 4 consecutive times, declare “death”

This heuristically reduces false positives, and introduces recovery latency.

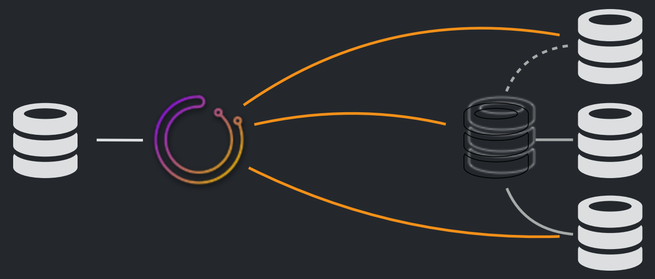
Detection

orchestrator continuously probes all MySQL topology servers

At time of crash, orchestrator knows what the topology should look like, because it knows how it looked like a moment ago

What insights can orchestrator draw from this fact?

Detection: dead master, holistic approach

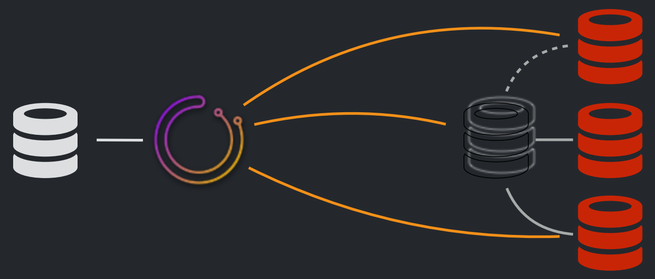


orchestrator uses a holistic approach. It harnesses the topology itself.

orchestrator observes the master and the replicas.

If the master is unreachable, but all replicas are happy, then there's no failure. It may be a network glitch.

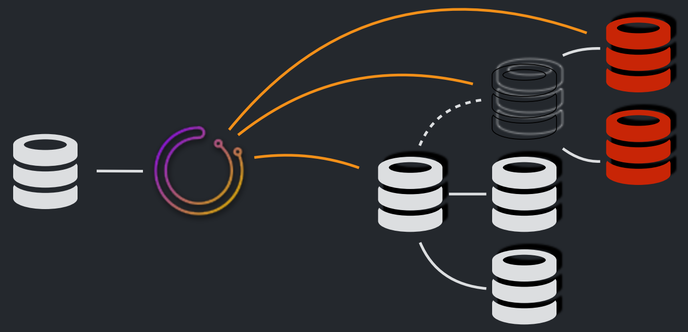
Detection: dead master, holistic approach



If the master is unreachable, and all of the replicas are in agreement (replication broken), then declare “death”.

There is no need for repetitive checks. Replication broke on all replicas due to a reason, and following its own timeout.

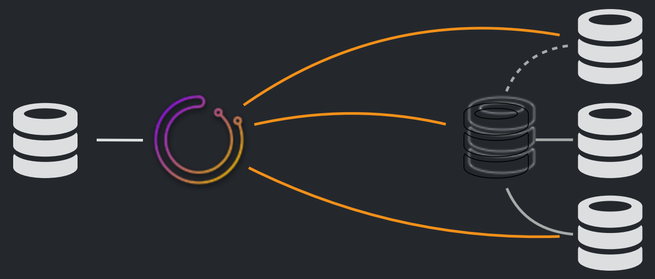
Detection:
dead intermediate master



orchestrator uses exact same holistic approach logic

If intermediate master is unreachable and its replicas are broken, then declare “death”

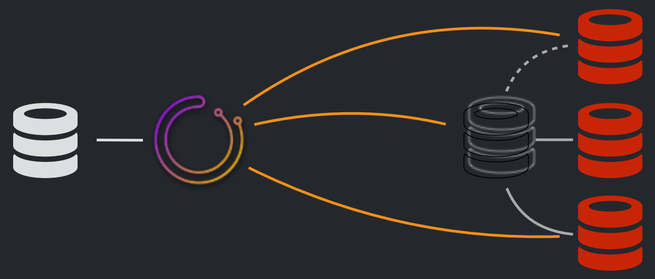
Detection: holistic approach



False positives extremely low

Some cases left for humans to handle

Faster detection: MySQL config

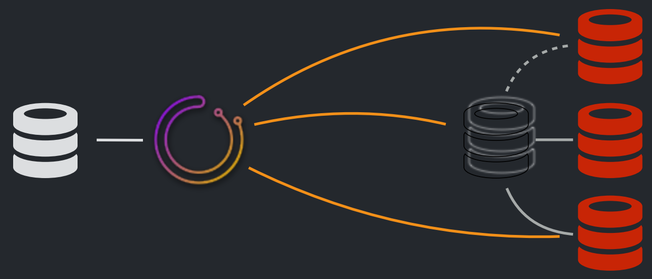


```
set global slave_net_timeout = 4;
```

Implies:

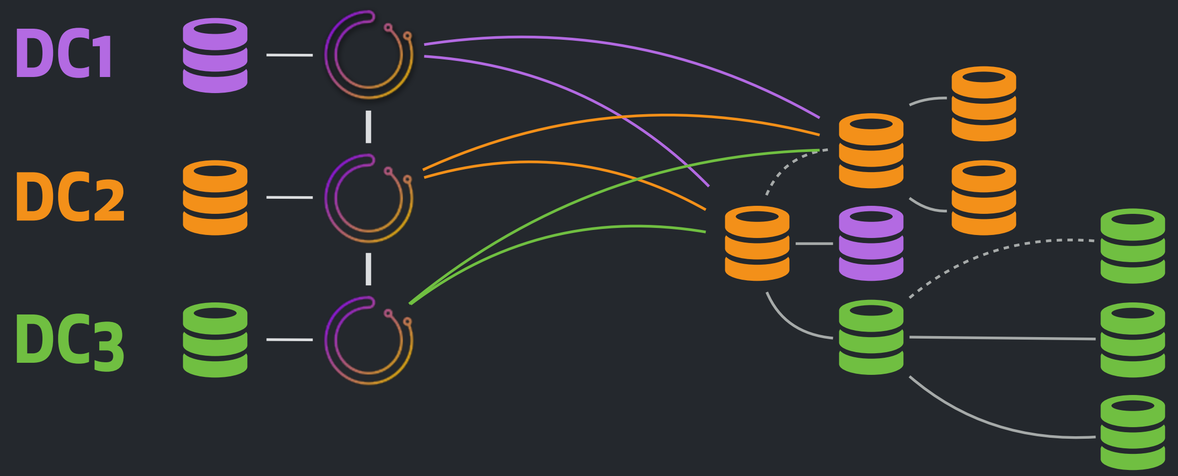
```
master_heartbeat_period = 2
```

Faster detection: MySQL config



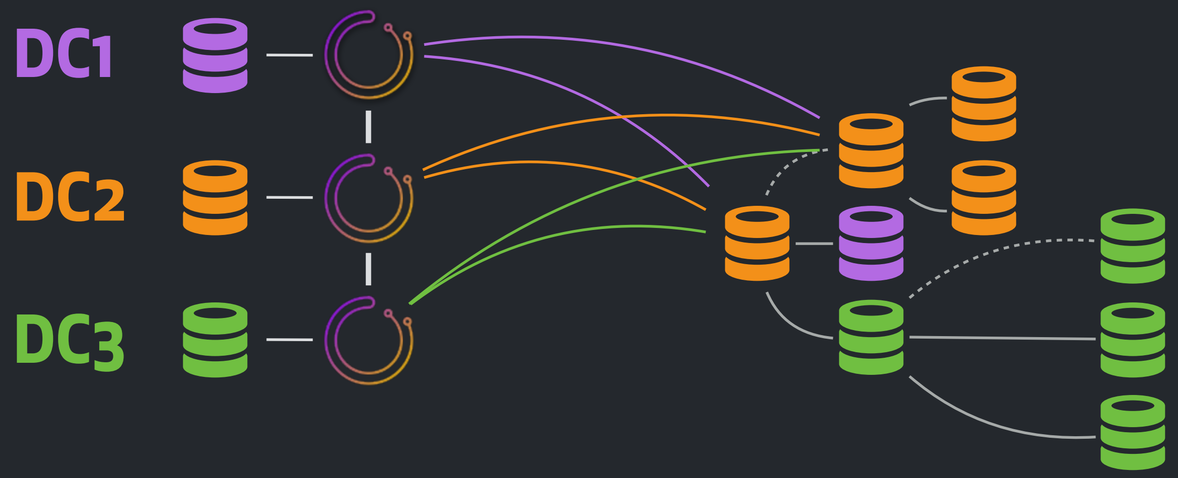
```
change master to
  MASTER_CONNECT_RETRY = 1
  MASTER_RETRY_COUNT = 86400
```

Detection: DC fencing



orchestrator/raft detects and responds to DC fencing (DC network isolation)

Detection: DC fencing



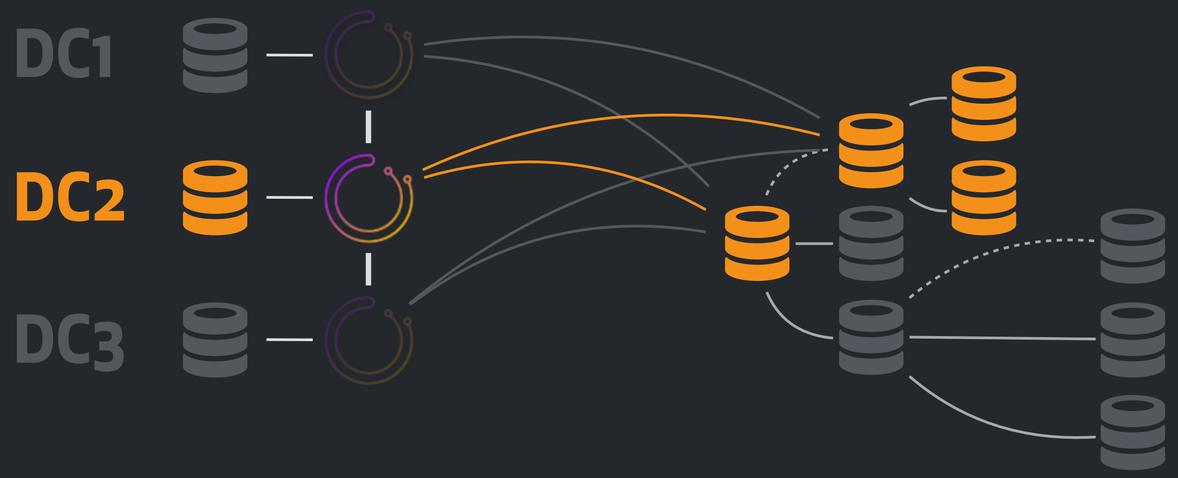
Assume this 3 DC setup:

One orchestrator node in each DC,

Master and a few replicas in **DC2**.

What happens if **DC2** gets network partitioned? i.e. no network in or out **DC2**

Detection: DC fencing



From the point of view of **DC2** servers, and in particular in the point of view of **DC2**'s orchestrator node:

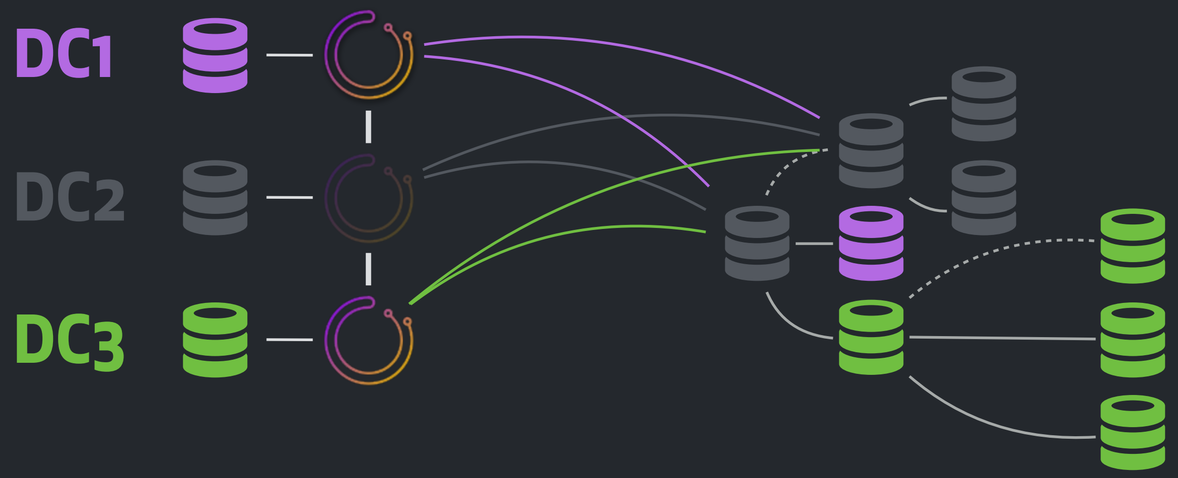
Master and replicas are fine.

DC1 and **DC3** servers are all dead.

No need for fail over.

However, **DC2**'s orchestrator is not part of a quorum, hence not the leader. It doesn't call the shots.

Detection: DC fencing



In the eyes of either **DC1**'s or **DC3**'s orchestrator:

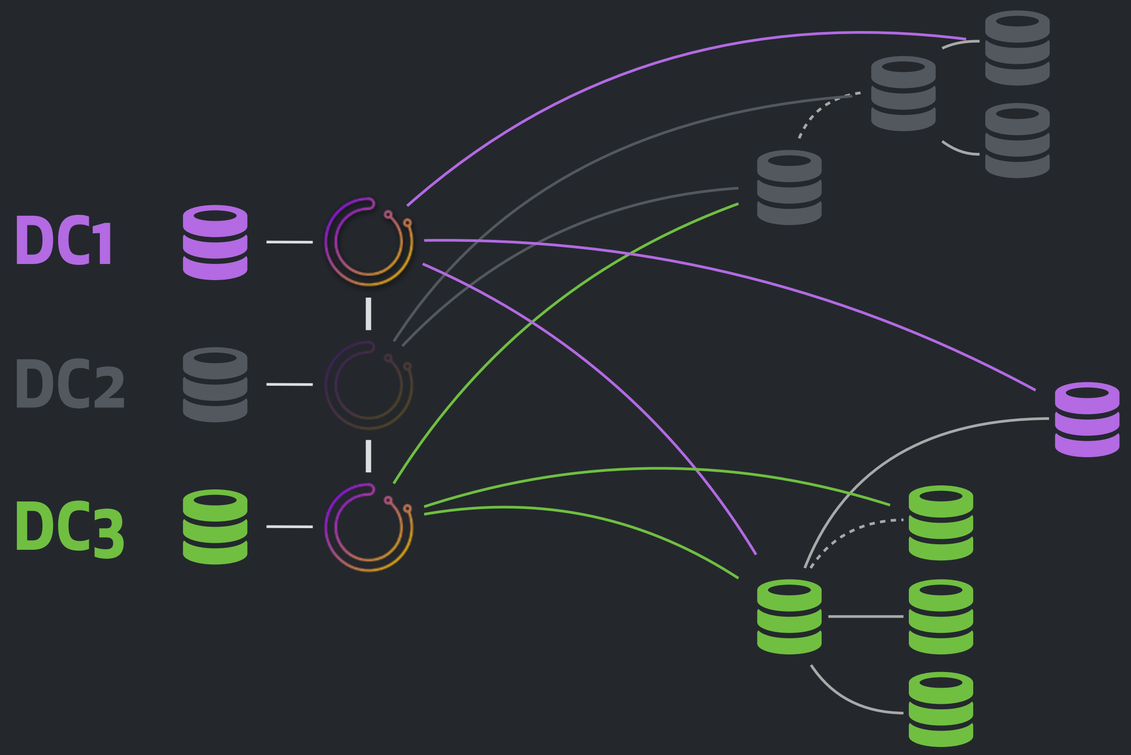
All **DC2** servers, including the master, are dead.

There is need for failover.

DC1's and **DC3**'s orchestrator nodes form a quorum. One of them will become the leader.

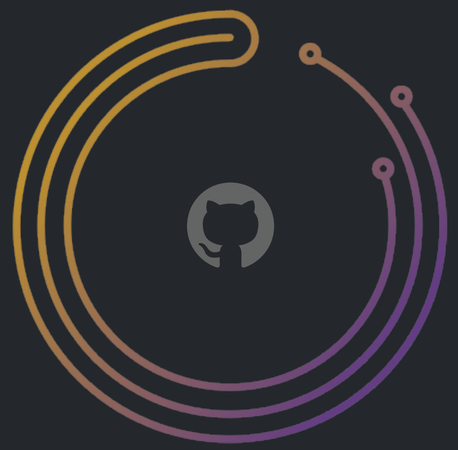
The leader will initiate failover.

Detection: DC fencing



Depicted potential failover result. New master is from **DC3**.

Recovery & promotion constraints



You've made the decision to promote a new master

Which one?

Are all options valid?

Is the current state what you think the current state is?

Recovery & promotion constraints

~~Promote the most up-to-date replica~~

An anti-pattern

Promotion constraints

most up to date

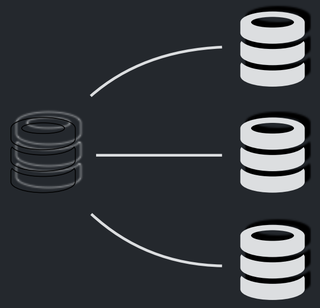
less up to date

delayed 24 hours

You wish to promote the most up to date replica,

otherwise you give up on any replica that is more advanced

Promotion constraints



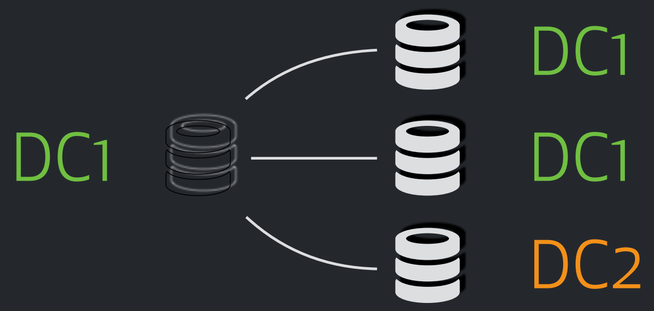
log_slave_updates

log_slave_updates

no binary logs

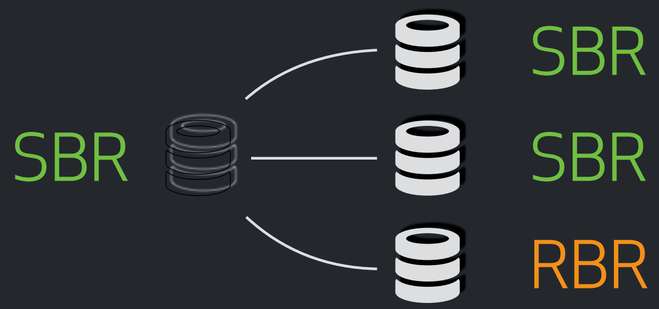
You must not promote a replica that has no binary logs, or without `log_slave_updates`

Promotion constraints



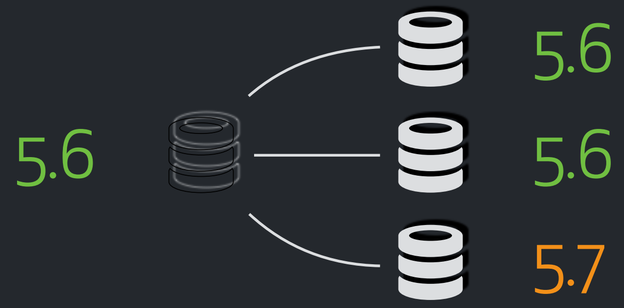
You prefer to promote a replica from same DC as failed master

Promotion constraints



You must not promote Row Based Replication server on top of Statement Based Replication

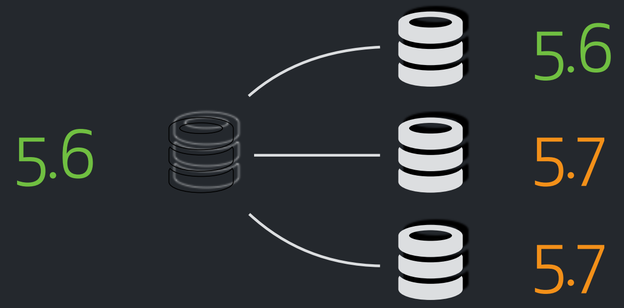
Promotion constraints



Promoting 5.7 means losing 5.6 (replication not forward compatible)

So Perhaps worth losing the 5.7 server?

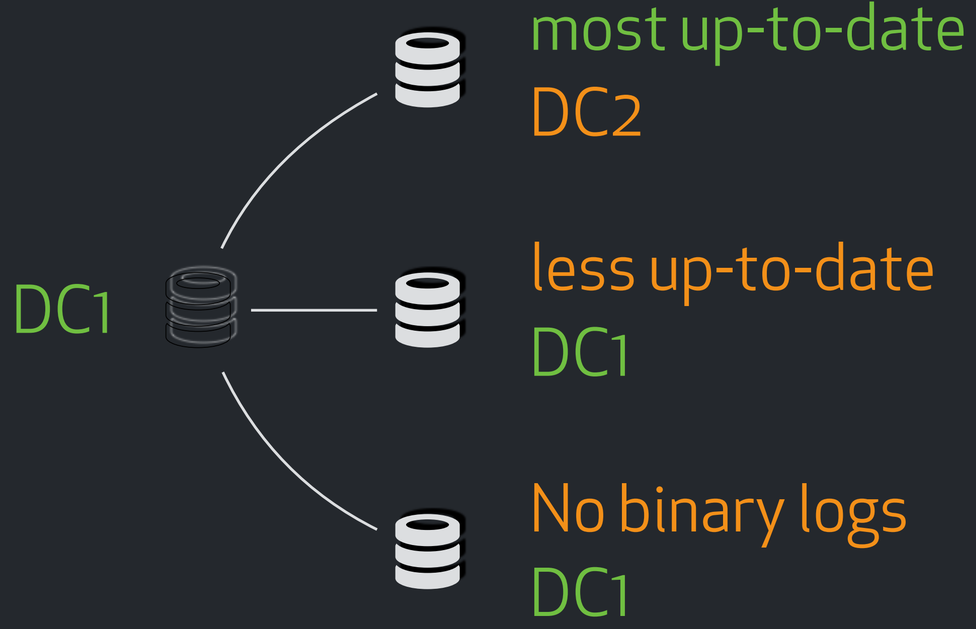
Promotion constraints



But if most of your servers are 5.7, and 5.7 turns to be most up to date, better promote 5.7 and drop the 5.6

Orchestrator handles this logic and prioritizes promotion candidates by overall count and state of replicas

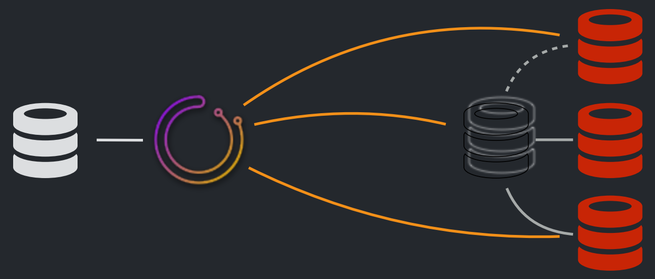
Promotion constraints: real life



Orchestrator can promote one, non-ideal replica, have the rest of the replicas converge,

and then *refactor again*, promoting an *ideal* server.

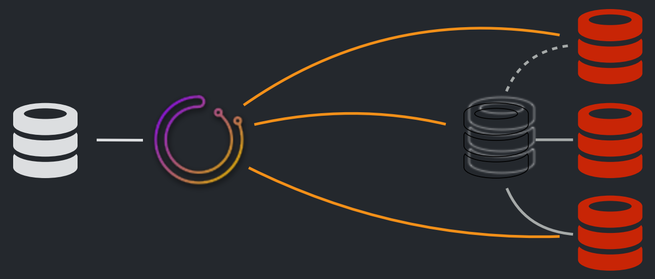
Other tools: MHA



Avoids the problem by syncing relay logs.

Identity of replica-to-promote dictated by config. No state-based resolution.

Other tools: replication-manager



Potentially uses *flashback*, unapplying binlog events. This works on MariaDB servers.
https://www.percona.com/blog/2018/04/12/point-in-time-recovery-pitr-in-mysql-mariadb-percona-server/

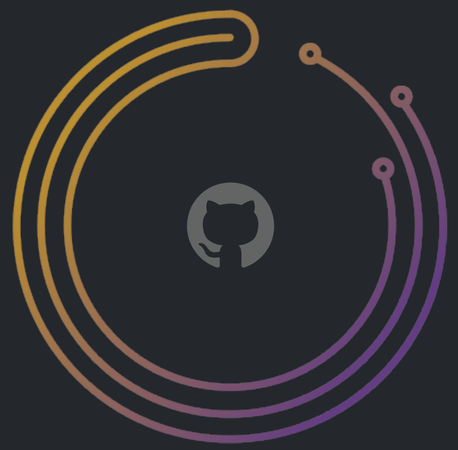
No state-based resolution.

Recovery & promotion constraints

More on the complexity of choosing a recovery path:

http://code.openark.org/blog/mysql/whats-so-complicated-about-a-master-failover

Recovery, meta



Flapping

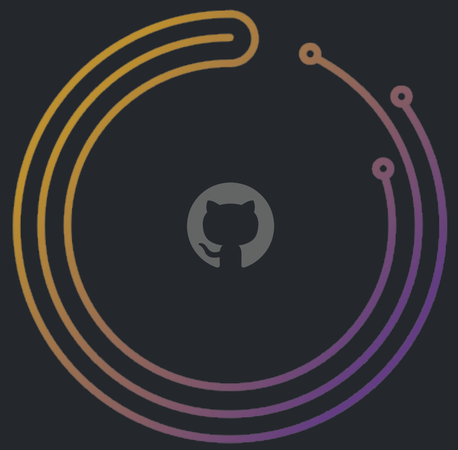
Acknowledgements

Audit

Downtime

Promotion rules

Recovery, flapping



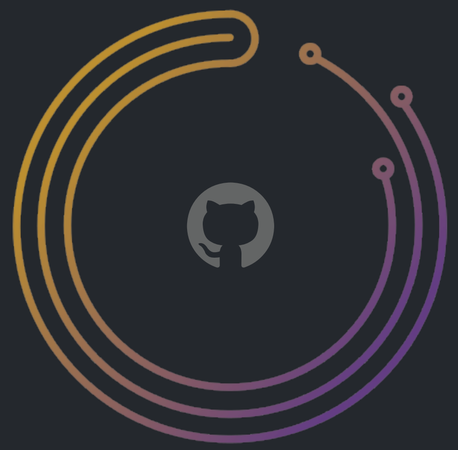
```
"RecoveryPeriodBlockSeconds": 3600,
```

Sets minimal period between two automated recoveries on same cluster.

Avoid server exhaustion on grand disasters.

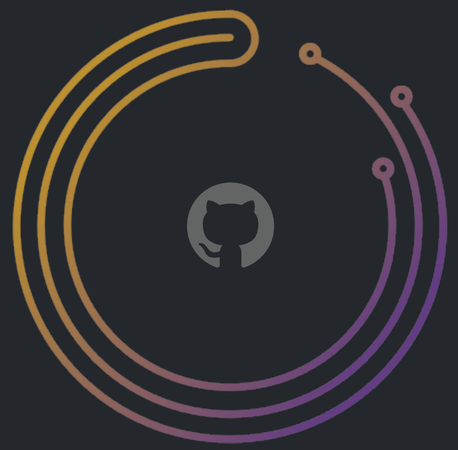
A human may acknowledge.

Recovery, acknowledgements



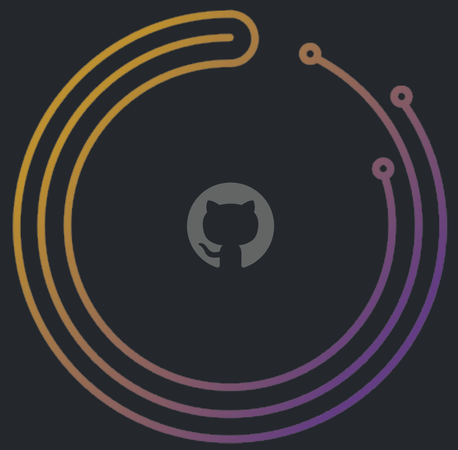
```
$ orchestrator-client -c ack-cluster-recoveries
  -alias mycluster -reason “testing”

$ orchestrator-client -c ack-cluster-recoveries
  -i instance.in.cluster.com -reason “fixed it”

$ orchestrator-client -c ack-all-recoveries
  -reason “I know what I’m doing”
```

Recovery, audit



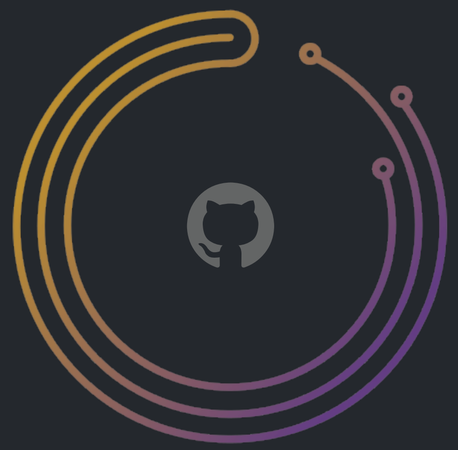
```
/web/audit-failure-detection

/web/audit-recovery

/web/audit-recovery/alias/mycluster

/web/audit-recovery-steps/
15208578417543688804:73fdd23f0415dc3f96f57dd4
c32d2d1d8ff829572428c7be3e796aec895e2ba1
```

Recovery, audit



```
/api/audit-failure-detection

/api/audit-recovery

/api/audit-recovery/alias/mycluster

/api/audit-recovery-steps/
15208578417543688804:73fdd23f0415dc3f96f57dd4
c32d2d1d8ff829572428c7be3e796aec895e2ba1
```

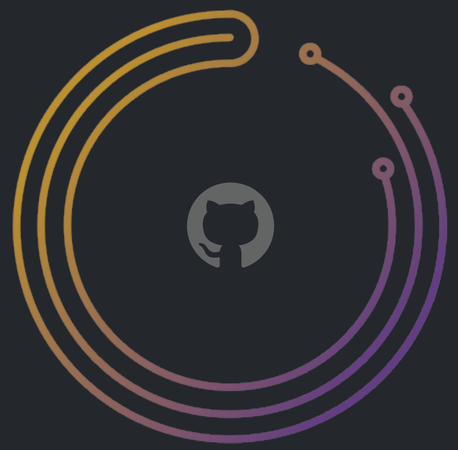
Recovery, downtime



```
$ orchestrator-client -c begin-downtime
  -i my.instance.com
  -duration 30m -reason "experimenting"
```

orchestrator will not auto-failover downtimed servers

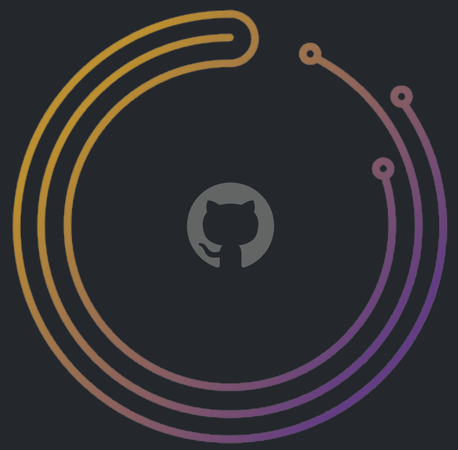
Recovery, downtime



On automated failovers, **orchestrator** will mark dead or lost servers as downtimed.

Reason is set to **lost-in-recovery**.

Recovery, promotion rules



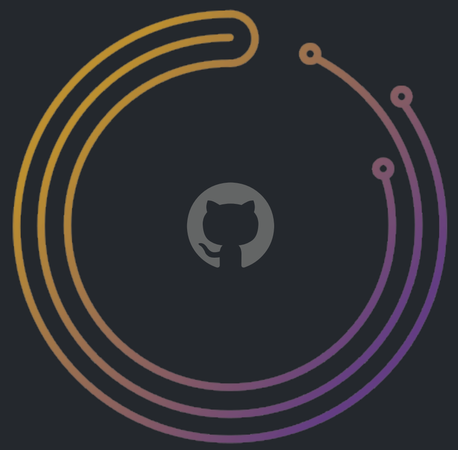
orchestrator takes a dynamic approach as opposed to a configuration approach.

You may have “preferred” replicas to promote. You may have replicas you don’t want to promote.

You may indicate those to **orchestrator** dynamically, and/or change your mind, without touching configuration.

Works well with puppet/chef/ansible.

Recovery, promotion rules

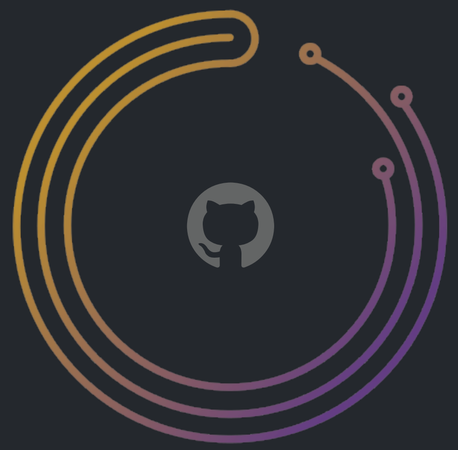


```
$ orchestrator-client -c register-candidate
  -i my.instance.com
  -promotion-rule=prefer
```

Options are:

- `prefer`
- `neutral`
- `prefer_not`
- `must_not`

Recovery, promotion rules



- `prefer`
  `If possible, promote this server`
- `neutral`
- `prefer_not`
  `Can be used in two-step promotion`
- `must_not`
  `Dirty, do not even use`

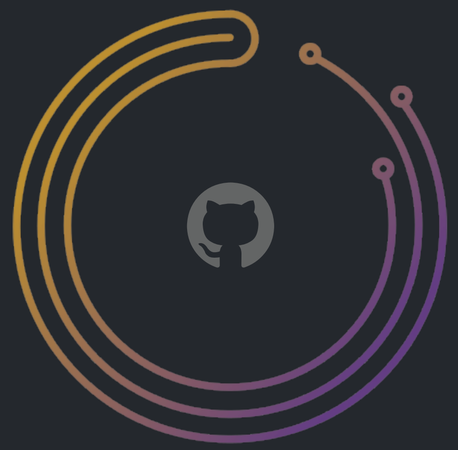
Examples: we set **prefer** for servers with better raid setup.
prefer_not for backup servers or servers loaded with other tasks.
must_not for *gh-ost* testing servers

Failovers

orchestrator supports:

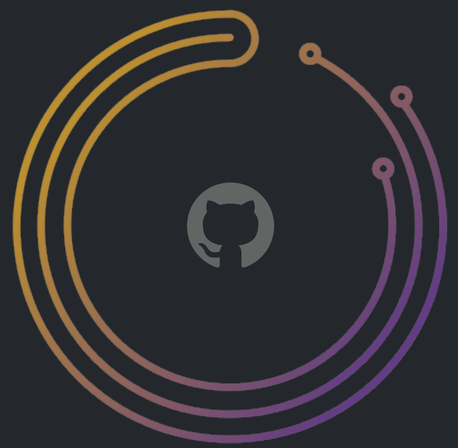
- Automated master & intermediate master failovers
- Manual master & intermediate master failovers per detection
- Graceful (manual, planned) master takeovers
- Panic (user initiated) master failovers

Failover configuration



```
"RecoverMasterClusterFilters": [
  “opt-in-cluster“,
  “another-cluster”
],

"RecoverIntermediateMasterClusterFilters": [
  "*"
],
```

Failover configuration



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"MasterFailoverLostInstancesDowntimeMinutes": 10,
"FailMasterPromotionIfSQLThreadNotUpToDate": true,
"DetachLostReplicasAfterMasterFailover": true,
```

Special note for `ApplyMySQLPromotionAfterMasterFailover`:

```
RESET SLAVE ALL
SET GLOBAL read_only = 0
```

Failover configuration

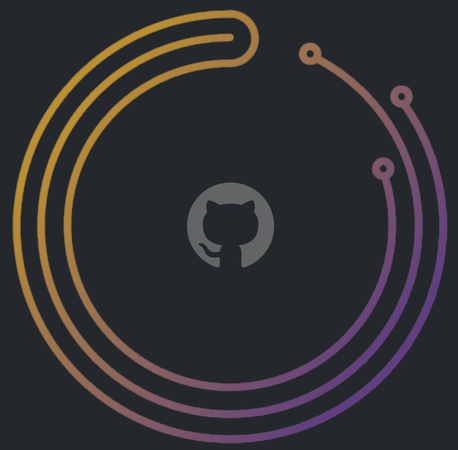
```
"PreGracefulTakeoverProcesses": [],
"PreFailoverProcesses": [
  "echo 'Will recover from {failureType} on {failureCluster}’ >> /tmp/recovery.log"
],

"PostFailoverProcesses": [
  "echo '(for all types) Recovered from {failureType} on {failureCluster}.
    Failed: {failedHost}:{failedPort}; Successor: {successorHost}:{successorPort}'
    >> /tmp/recovery.log"
],
"PostUnsuccessfulFailoverProcesses": [],
"PostMasterFailoverProcesses": [
  "echo 'Recovered from {failureType} on {failureCluster}. Failed: {failedHost}:
    {failedPort}; Promoted: {successorHost}:{successorPort}' >> /tmp/recovery.log"
],
"PostIntermediateMasterFailoverProcesses": [],
"PostGracefulTakeoverProcesses": [],
```

$1M Question

What do you use for your pre/post failover hooks?

To be discussed and demonstrated shortly.

KV configuration



```
"KVClusterMasterPrefix": "mysql/master",
"ConsulAddress": "127.0.0.1:8500",
"ZkAddress": "srv-a,srv-b:12181,srv-c",
```

ZooKeeper not implemented yet (v3.0.10)

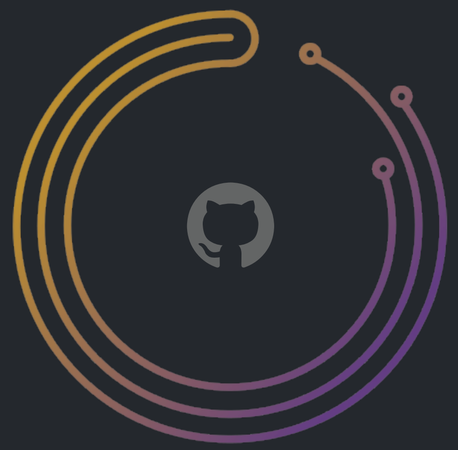
orchestrator updates KV stores at each failover

KV contents

```
$ consul kv get -recurse mysql

mysql/master/orchestrator-ha:my.instance-13ff.com:3306
mysql/master/orchestrator-ha/hostname:my.instance-13ff.com
mysql/master/orchestrator-ha/ipv4:10.20.30.40
mysql/master/orchestrator-ha/ipv6:
mysql/master/orchestrator-ha/port:3306
```

KV writes *successive*, non atomic.

Manual failovers



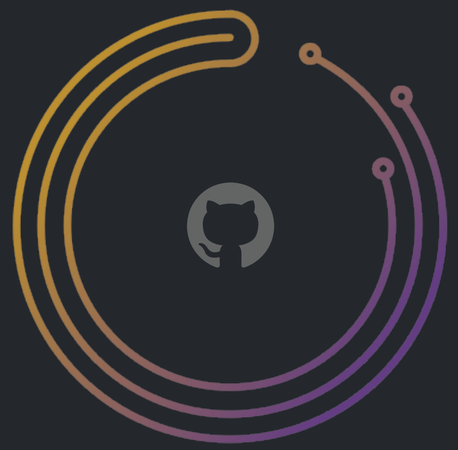
Assuming **orchestrator** agrees there's a problem:

```
orchestrator-client -c recover -i failed.instance.com
```

or via web, or via API

```
/api/recover/failed.instance.com/3306
```

Graceful (planned) master takeover



Initiate a graceful failover.

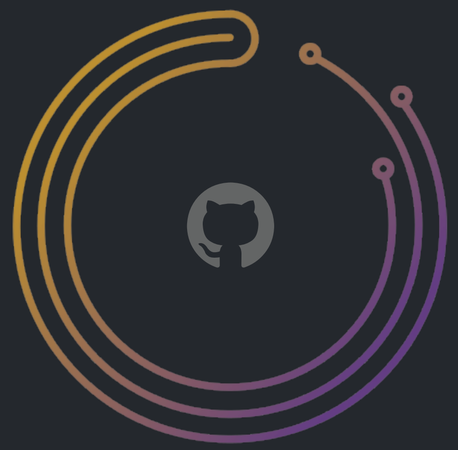
Sets `read_only/super_read_only` on master, promotes replica once caught up.

```
orchestrator-client -c graceful-master-takeover
  -alias mycluster
```

or via web, or via API.

See `PreGracefulTakeoverProcesses`, `PostGracefulTakeoverProcesses` config.

Panic (human operated) master failover



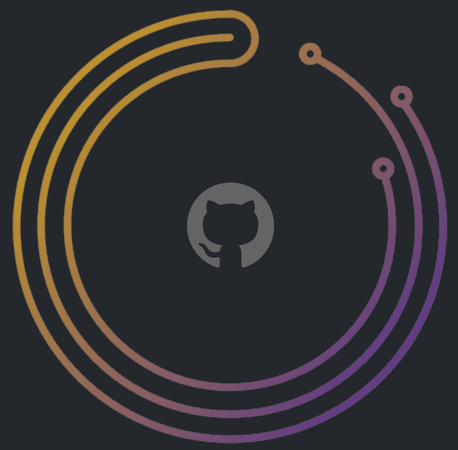
Even if **orchestrator** disagrees there's a problem:

```
orchestrator-client -c force-master-failover
  -alias mycluster
```

or via API.

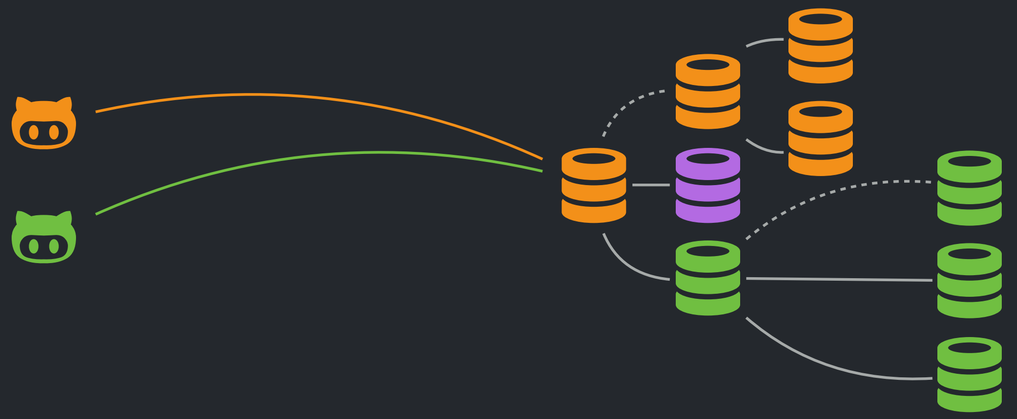
Forces **orchestrator** to initiate a failover as if the master is dead.

Master discovery

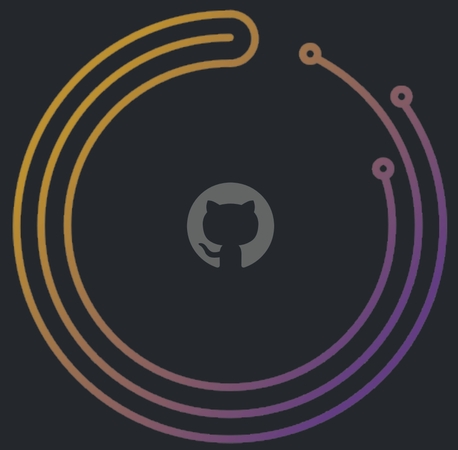


How do applications know which MySQL server is the master?

How do applications learn about master failover?



Master discovery

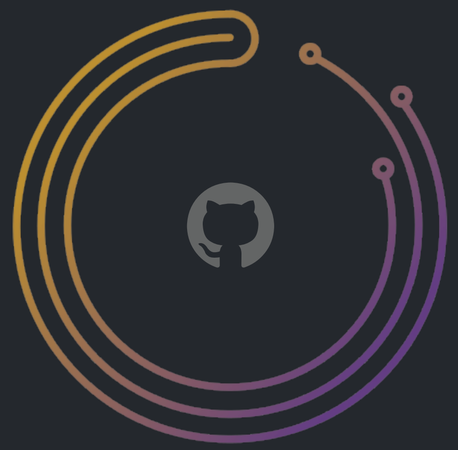


The answer dictates your HA strategy and capabilities.

Master discovery methods

Hard code IPs, DNS/VIP, Service Discovery, Proxy, combinations of the above

Master discovery via hard coded IP address

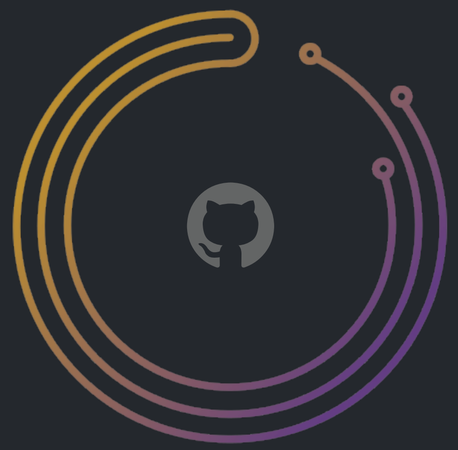


e.g. committing identity of master in config/yml file and distributing via chef/puppet/ansible

Cons:

- Slow to deploy
- Using code for state

Master discovery via DNS



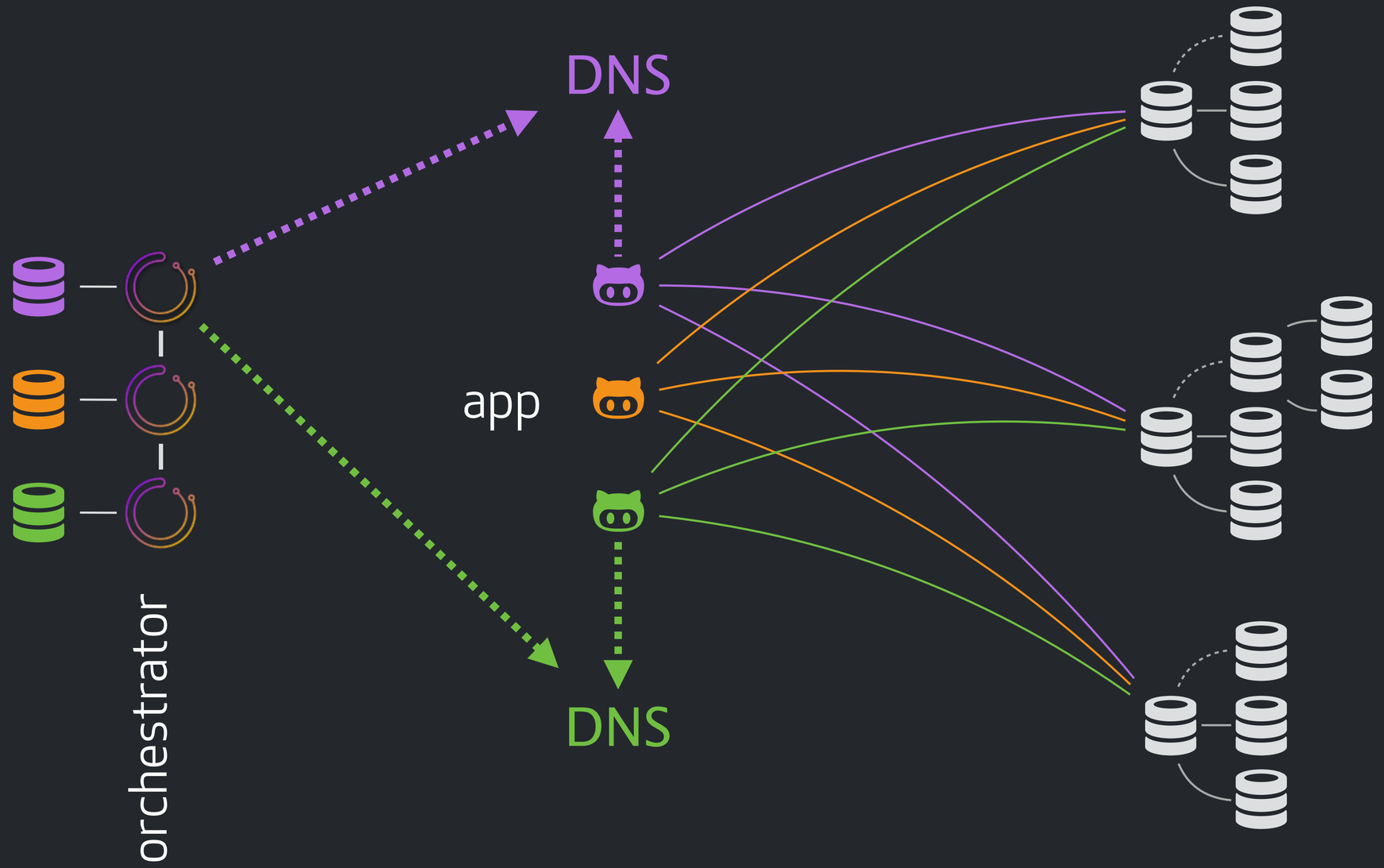
Pros:

- No changes to the app which only knows about the host Name/CNAME
- Cross DC/Zone

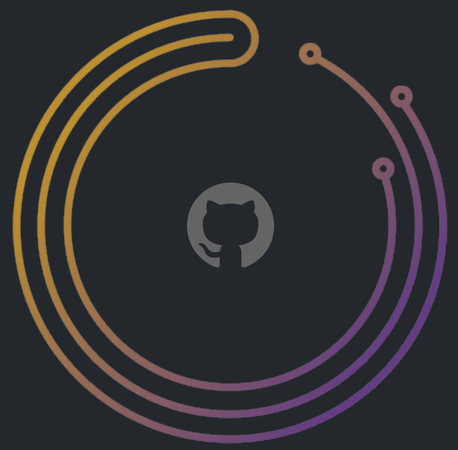
Cons:

- TTL
- Shipping the change to all DNS servers
- Connections to old master potentially uninterrupted

Master discovery via DNS

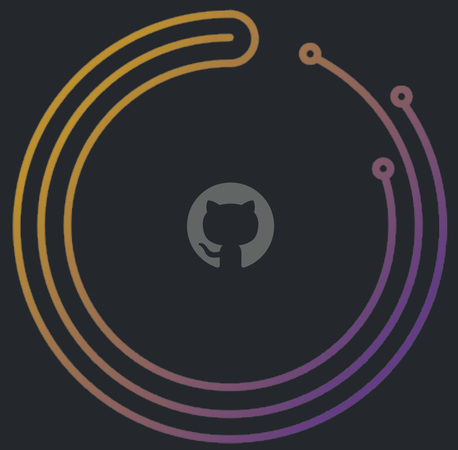


Master discovery via DNS



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"PostMasterFailoverProcesses": [
  "/do/what/you/gotta/do to apply dns change for
{failureClusterAlias}-writer.example.net to {successorHost}"
],
```

Master discovery via VIP



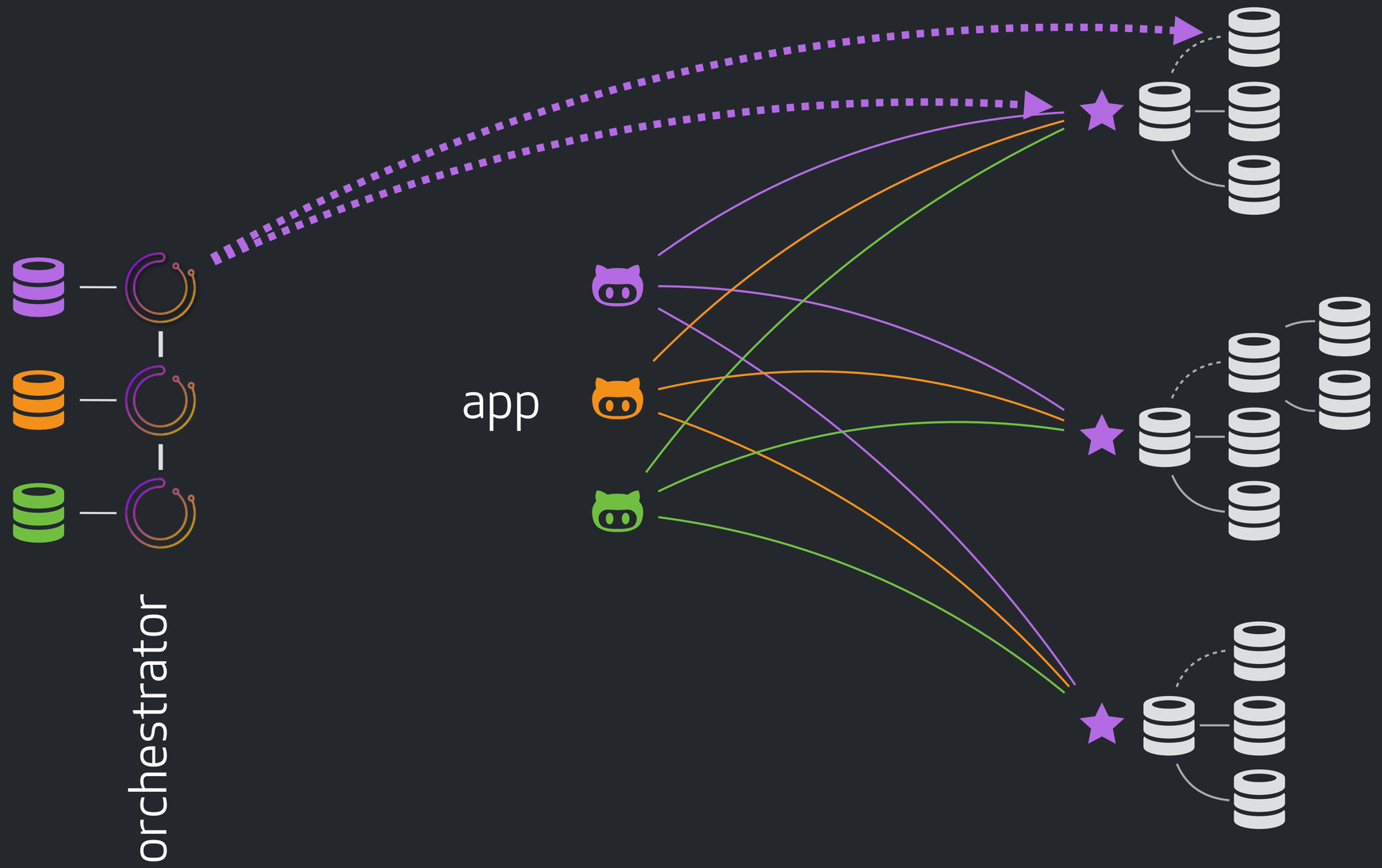
Pros:

- No changes to the app which only knows about the VIP

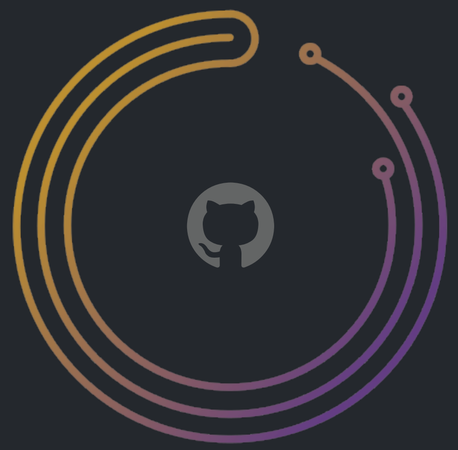
Cons:

- Cooperative assumption
- Remote SSH / Remote exec
- Sequential execution: only grab VIP after old master gave it away.
- Constrained to physical boundaries. DC/Zone bound.

Master discovery via VIP

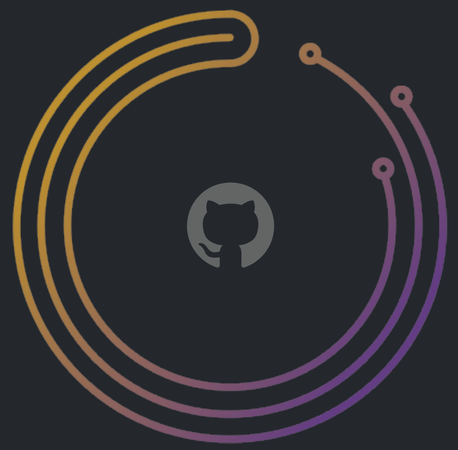


Master discovery via VIP



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"PostMasterFailoverProcesses": [
  "ssh {failedHost} 'sudo ifconfig the-vip-interface down'",
  "ssh {successorHost} 'sudo ifconfig the-vip-interface up'",
  "/do/what/you/gotta/do to apply dns change for
{failureClusterAlias}-writer.example.net to {successorHost}"
],
```

Master discovery via VIP+DNS



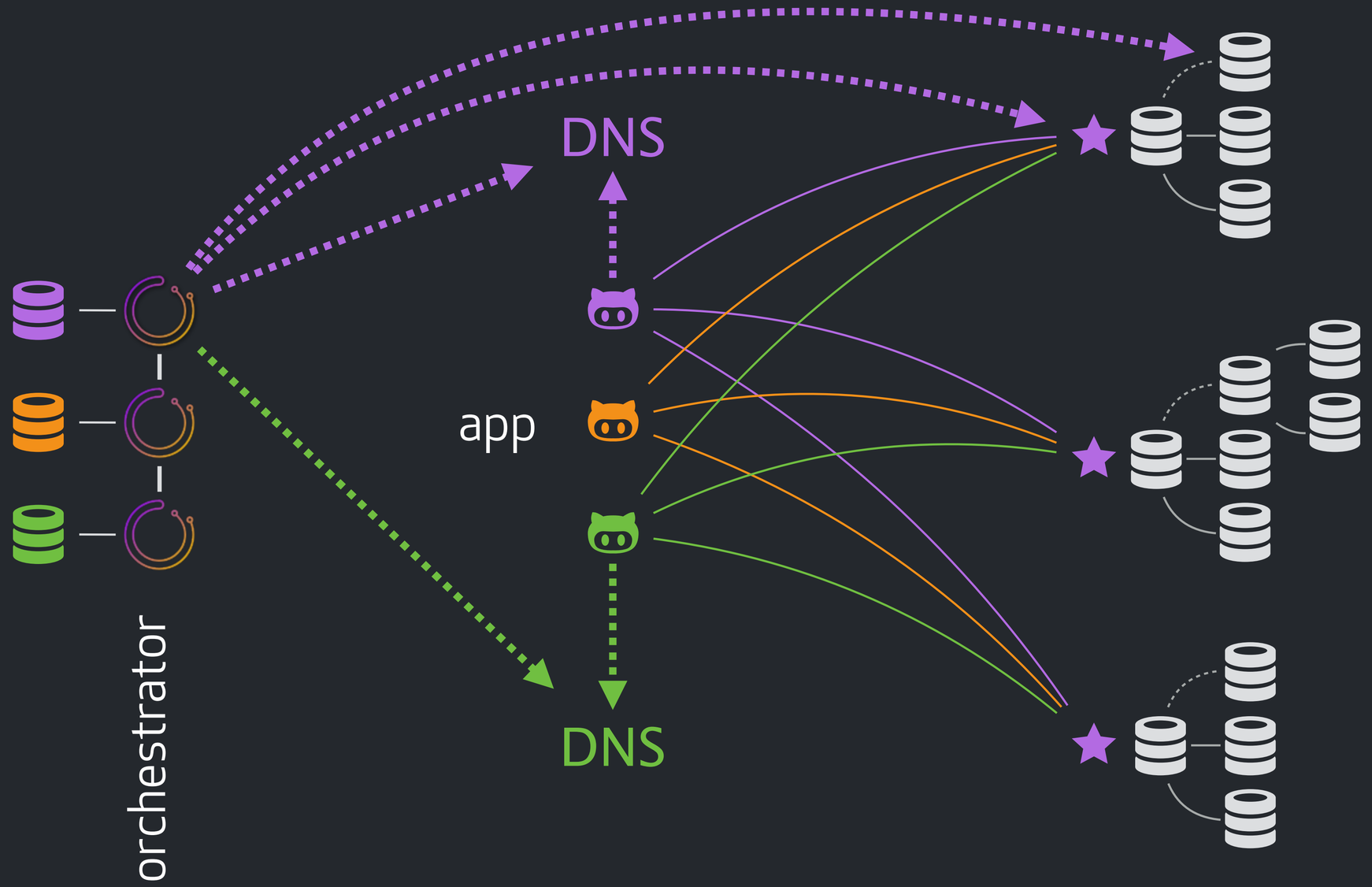
Pros:

- Fast on inter DC/Zone

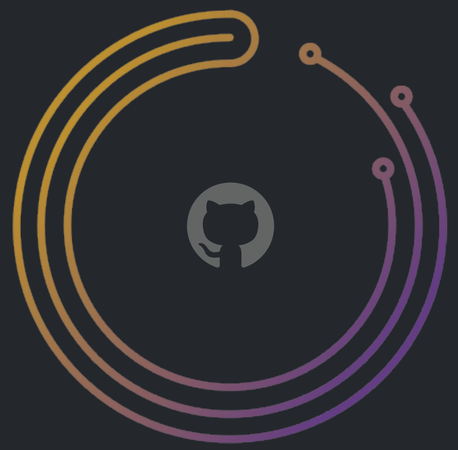
Cons:

- TTL on cross DC/Zone
- Shipping the change to all DNS servers
- Connections to old master potentially uninterrupted
- Slightly more complex logic

Master discovery via VIP+DNS



Master discovery
via service discovery, client based



e.g. ZooKeeper is source of truth, all clients poll/listen on Zk

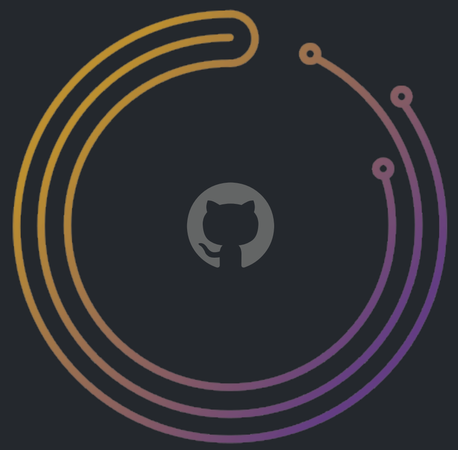
Cons:

- Distribute the change cross DC
- Responsibility of clients to disconnect from old master
- Client overload
- How to verify all clients are up-to-date

Pros: (continued)

Master discovery
via service discovery, client based

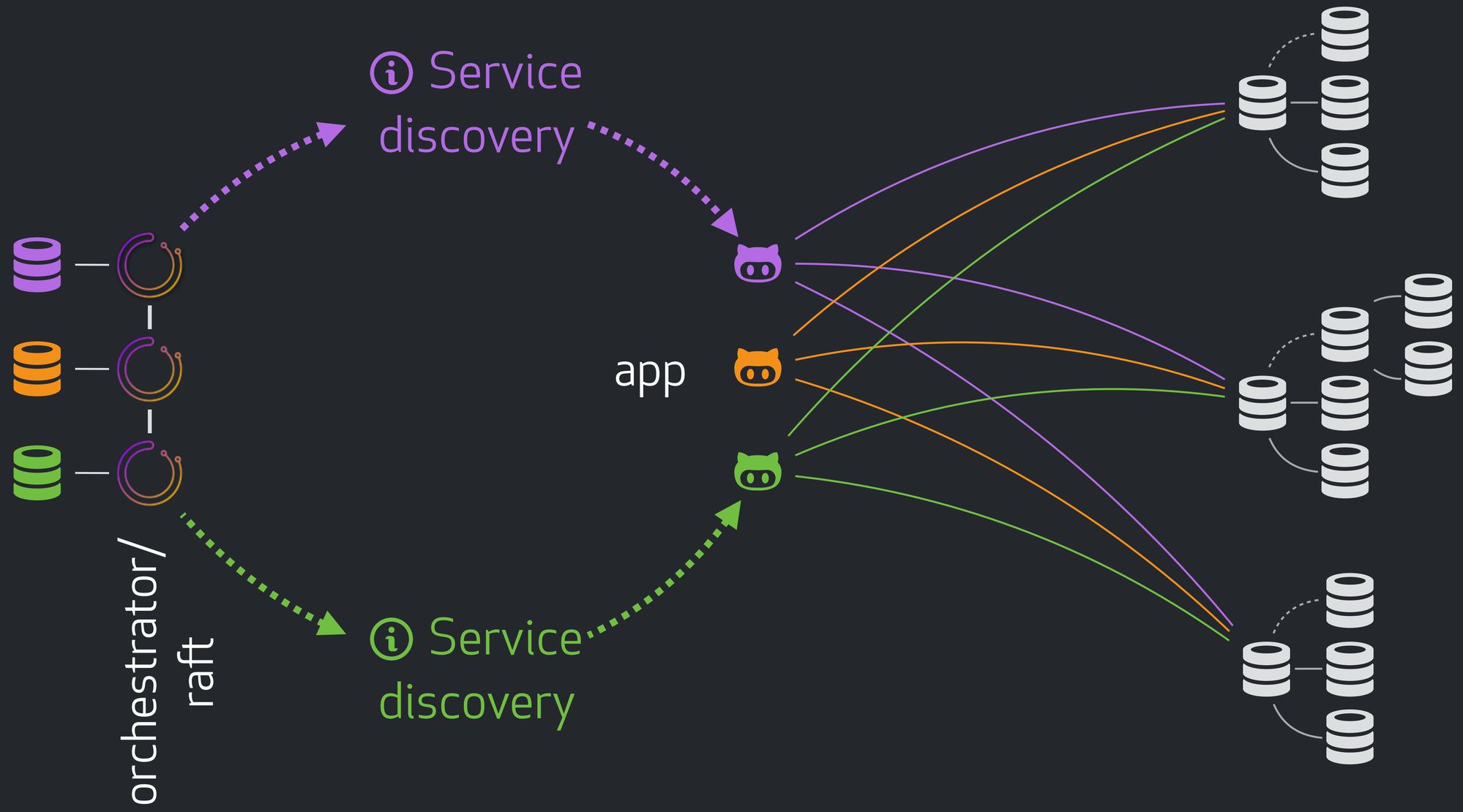
e.g. ZooKeeper is source of truth, all clients poll/listen on Zk



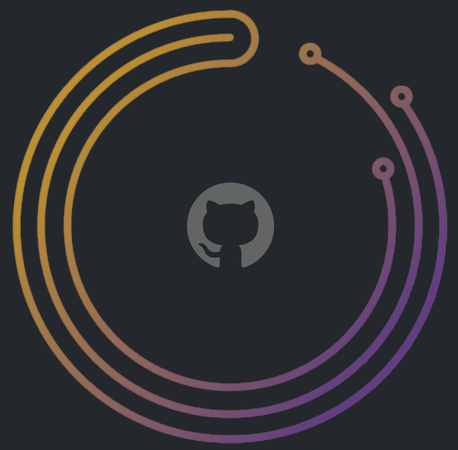
Pros:

- No geographical constraints
- Reliable components

Master discovery via service discovery, client based



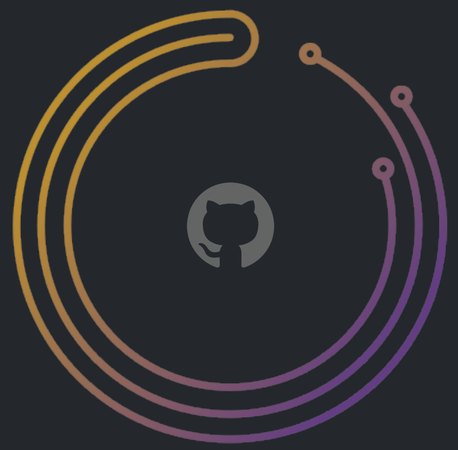
Master discovery via service discovery, client based



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"PostMasterFailoverProcesses": [
  “/just/let/me/know about failover on {failureCluster}“,
],
"KVClusterMasterPrefix": "mysql/master",
"ConsulAddress": "127.0.0.1:8500",
"ZkAddress": "srv-a,srv-b:12181,srv-c",
```

ZooKeeper not implemented yet (v3.0.10)

Master discovery via service discovery, client based

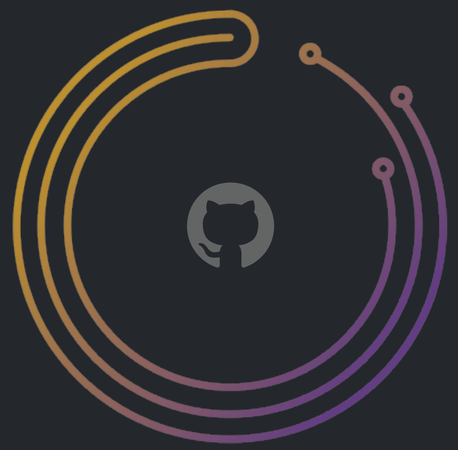


```
"RaftEnabled": true,
"RaftDataDir": "/var/lib/orchestrator",
"RaftBind": "node-full-hostname-2.here.com",
"DefaultRaftPort": 10008,
"RaftNodes": [
  "node-full-hostname-1.here.com",
  "node-full-hostname-2.here.com",
  "node-full-hostname-3.here.com"
],
```

Cross-DC local KV store updates via **raft**

ZooKeeper not implemented yet (v3.0.10)

Master discovery via proxy heuristic



Proxy to pick writer based on `read_only = 0`

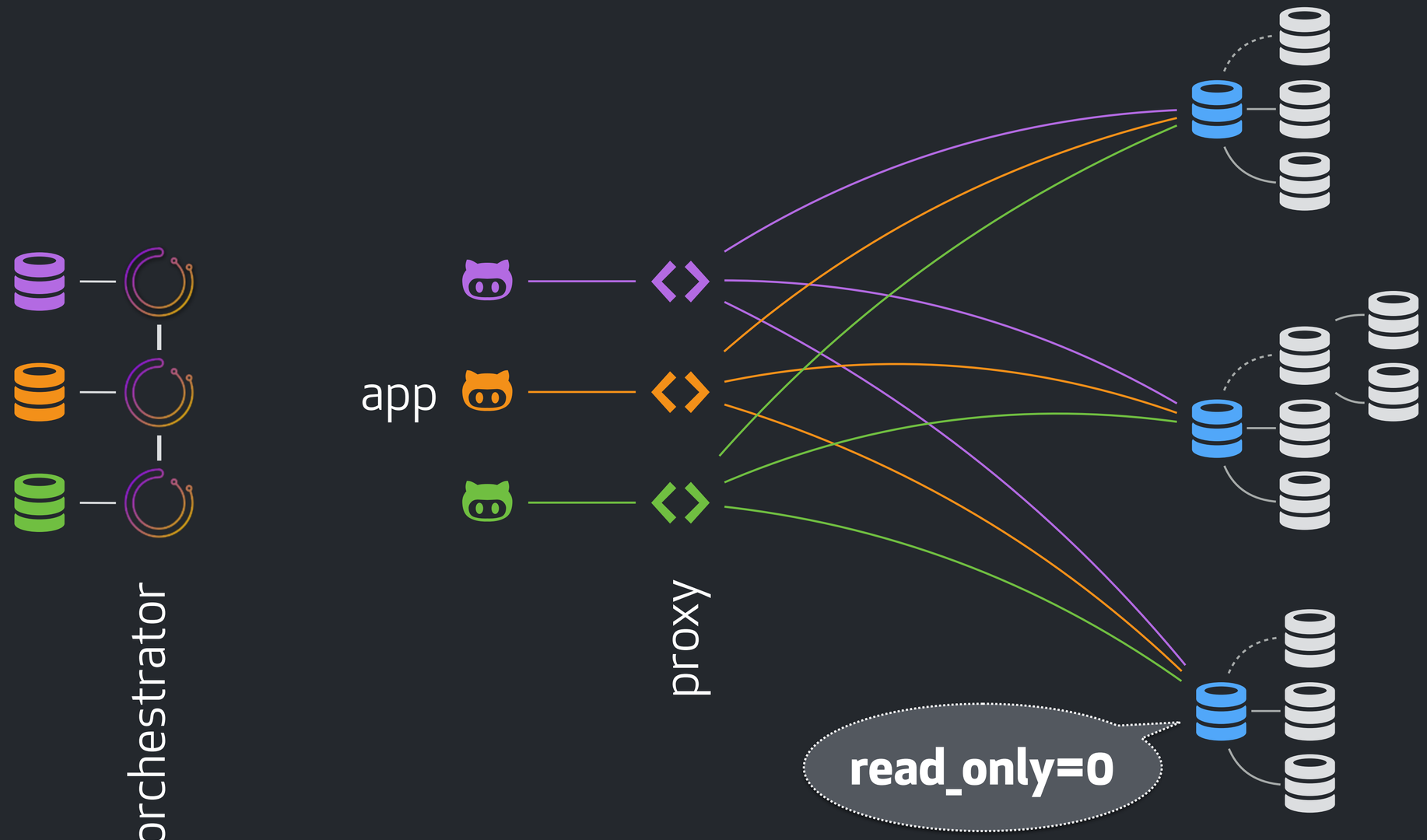
Cons:

An Anti-pattern. **Do not use this method**. Reasonable risk for split brain, two active masters.

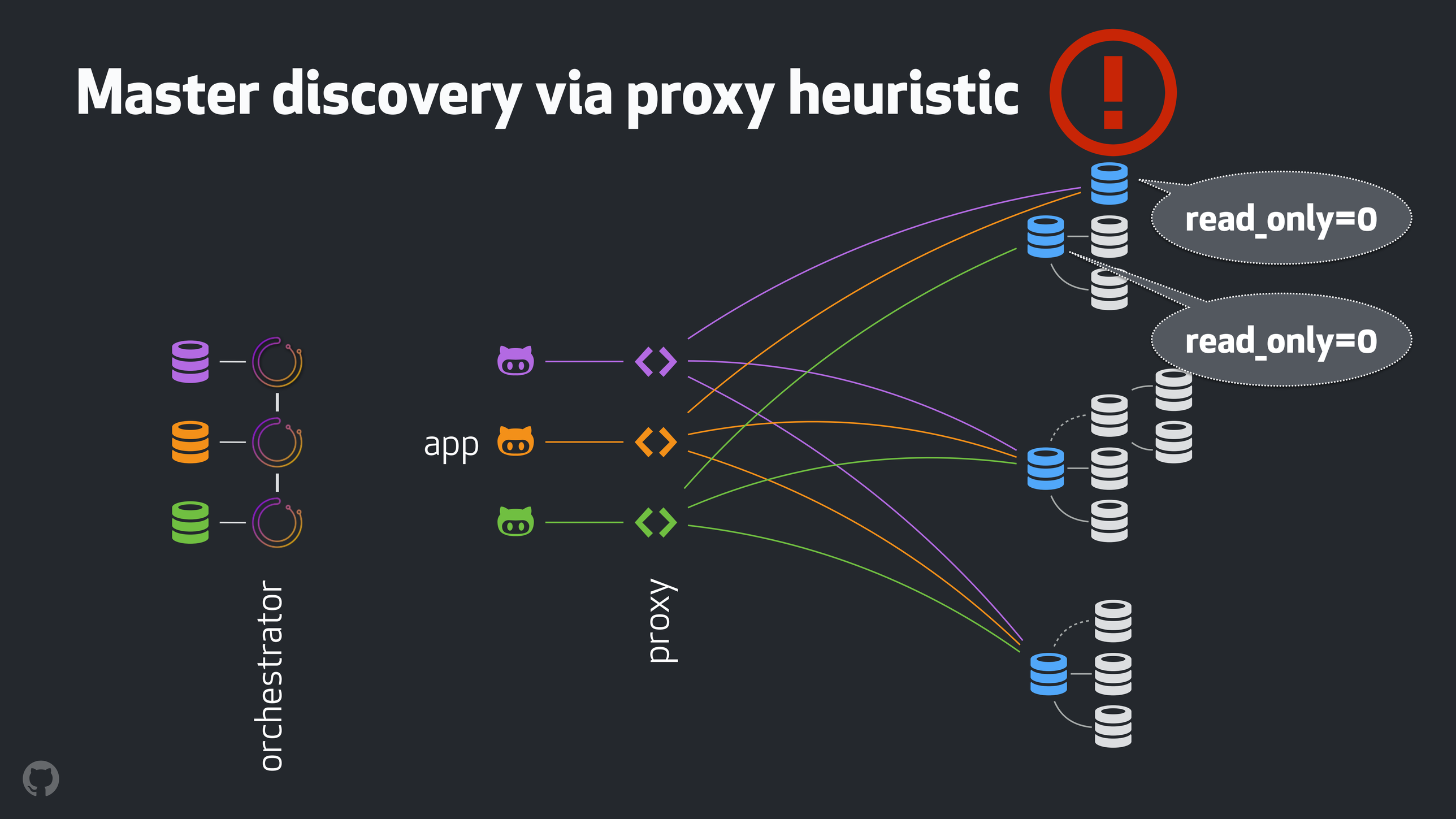
Pros:

Very simple to set up, hence its appeal.

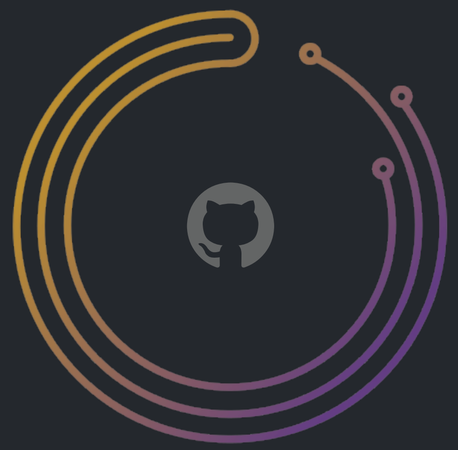
Master discovery via proxy heuristic



Master discovery via proxy heuristic



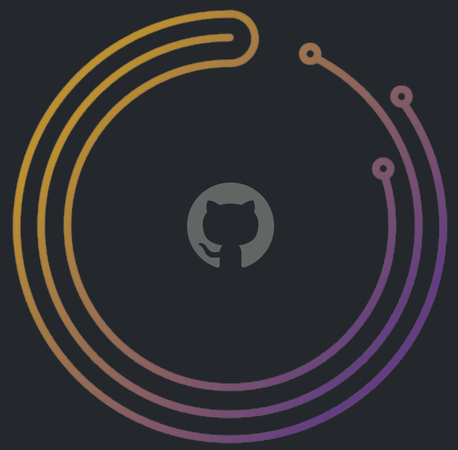
Master discovery via proxy heuristic



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"PostMasterFailoverProcesses": [
  “/just/let/me/know about failover on {failureCluster}“,
],
```

An Anti-pattern. **Do not use this method**. Reasonable risk for split brain, two active masters.

Master discovery via service discovery & proxy



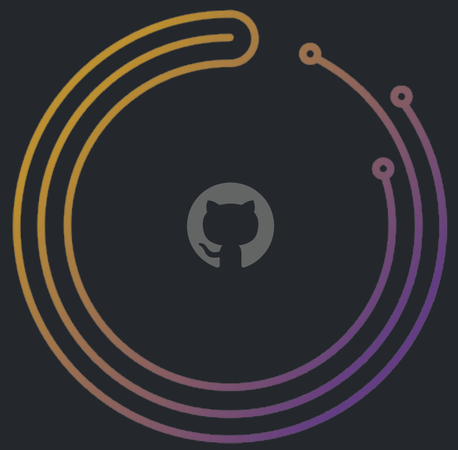
e.g. Consul authoritative on current master identity, consul-template runs on proxy, updates proxy config based on Consul data

Cons:

- Distribute changes cross DC
- Proxy HA?

Pros: (continued)

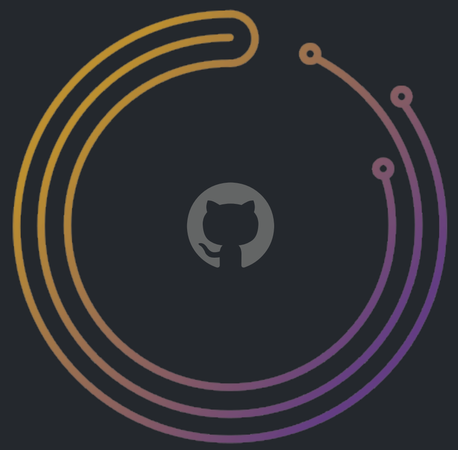
Master discovery via service discovery & proxy



Pros:

- No geographical constraints
- Decoupling failvoer logic from master discovery logic
- Well known, highly available components
- No changes to the app
- Can hard-kill connections to old master

Master discovery via service discovery & proxy



Used at GitHub

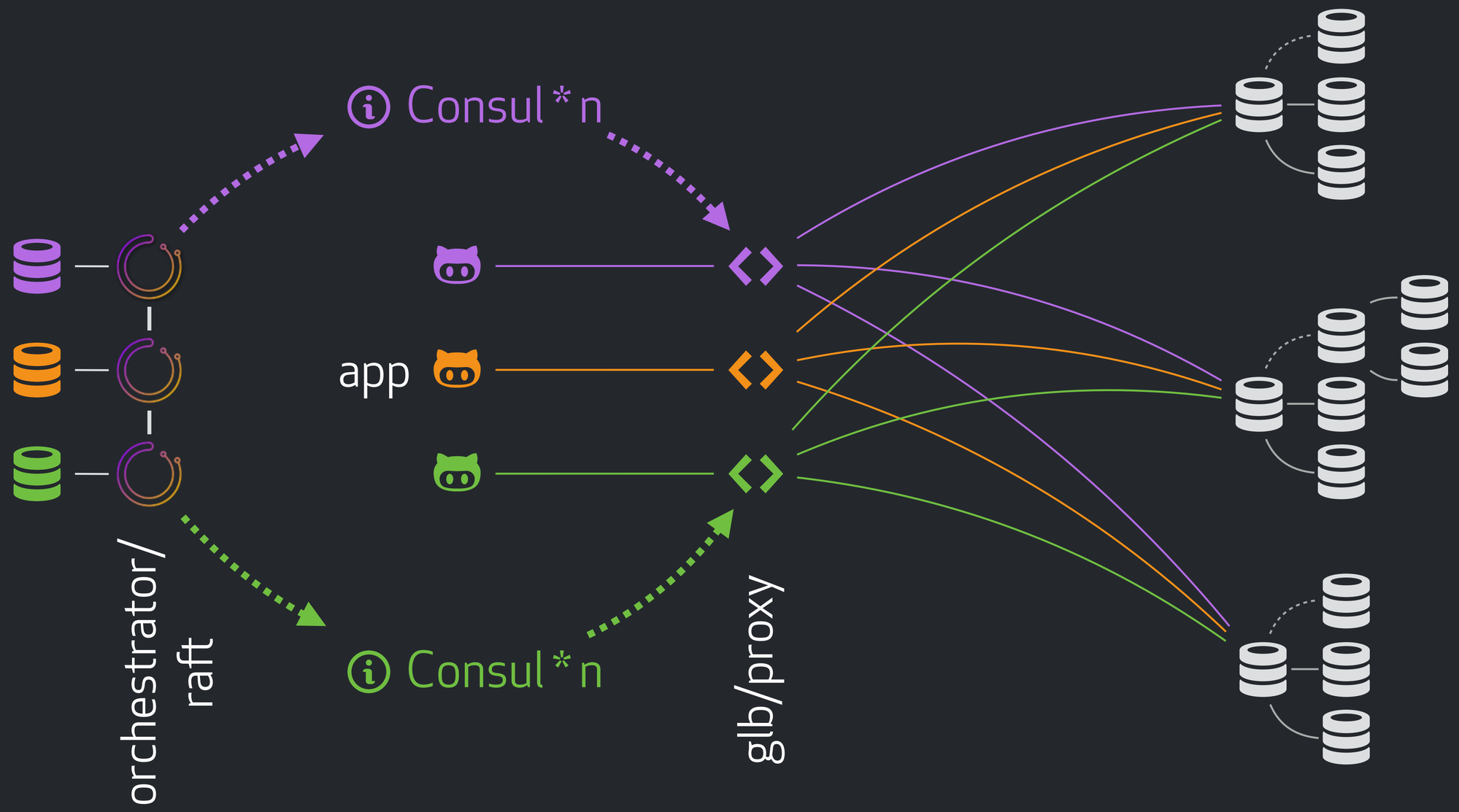
orchestrator fails over, updates **Consul**

orchestrator/raft deployed on all DCs. Upon failover, each orchestrator/raft node updates local Consul setup.

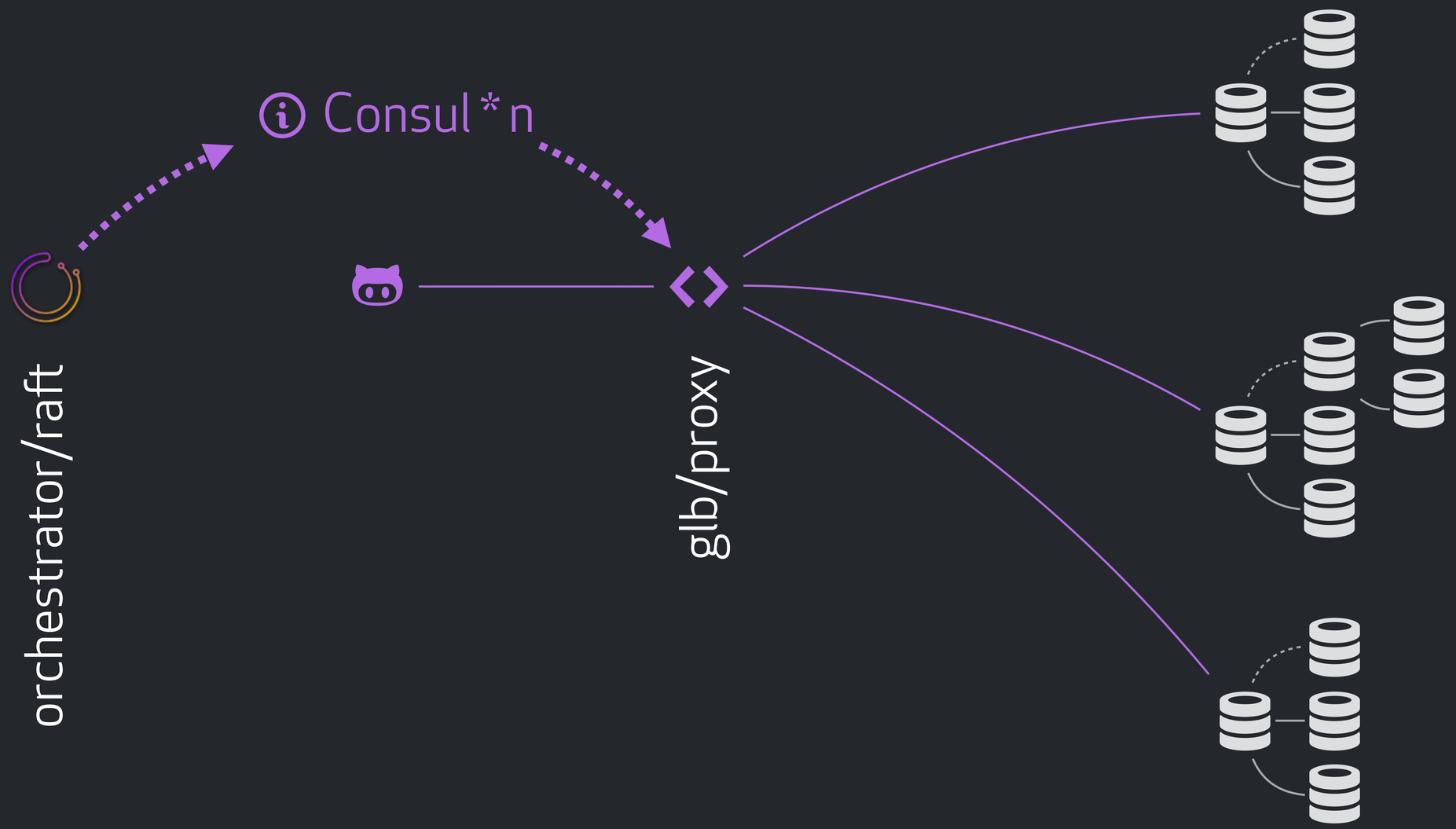
consul-template runs on **GLB** (redundant HAProxy array), reconfigured + reloads GLB upon master identity change

App connects to **GLB**/Haproxy, gets routed to master

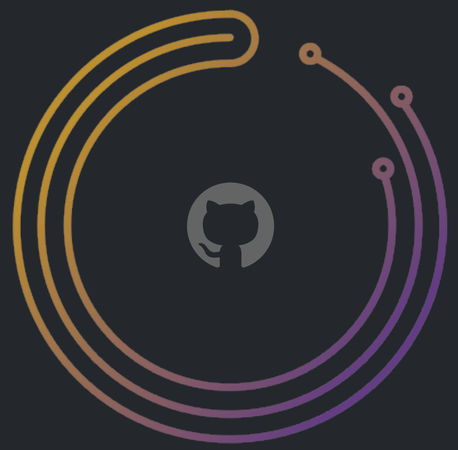
orchestrator/Consul/GLB(HAProxy) @ GitHub



orchestrator/Consul/GLB(HAProxy), simplified



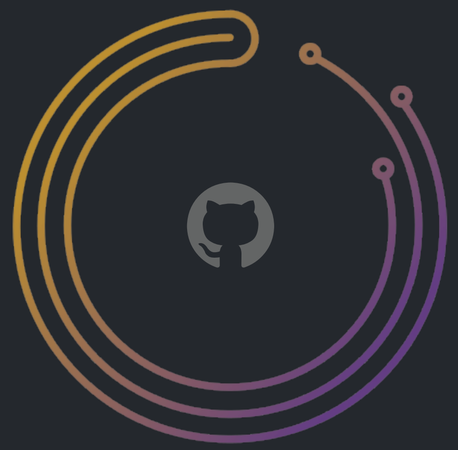
Master discovery via service discovery & proxy



```
"ApplyMySQLPromotionAfterMasterFailover": true,
"PostMasterFailoverProcesses": [
  “/just/let/me/know about failover on {failureCluster}“,
],
"KVClusterMasterPrefix": "mysql/master",
"ConsulAddress": "127.0.0.1:8500",
"ZkAddress": "srv-a,srv-b:12181,srv-c",
```

ZooKeeper not implemented yet (v3.0.10)

Master discovery via service discovery & proxy

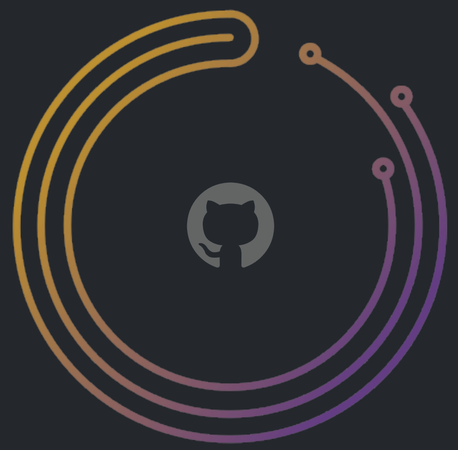


```
"RaftEnabled": true,
"RaftDataDir": "/var/lib/orchestrator",
"RaftBind": "node-full-hostname-2.here.com",
"DefaultRaftPort": 10008,
"RaftNodes": [
  "node-full-hostname-1.here.com",
  "node-full-hostname-2.here.com",
  "node-full-hostname-3.here.com"
],
```

Cross-DC local KV store updates via **raft**

ZooKeeper not implemented yet (v3.0.10)

Master discovery via service discovery & proxy



Vitess' master discovery works in similar manner: **vtgate** servers serve as proxy, consult with backend **etcd/consul/zk** for identity of cluster master.

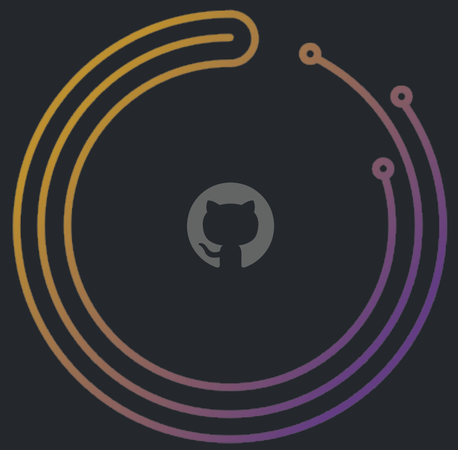
kubernetes works in similar manner. **etcd** lists roster for backend servers.

See also:

Automatic Failovers with Kubernetes using Orchestrator, ProxySQL and Zookeeper
Tue 15:50 - 16:40
Jordan Wheeler, Sami Ahlroos (Shopify)
https://www.percona.com/live/18/sessions/automatic-failovers-with-kubernetes-using-orchestrator-proxysql-and-zookeeper

Orchestrating ProxySQL with Orchestrator and Consul
PerconaLive Dublin
Avraham Apelbaum (wix.COM)
https://www.percona.com/live/e17/sessions/orchestrating-proxysql-with-orchestrator-and-consul

orchestrator HA



What makes orchestrator itself highly available?

orchestrator HA via Raft Concensus

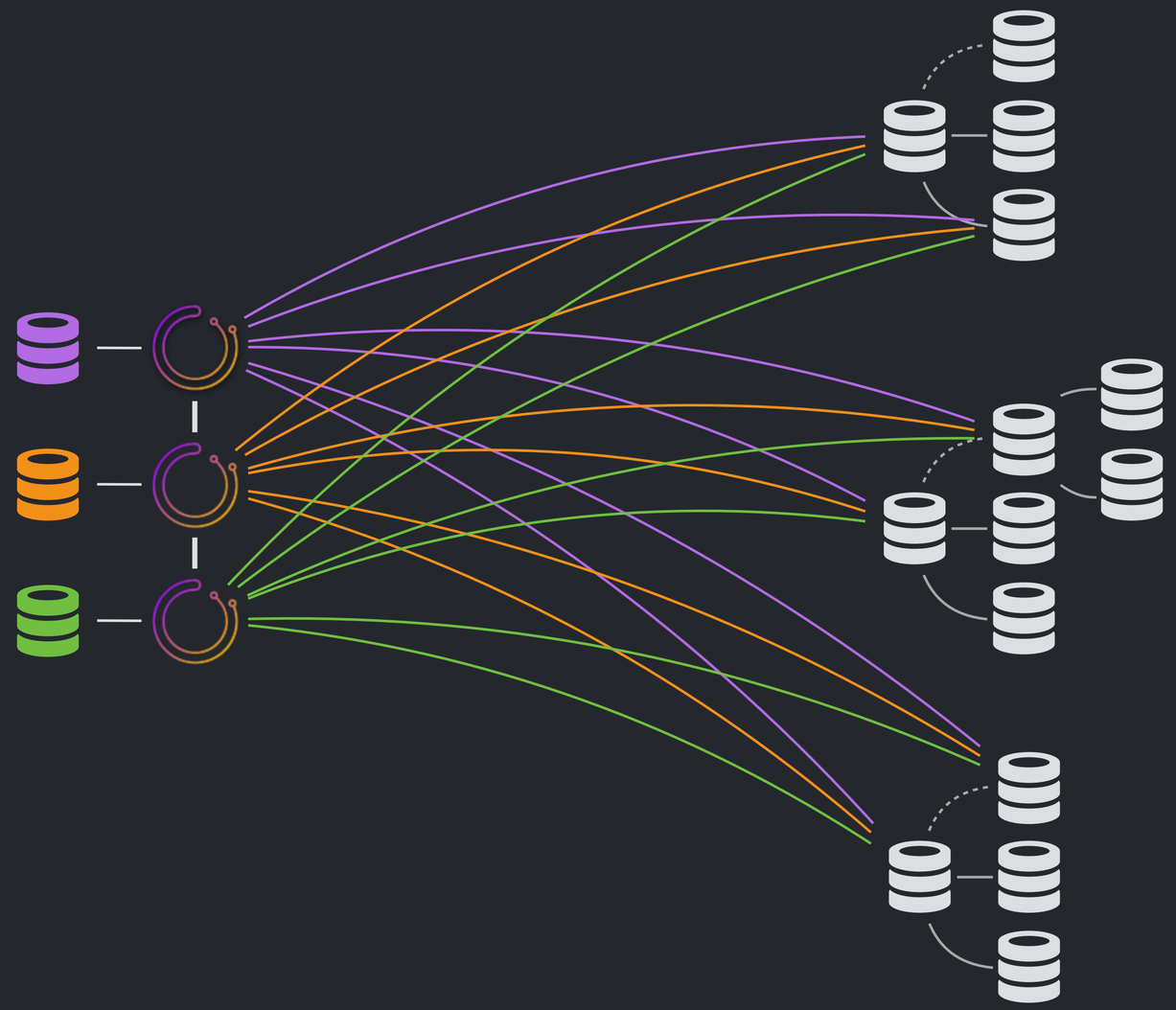
orchestrator/raft for out of the box HA.

orchestrator nodes communicate via raft protocol.

Leader election based on quorum.

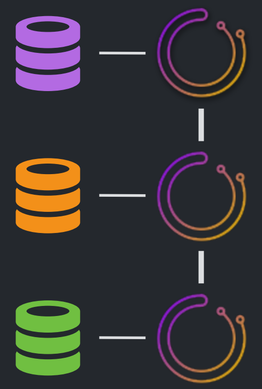
Raft replication log, snapshots. Node can leave, join back, catch up.

https://github.com/github/orchestrator/blob/master/docs/deployment-raft.md



orchestrator HA via Raft Concensus

```
"RaftEnabled": true,
"RaftDataDir": "/var/lib/orchestrator",
"RaftBind": "node-full-hostname-2.here.com",
"DefaultRaftPort": 10008,
"RaftNodes": [
  "node-full-hostname-1.here.com",
  "node-full-hostname-2.here.com",
  "node-full-hostname-3.here.com"
],
```

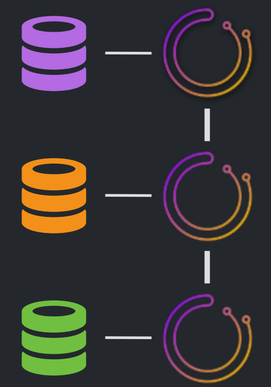


Config docs:

https://github.com/github/orchestrator/blob/master/docs/configuration-raft.md

orchestrator HA via Raft Concensus

```
"RaftAdvertise": “node-external-ip-2.here.com“,


“BackendDB": "sqlite",
"SQLite3DataFile": "/var/lib/orchestrator/orchestrator.db",
```



Config docs:
https://github.com/github/orchestrator/blob/master/docs/configuration-raft.md

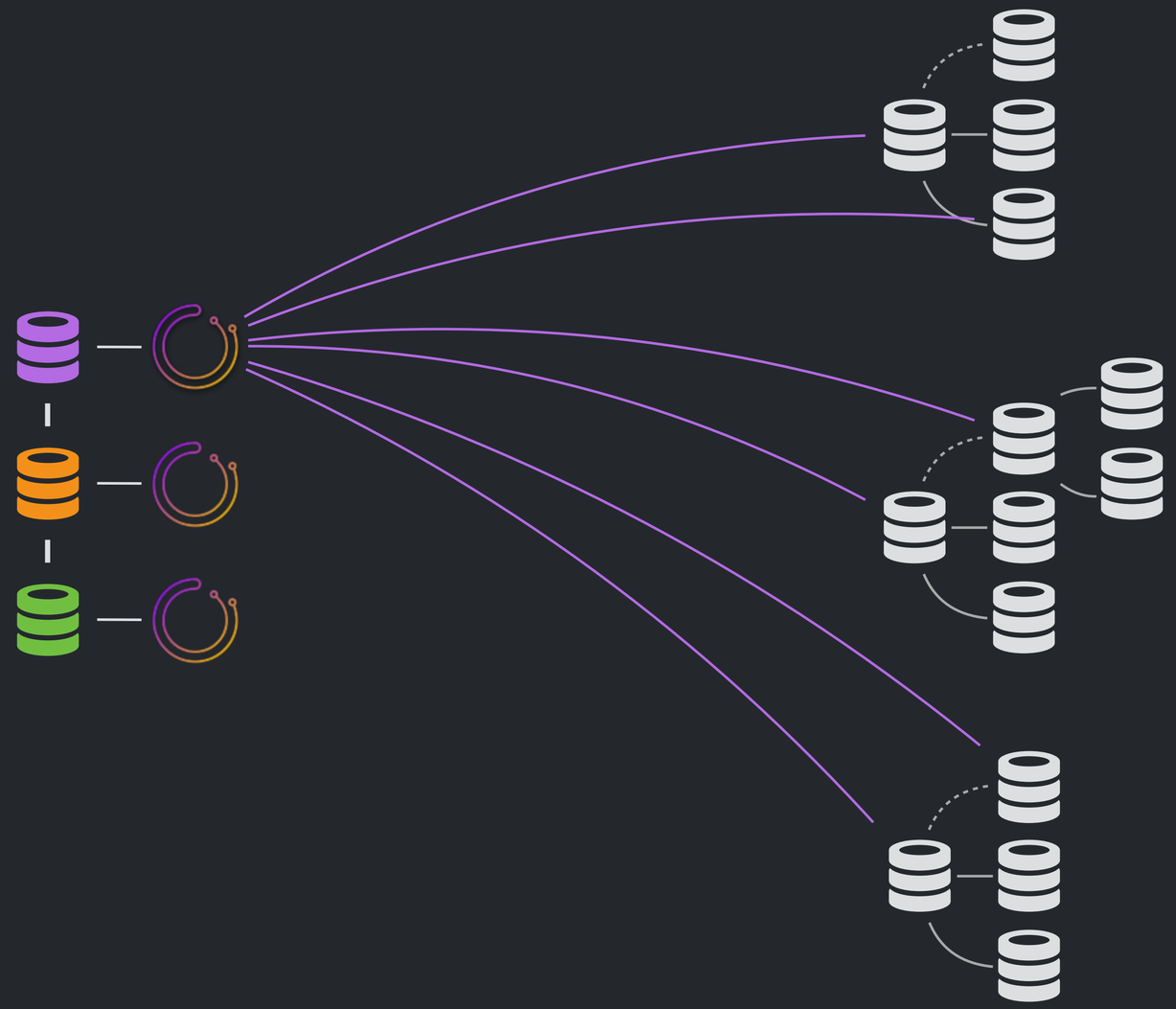
orchestrator HA via shared backend DB

As alternative to **orchestrator/raft**, use Galera/XtraDB Cluster/InnoDB Cluster as shared backend DB.

1:1 mapping between orchestrator nodes and DB nodes.

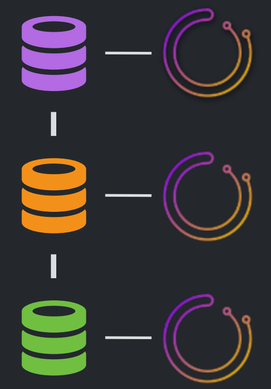
Leader election via relational statements.

https://github.com/github/orchestrator/blob/master/docs/deployment-shared-backend.md



orchestrator HA via shared backend DB

```
"MySQLOrchestratorHost": “127.0.0.1”,
"MySQLOrchestratorPort": 3306,
"MySQLOrchestratorDatabase": "orchestrator",
"MySQLOrchestratorCredentialsConfigFile": “/etc/mysql/
orchestrator-backend.cnf",
```

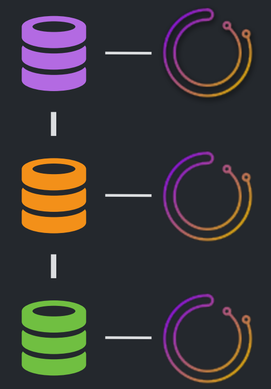


Config docs:

https://github.com/github/orchestrator/blob/master/docs/configuration-backend.md

orchestrator HA via shared backend DB

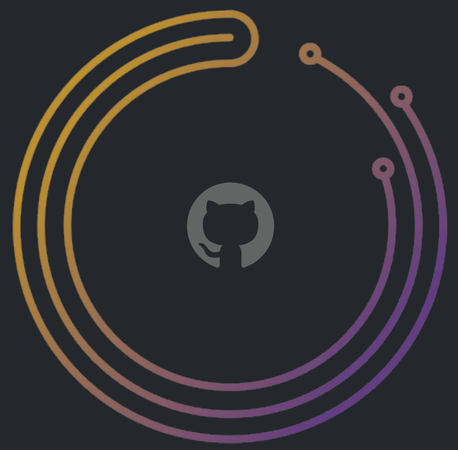
```
$ cat /etc/mysql/orchestrator-backend.cnf
[client]
user=orchestrator_srv
password=${ORCHESTRATOR_PASSWORD}
```



Config docs:

https://github.com/github/orchestrator/blob/master/docs/configuration-backend.md

orchestrator HA approaches



Ongoing investment in **orchestrator/raft**. orchestrator owns its own HA.

Synchronous replication backend owned and operated by the user, not by orchestrator

Comparison of the two approaches:
https://github.com/github/orchestrator/blob/master/docs/raft-vs-sync-repl.md

Other approaches are Master-Master replication or standard replication backend. Owned and operated by the user, not by orchestrator.

Supported

Oracle MySQL, Percona Server, MariaDB

GTID (Oracle + MariaDB)

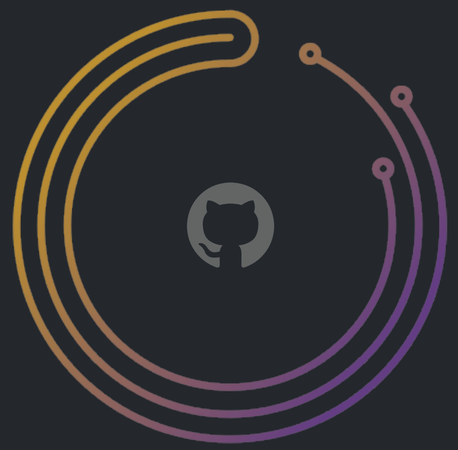
Semi-sync, statement/mixed/row, parallel replication

Master-master (2 node circular) replication

SSL/TLS

Consul, Graphite, MySQL/SQLite backend

Not supported



Galera/XtraDB Cluster

InnoDB Cluster

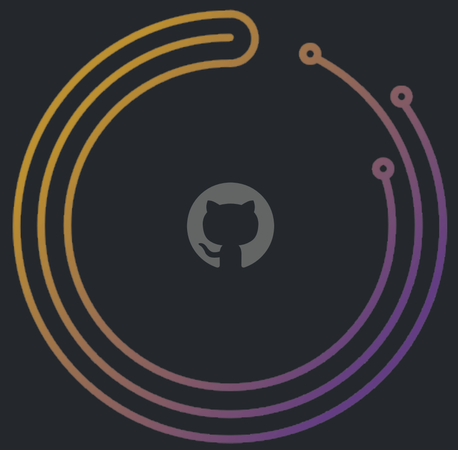
Multi source replication

Tungsten

3+ nodes circular replication

5.6 parallel replication for Pseudo-GTID

Conclusions



orchestrator/raft makes for a good, cross DC highly available self sustained setup, Kubernetes friendly. Consider **sqlite** backend.

Master discovery methods vary. Reduce hooks/friction by using a discovery service.

Thank you!

Questions?

github.com/shlomi-noach
@ShlomiNoach