

a postgres orchestra

Akshay Gupta / kitallis / nilenso.com

gitlab incident



GitLab.com Status

@gitlabstatus

 Follow

We are performing emergency database maintenance,
[GitLab.com](#) will be taken offline

4:58 AM - 1 Feb 2017

  42  33



GitLab.com Status

@gitlabstatus

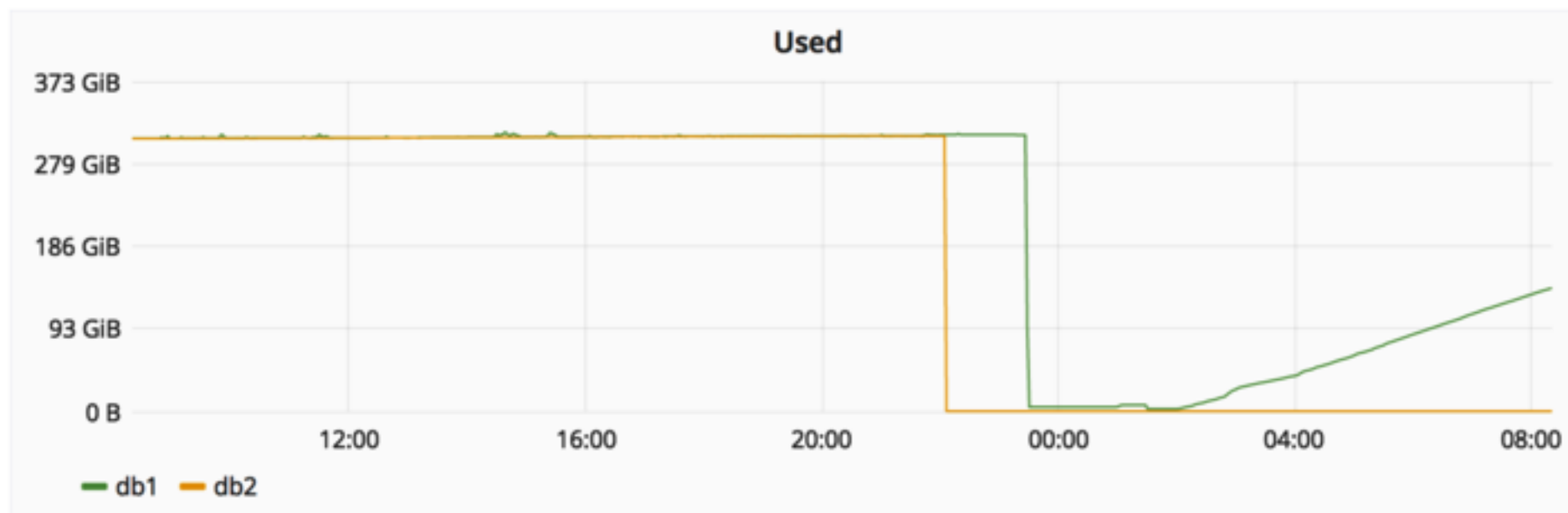
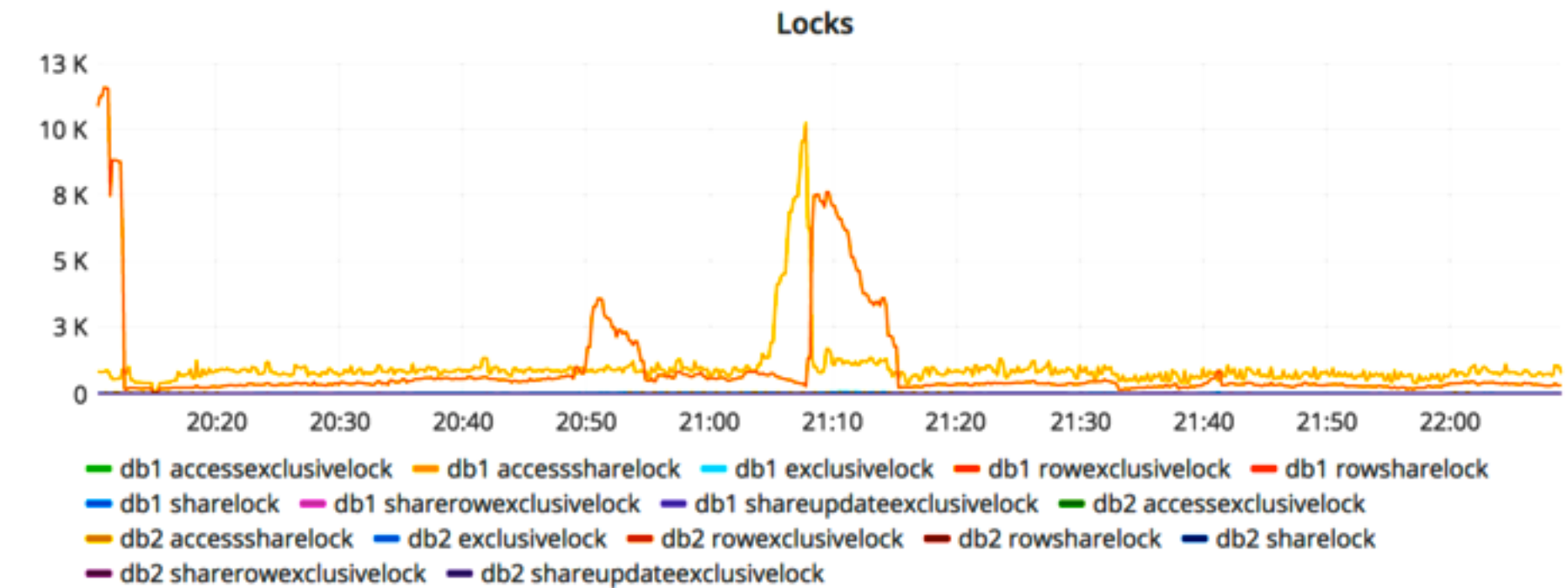
 Follow

We accidentally deleted production data and might have to
restore from backup. Google Doc with live notes
docs.google.com/document/d/1GC...

6:14 AM - 1 Feb 2017

  2,934  2,750

gitlab incident



gitlab incident

Impact

1. ±6 hours of data loss
2. 4613 regular projects, 74 forks, and 350 imports are lost (roughly); 5037 projects in total. Since Git repositories are NOT lost, we can recreate all of the projects whose user/group existed before the data loss, but we cannot restore any of these projects' issues, etc.
3. ±4979 (so ±5000) comments lost
4. 707 users lost potentially, hard to tell for certain from the Kibana logs
5. Webhooks created before Jan 31st 17:20 were restored, those created after this time are lost

So in other words, out of 5 backup/replication techniques deployed none are working reliably or set up in the first place.
=> we're now restoring a backup from 6 hours ago that worked

7. We don't have solid alerting/paging for when backups fails, we are seeing this in the dev host too now.

about

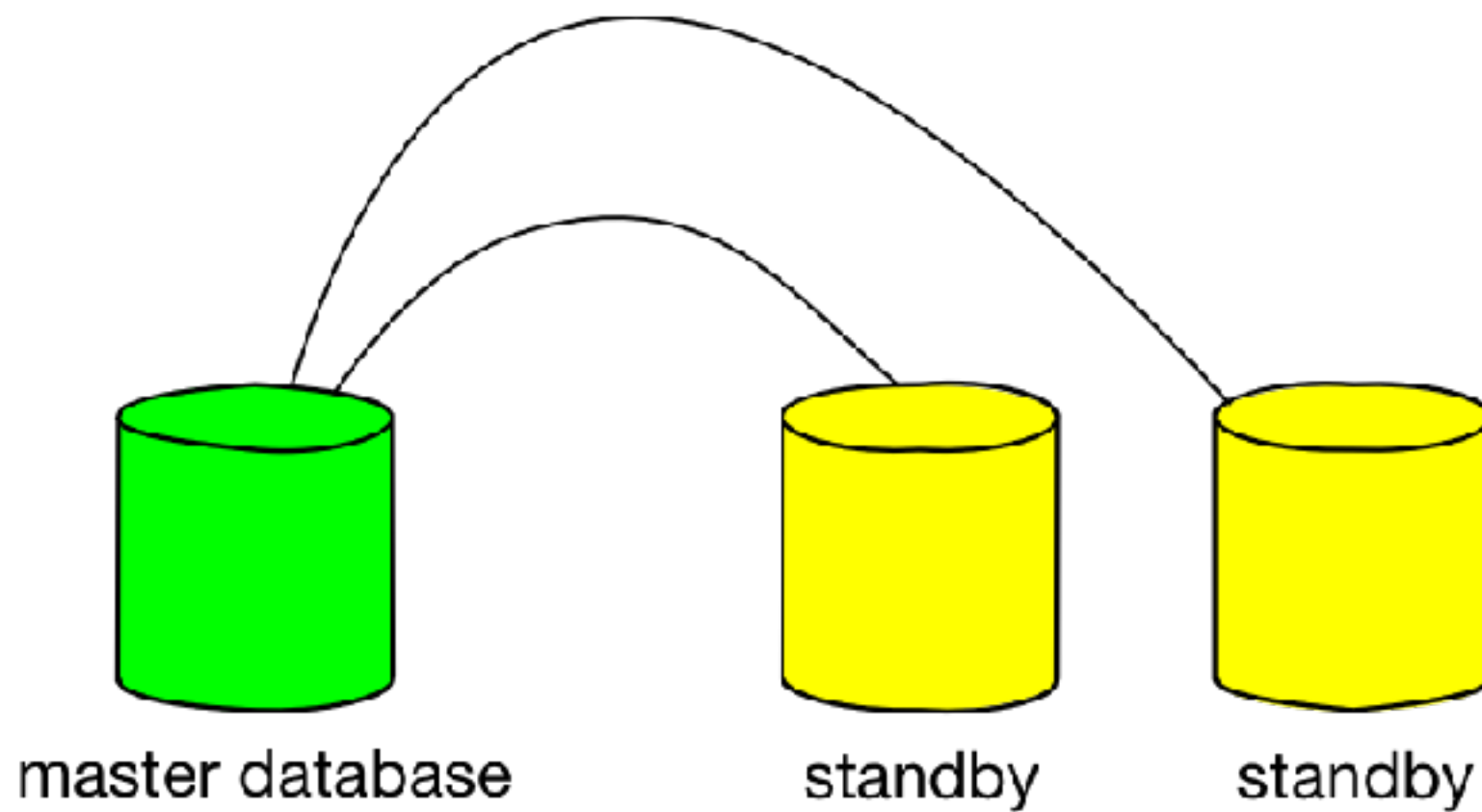
- points of failure in current practices and clustering systems
- design an abstraction on top of existing systems to ensure availability
- illustrate some scenarios

primer

db cluster / compute cluster

- a database cluster is a collection of databases that is managed by a single instance of a running database server
- a group of databases that work together to achieve higher performance and/or availability

one master, multiple standbys



totally available

- 0 single points of failure
- number of failures it can tolerate = infinite

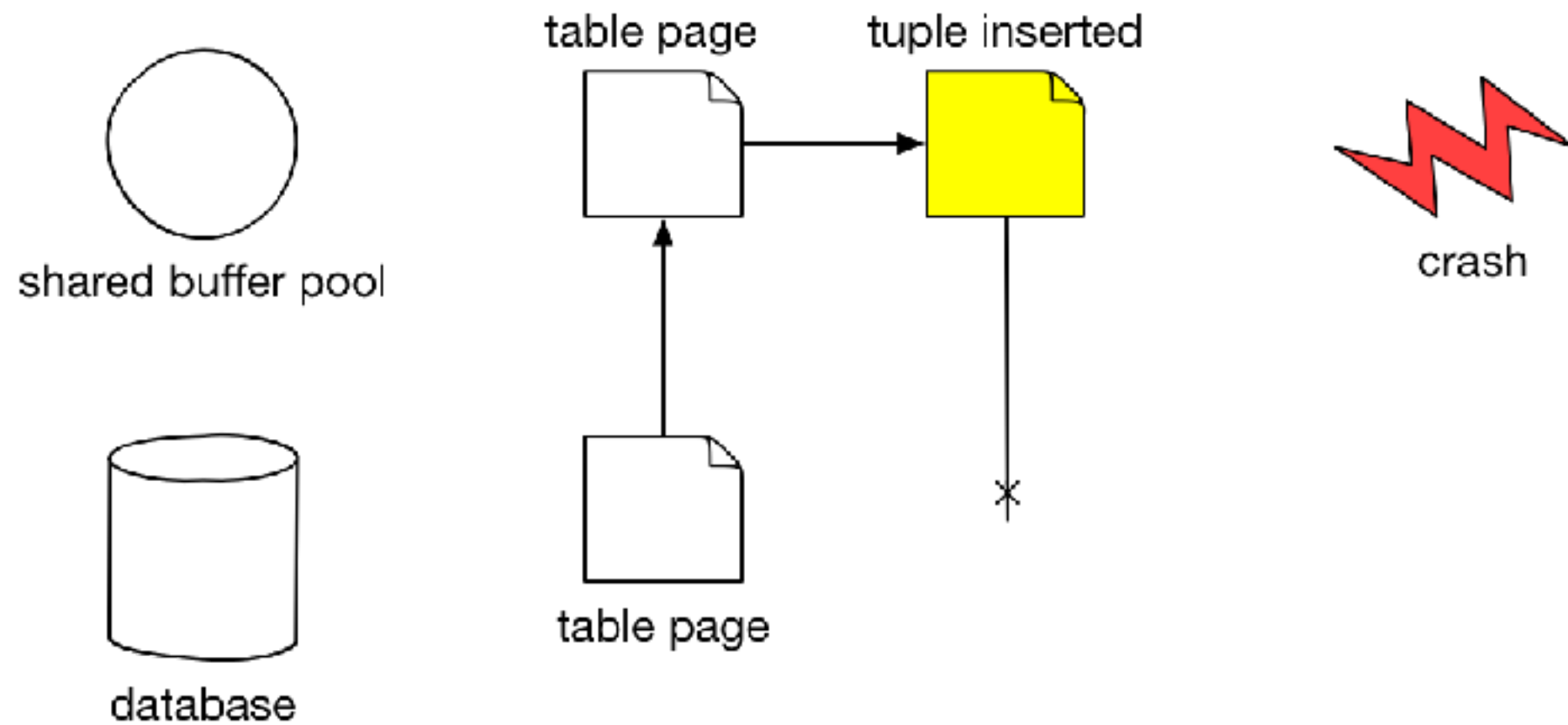
highly available

- tending towards 0 single point of failures
- number of failures it can tolerate = f

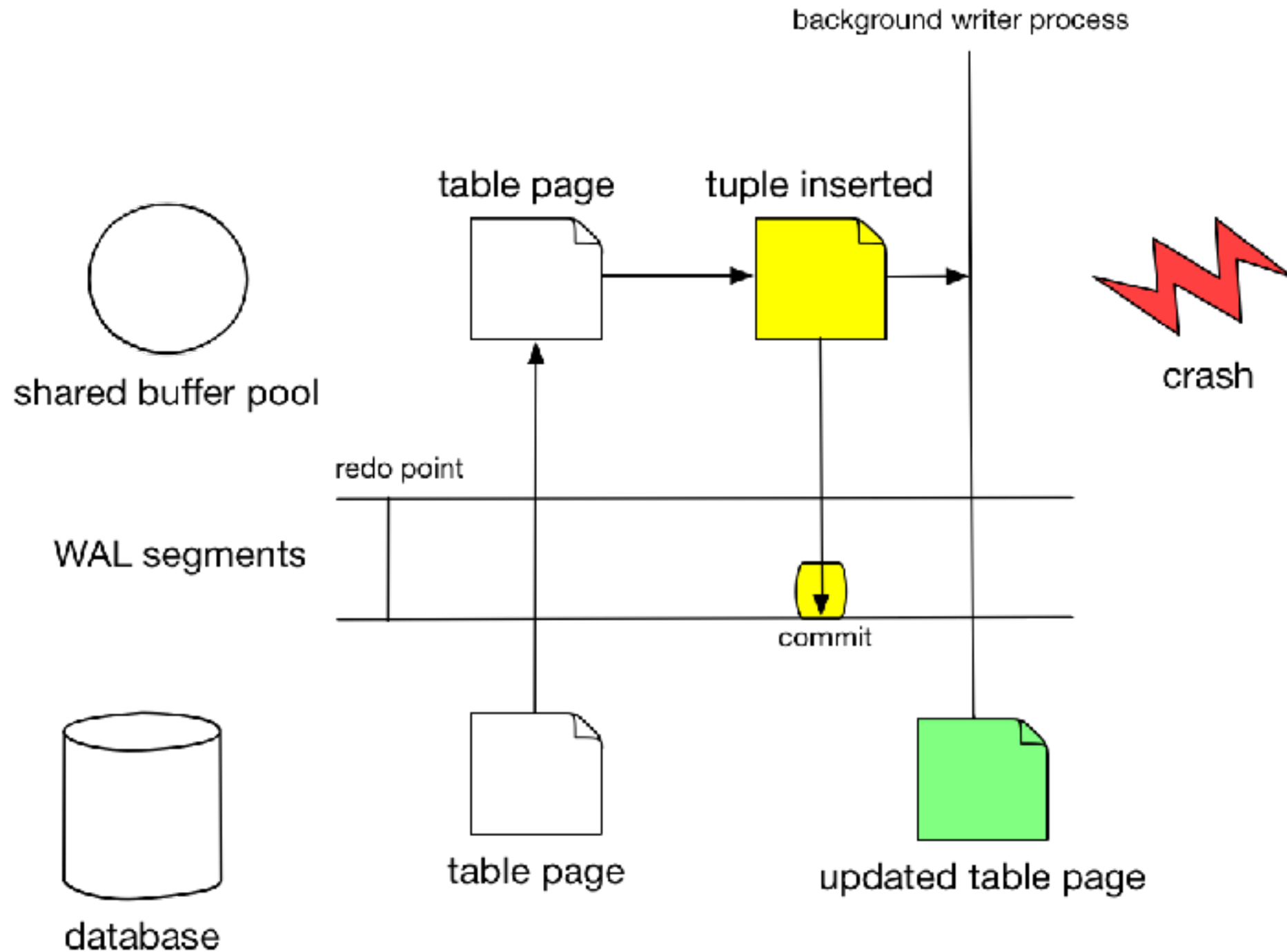
increase the area of success

- write ahead logs
- standbys / replicas

areas of success - WAL

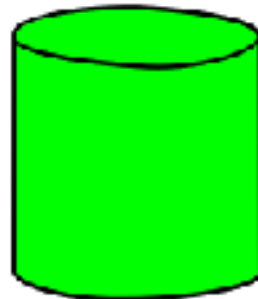


areas of success - WAL



areas of success - replica

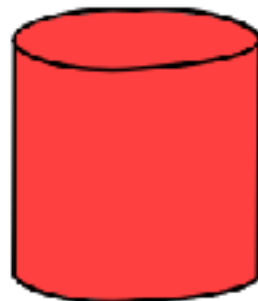
time
↓



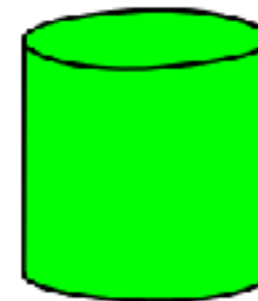
master database



standby




failed



master database

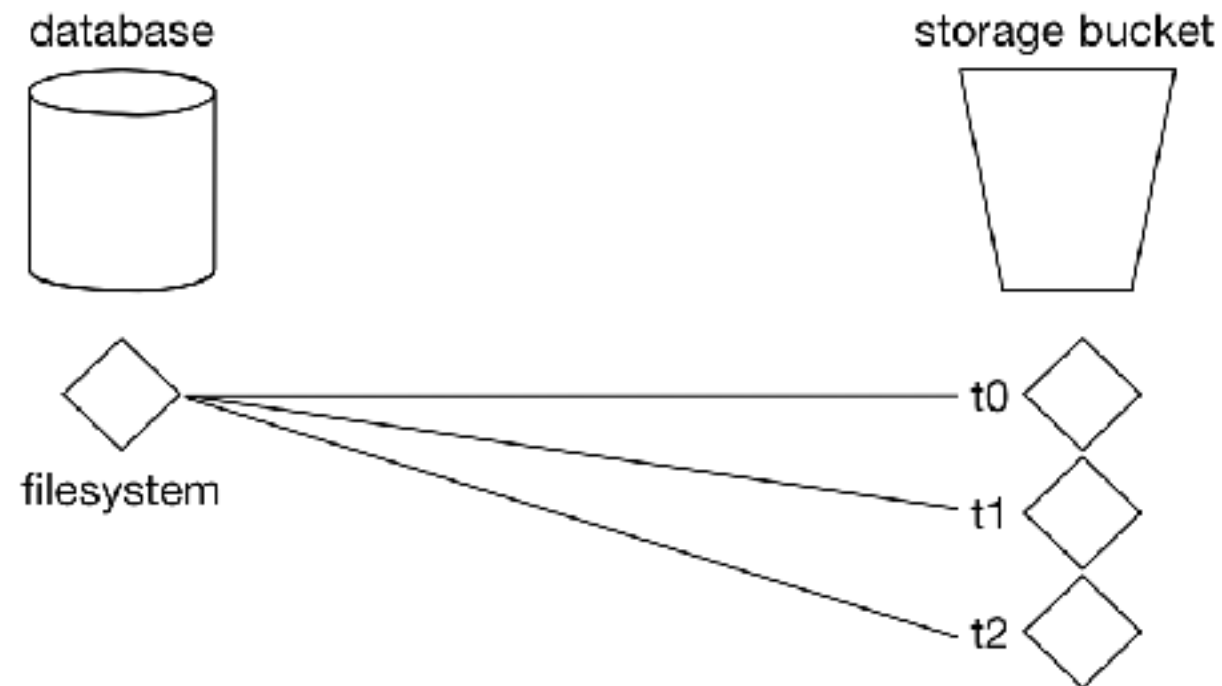
failures

- filesystem gives up at 80% disk usage
- high replication lag
- too many connections / semaphores
- network partitions
- cyclone Vardah 

replication

- file-system replication
- logical backups
- log shipping / streaming

replication - snapshots



```
$ zfs snapshot -r kitallis/home@now  
$ zfs list -t snapshot
```

NAME	USED	AVAIL	REFER	MOUNTPOINT
rpool/zfs	78.3M	-	4.53G	-
kitallis/home@now	0	-	26K	-

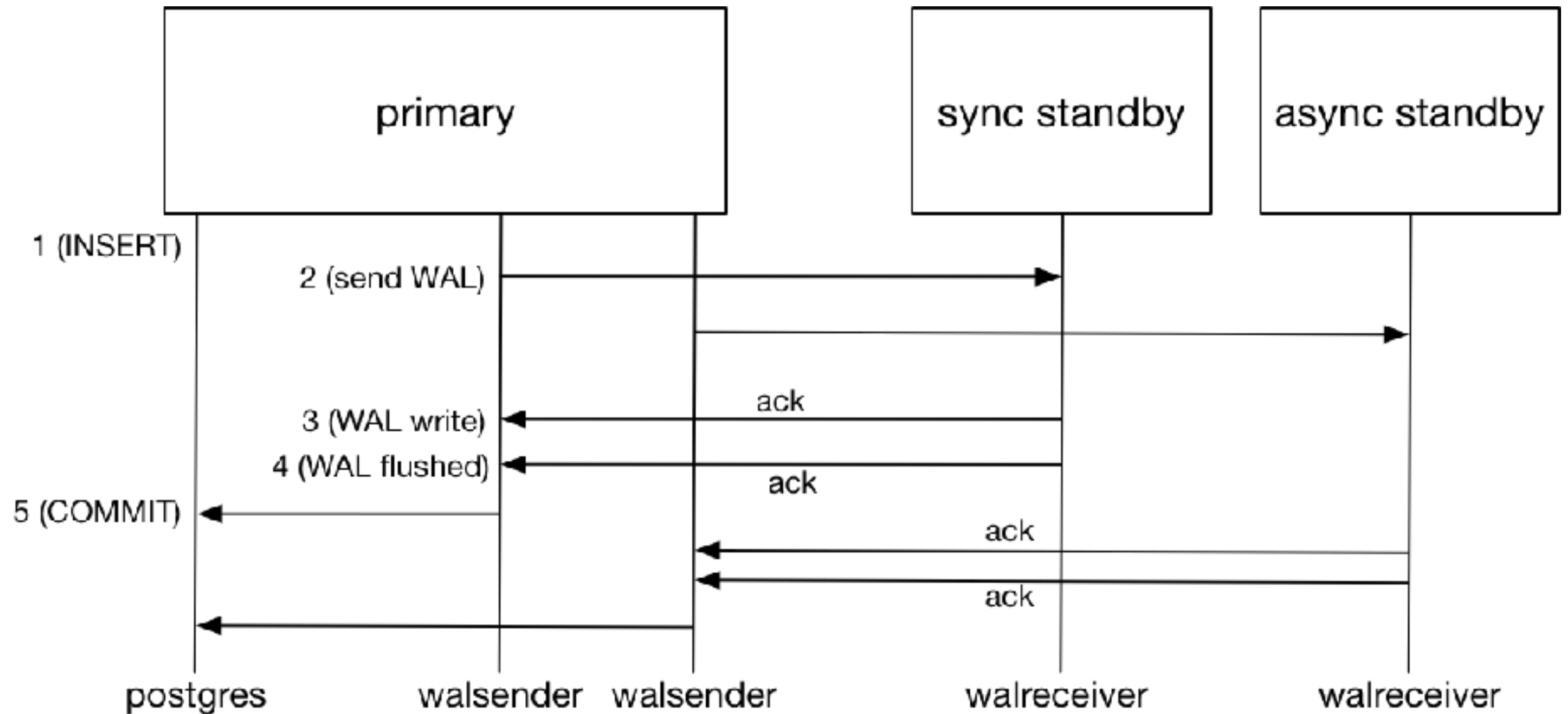
replication – logical

```
> pg_dump db1 > db.sql  
> psql -d db2 -f db.sql
```

```
> pg_dump -Fc db1 > db.dump  
> pg_restore -d db2 db.dump
```

```
> pg_dumpall > db.out  
> psql -f db.out postgres
```

replication - streaming



tools

clustering tools

Clustering

Contents [\[hide\]](#)

- 1 Projects
- 2 Stalled
- 3 Existing Overview Docs
- 4 Template for information

Projects

(alphabetical order)

- Bucardo
- BDR (Bi-Directional Replication for PostgreSQL)
- Citus 
- GridSQL
- HadoopDB
- pg_shard 
- Pgpool-II
- PL/Proxy
- Postgres-XC
- Postgres-XL , a Postgres-XC derivative/enhancement.
- PostgresForest
- SkyTools (Londiste)
- Slony
- Sludo
- Tungsten

Work in progress is being tracked at [Clustering Development Projects](#)

Stalled

Projects which seem to have stalled and have not had updates in 1+ year.

- Mammoth
- PgCluster
- Postgres-R
- rubyrep

clustering + orchestration

- we require more than just a nice API on top of replication
- govern the cluster as well

properties

- service is inherently a singleton
- high availability
- fail-overs are automatic
- consumers should be able to refresh connections
- zero / minimal application state

suggestions



pgDay Asia 2016

March 17th and 19th
Singapore

network-oriented

- keepalived
- UCARP

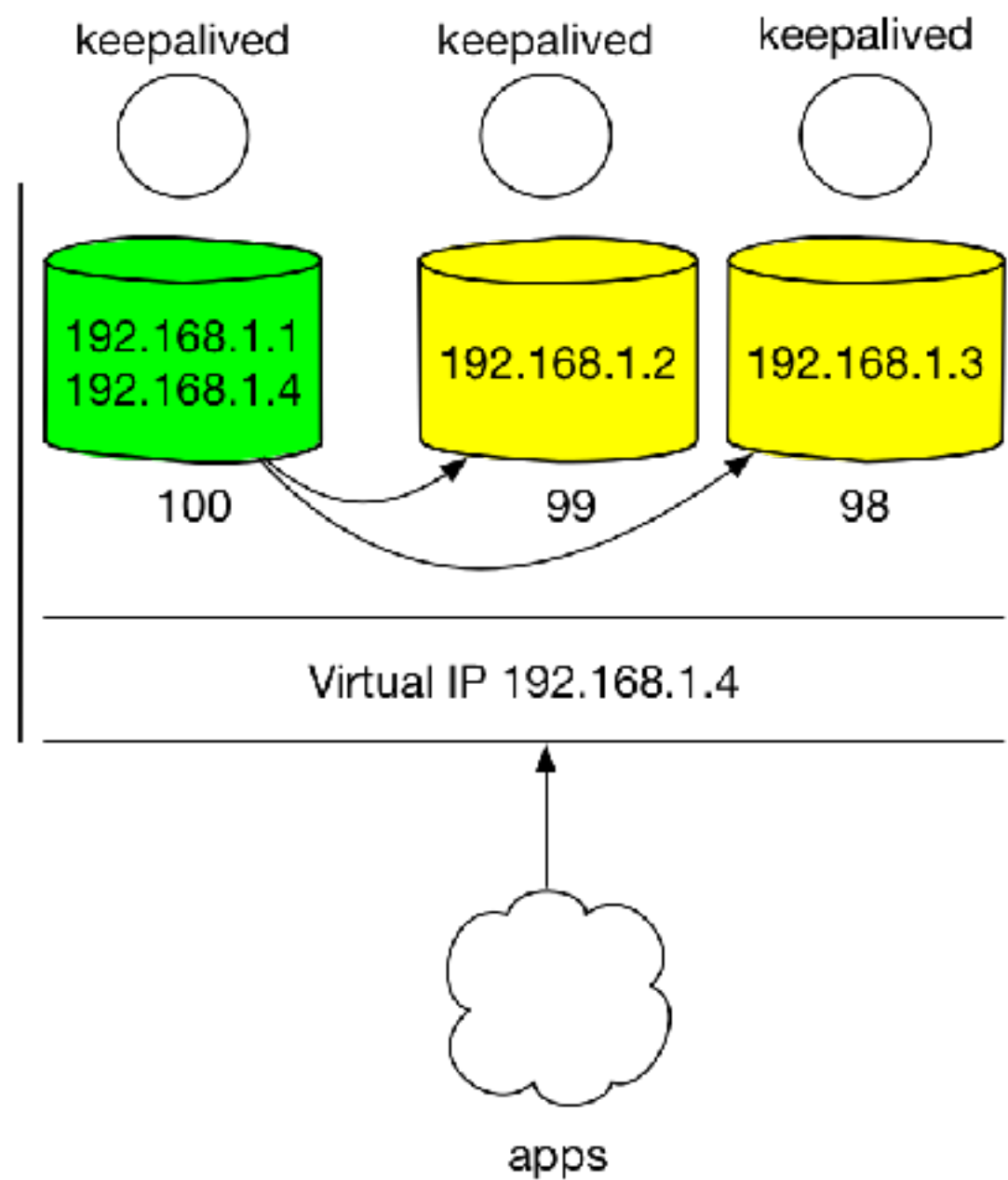
keepalived / UCARP

- built on the VRRP protocol
- provides health check for up/down service
- UCARP and keepalived are pretty similar

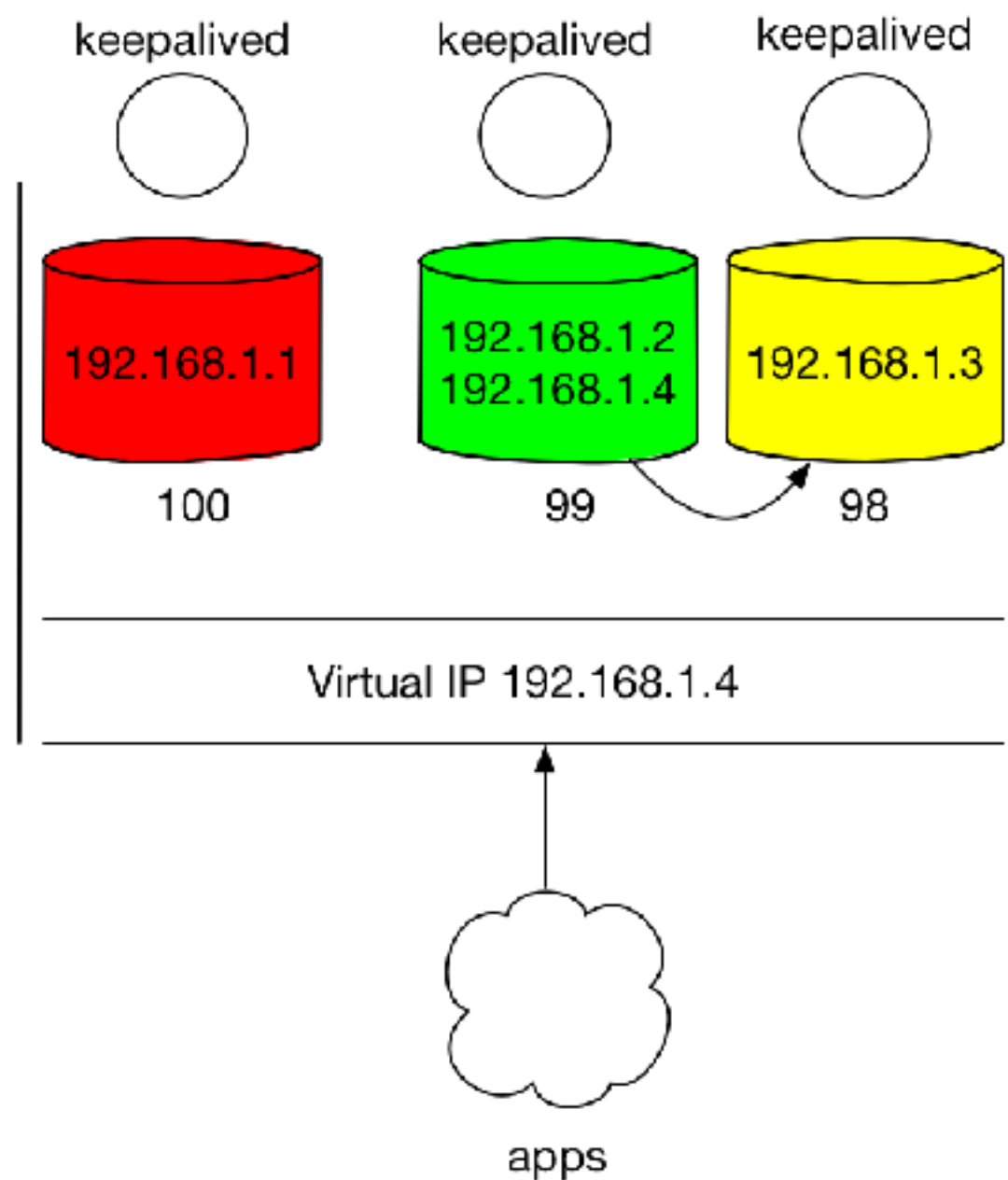
keepalived

- health check (try INSERTs or SELECTs)
- notify_master (promote standby)
- notify_standby (follow new master)

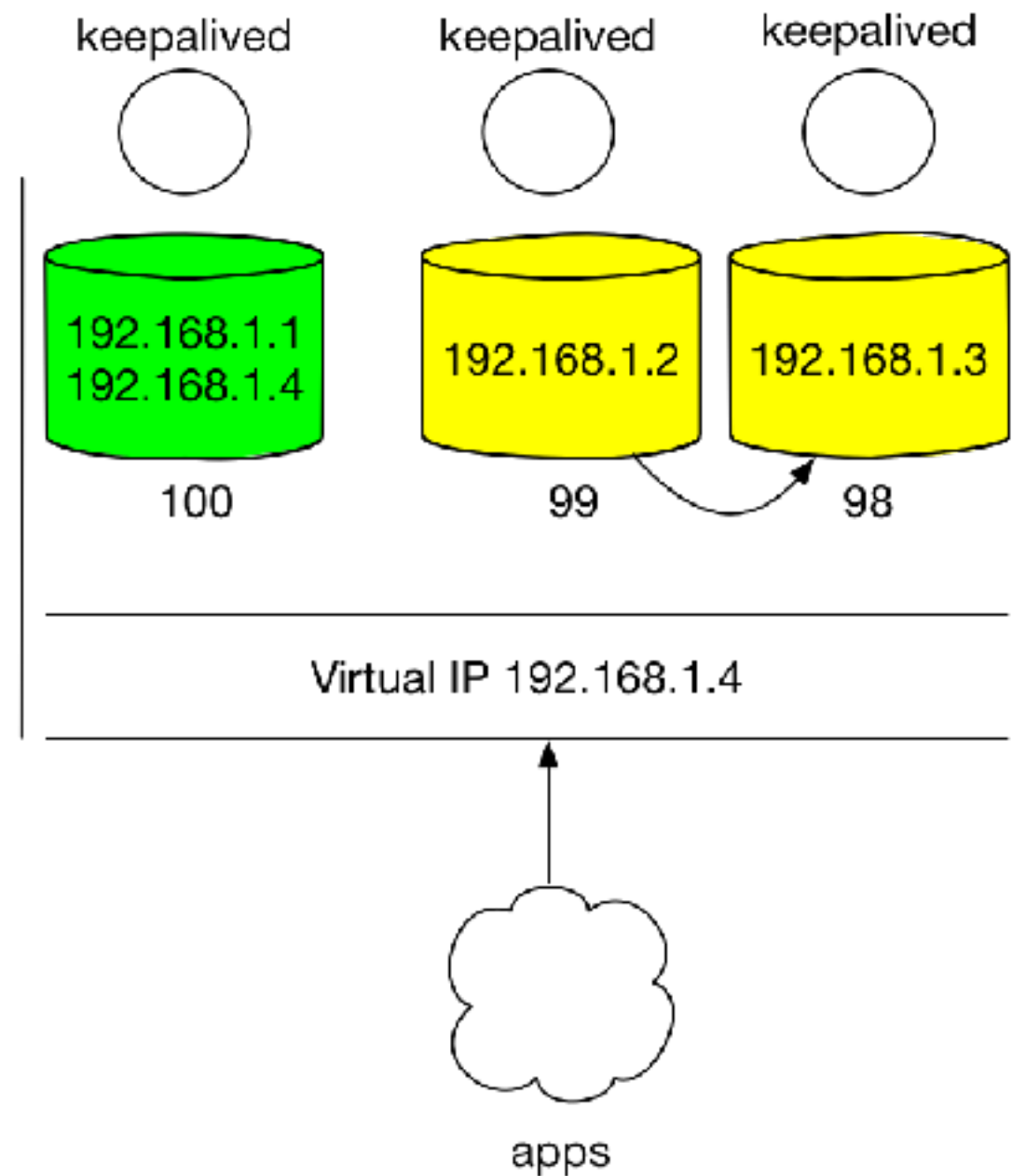
keepalived



keepalived



keepalived (problem)



keepalived / UCARP-
nopreempt

```
nopreempt  
state BACKUP  
priority 101
```

[-P, -preempt]

problems with keepalived

- switchover requires config reload
- does not try to down or up any service
- IP flapping
- doesn't really do a cluster-level consensus

low-level cluster-oriented tool

```
SELECT pg_catalog.pg_last_xlog_receive_location();
```

```
pg_last_xlog_receive_location
```

```
-----
```

```
0/29004560
```

```
(1 row)
```

property check

- treats the service as inherently singleton ✗
- high availability ✓
- fail-overs are automatic ~
- consumers should be able to refresh connections ✓
- zero / minimal application state ✓

cluster-oriented

- repmgrd
- Heartbeat / Pacemaker

repmgr

- maintains its own db and table with the node information and replication state

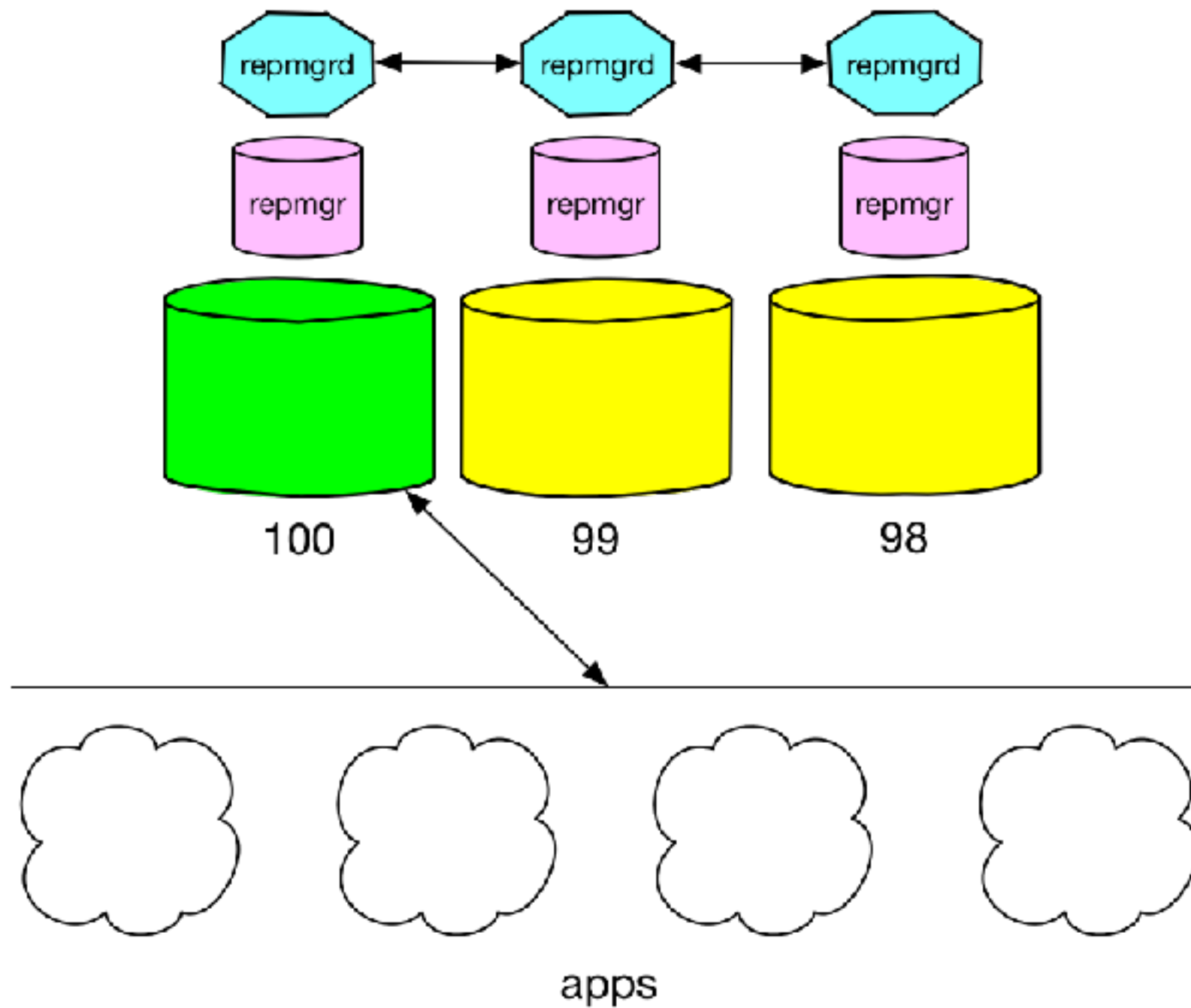
id	type	upstream_node_id	cluster	name	conninfo	slot_name	priority	active
1	master		test	node1	localhost	rs1	100	t
2	standby	1	test	node2	localhost	rs2	99	t
3	standby	1	test	node3	localhost	rs3	98	t

repmgr

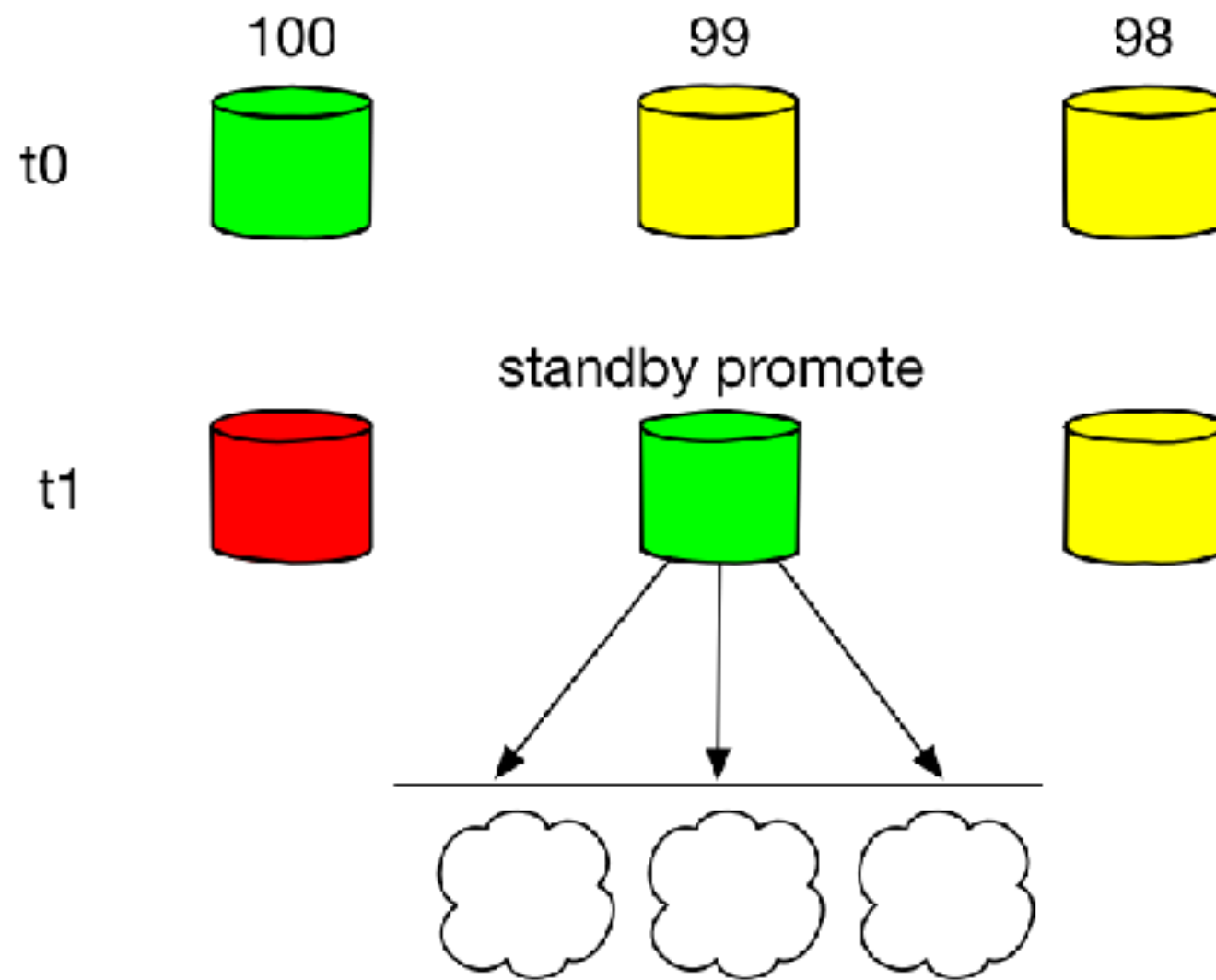
```
$ repmgr -f /apps/repmgr.conf standby register
```

- master register
- standby register
- standby clone
- standby promote
- standby follow
- cluster show

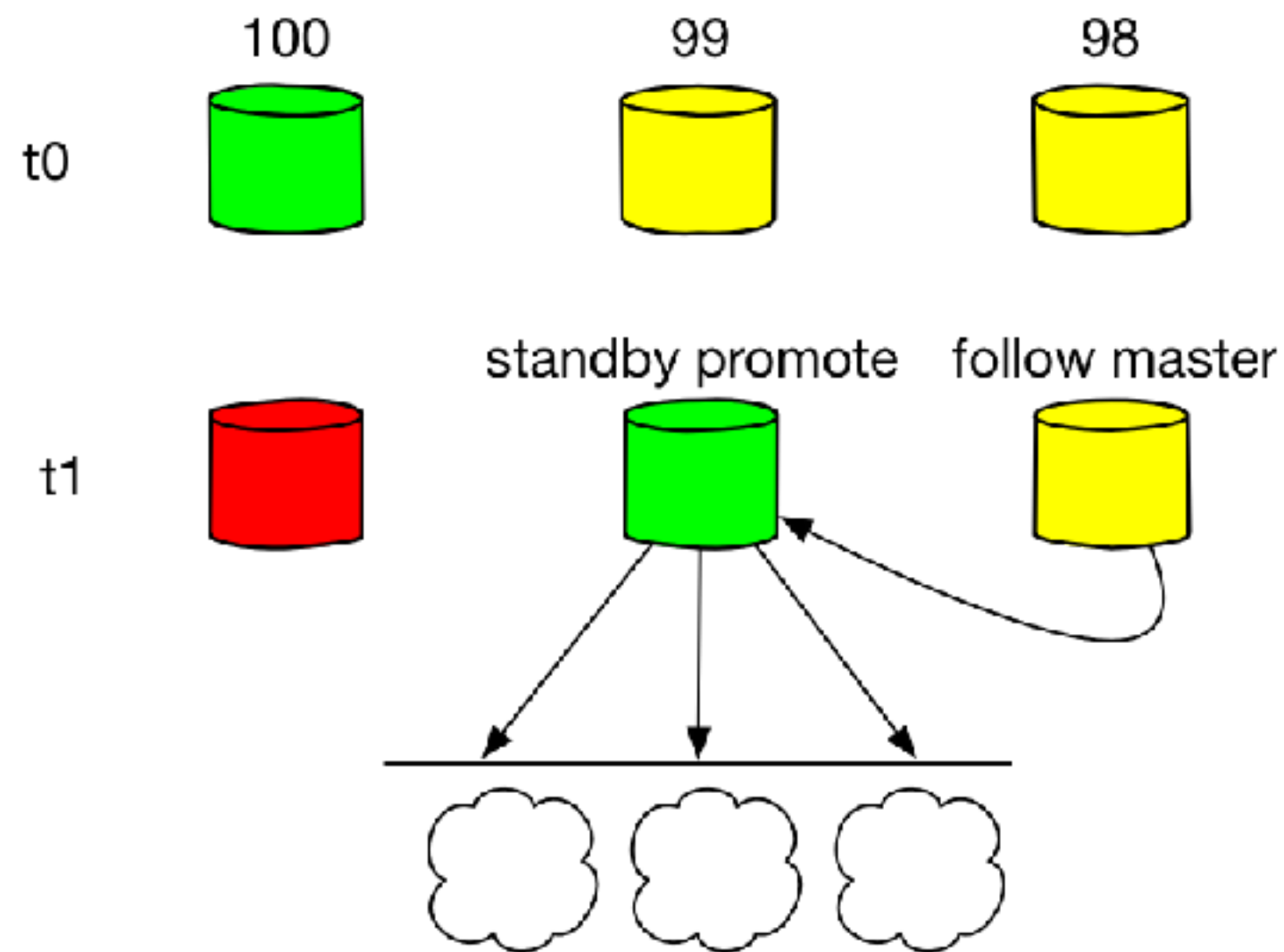
repmgrd



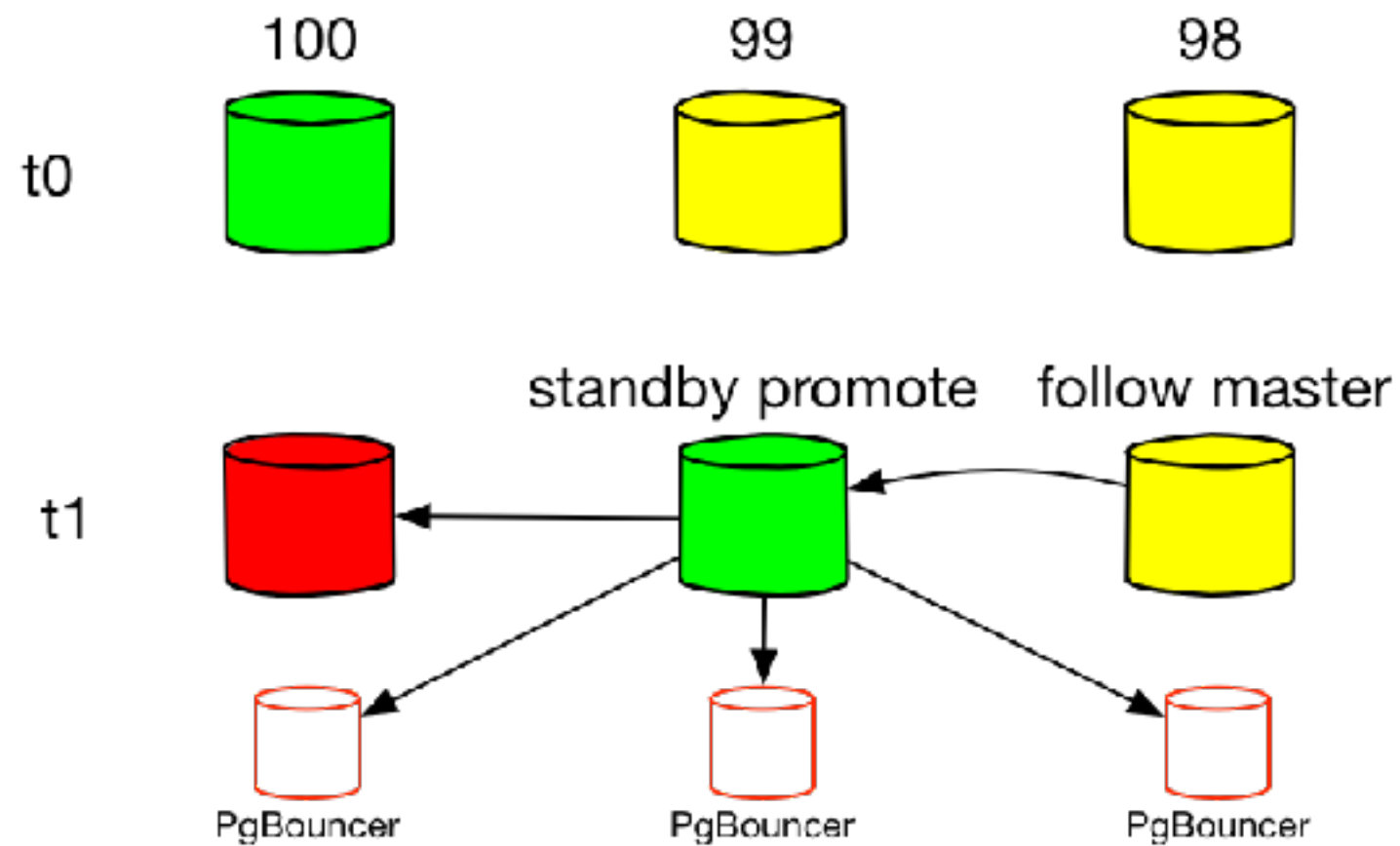
repmgrd



repmgrd - follow master



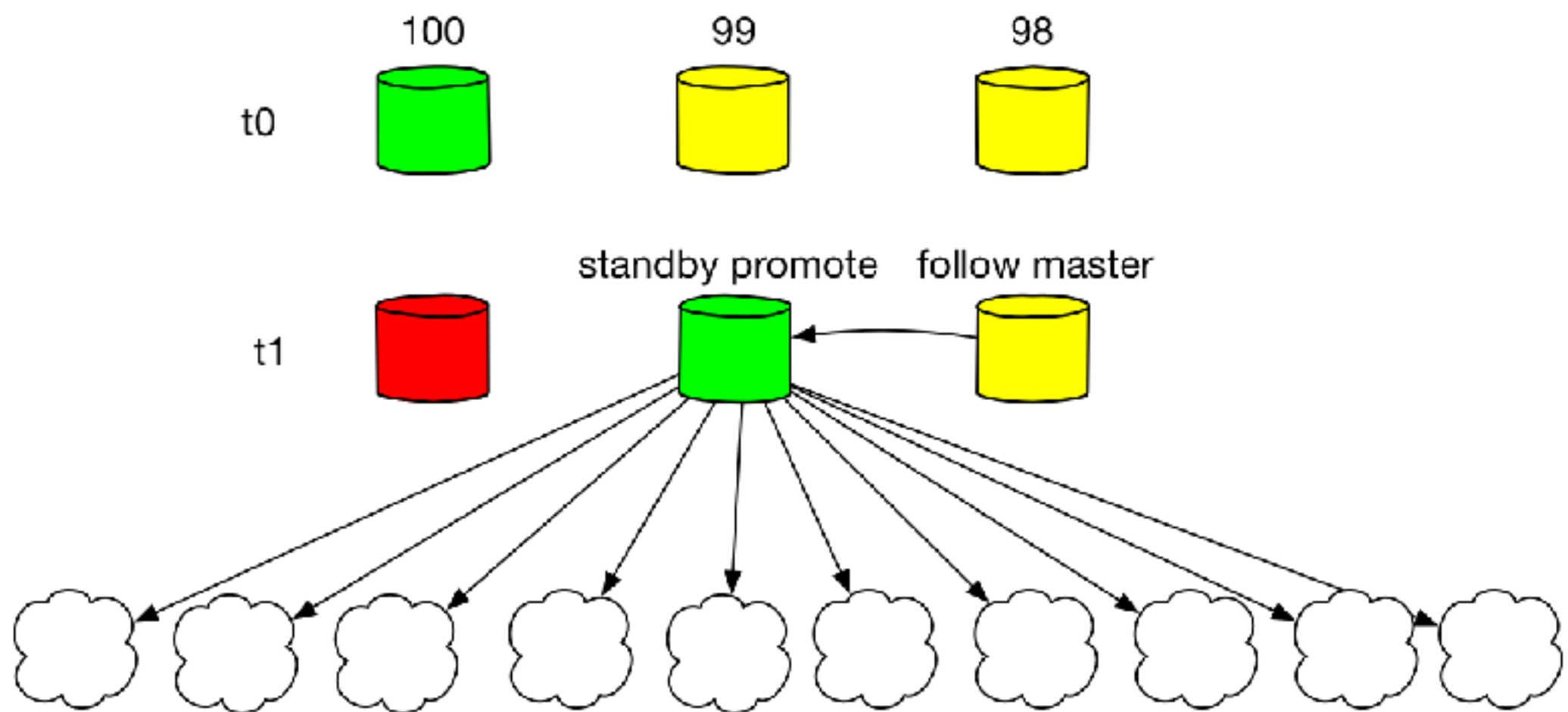
repmgrd - PgBouncer



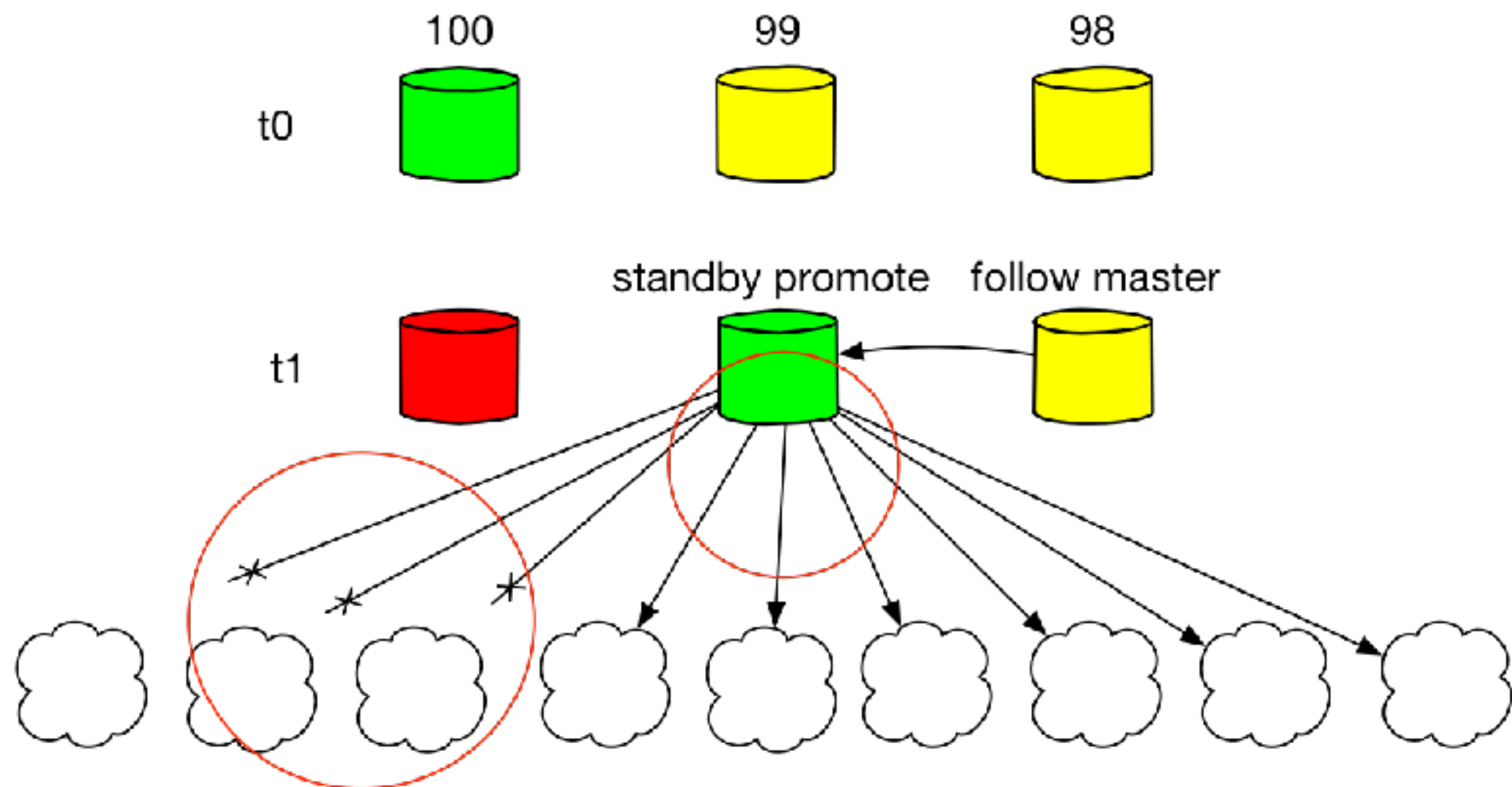
low-level node health

- the healthiest node will be picked for nodes that share the priority
- <https://github.com/nilenso/postgres-docker-cluster/blob/master/tests/sync-failover.sh>

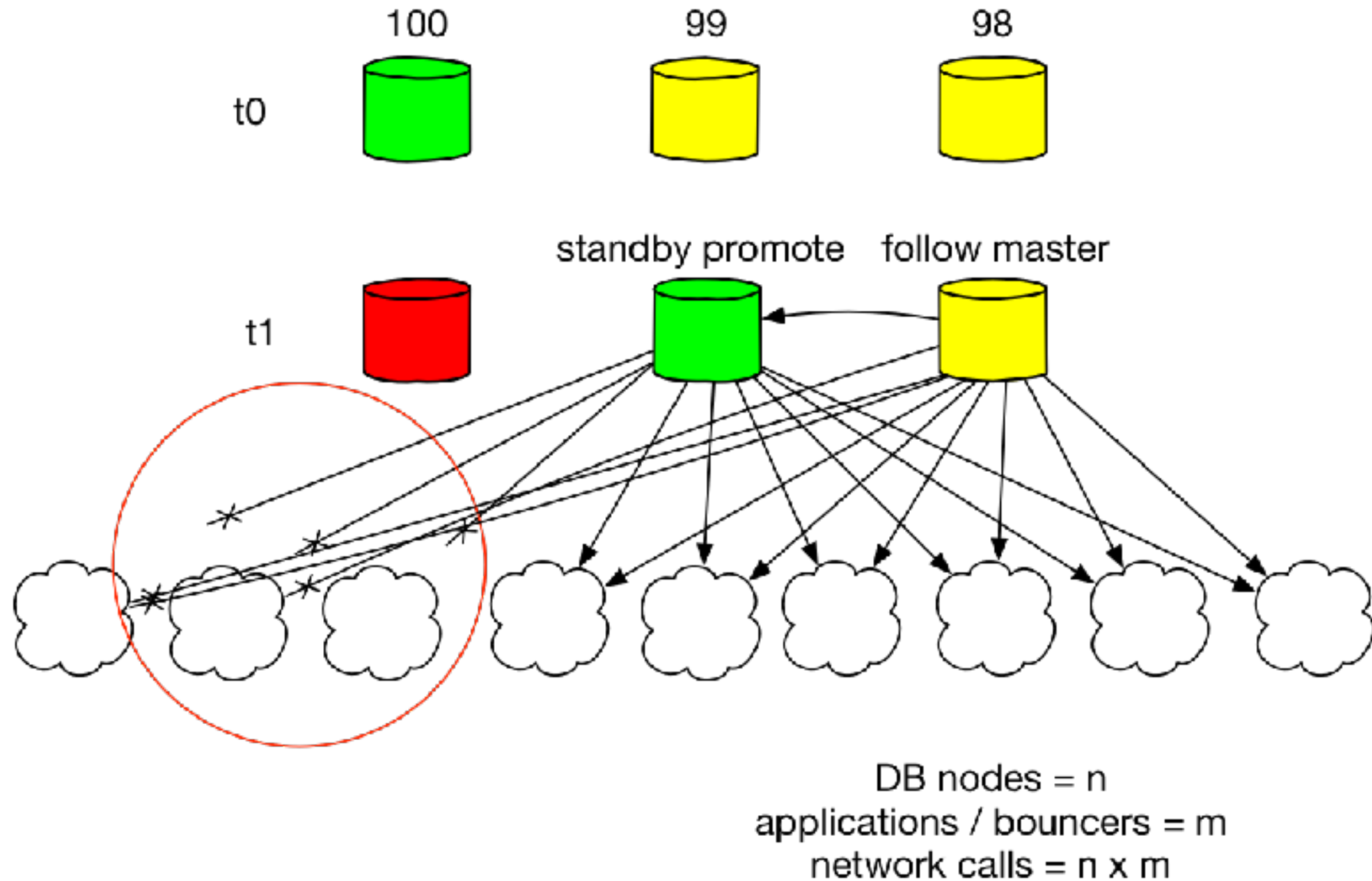
scale



points of failure



scale (absurd growth)



small anecdote

- we were running a system with strict SLAs (< 10ms, 99 percentile, 2000 RPS)
- we were not confident with just having the trigger mechanism, as it was too prone to failures, needed high availability
- running an application connection pool for performance reasons
- so we added another line of defence

more lines of defence

- custom master check
- list through all nodes and see if we can do an **INSERT**

kinds of push mechanisms

- synchronous: ensure that the promotion is dependent on **n** number of apps returning a successful response (atomic to some degree)
- asynchronous: promote first, fire notifications to all your apps

The actual script is as follows; adjust the configurable items as appropriate:

```
/var/lib/postgres/repmgr/promote.sh
```

```
#!/usr/bin/env bash
set -u
set -e

# Configurable items
PGBOUNCER_HOSTS="node1 node2 node3"
PGBOUNCER_DATABASE_INI="/etc/pgbouncer.database.ini"
PGBOUNCER_DATABASE="appdb"
PGBOUNCER_PORT=6432

REPMGR_DB="repmgr"
REPMGR_USER="repmgr"
REPMGR_SCHEMA="repmgr_test"

# 1. Pause running pgbouncer instances
for HOST in $PGBOUNCER_HOSTS
do
    psql -t -c "pause" -h $HOST -p $PGBOUNCER_PORT -U postgres pgbouncer
done

# 2. Promote this node from standby to master

repmgr standby promote -f /etc/repmgr.conf

# 3. Reconfigure pgbouncer instances

PGBOUNCER_DATABASE_INI_NEW="/tmp/pgbouncer.database.ini"
```

network is reliable

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

separation of concerns

- the database promotion is dependent of a large set of network calls being successful
- your database promotion script is aware of all your apps and/or bouncers
- your apps probably have an API endpoint to receive such updates

trigger mechanism

- fire and forget call at a critical decision point
- no retry logic built-in
- single point of failure (hopefully never make **$n*m$** calls)

property check

- treats the service as inherently singleton ✓
- high availability ✓
- fail-overs are automatic ✓
- consumers should be able to refresh connections ~
- zero / minimal application state ~

proposition

increase the area of success

- writing to the database ✓
- failing over to a new database
- telling your clients about the master database

`repmgrd` is good, could be better

- it's battle-tested
- has worked reliably in the past

poll and then push

- poll outside the promotion script, build better retries
- frees up the database promotion from any external services
- many nodes can publish the same information

polling allows heartbeats

- heartbeat / monitoring
- keep an eye on `repmgrd`
- zookeeper node monitoring

push to a central datastore

- we don't want to depend on the complex network graph and build retry mechanism around apps
- we want to avoid application state and exposing API endpoints

central datastore

- battle-tested
- strongly consistent
 - operations appear to execute atomically
- highly available

zookeeper

- compare and swap / atomic operations
- ephemeral nodes

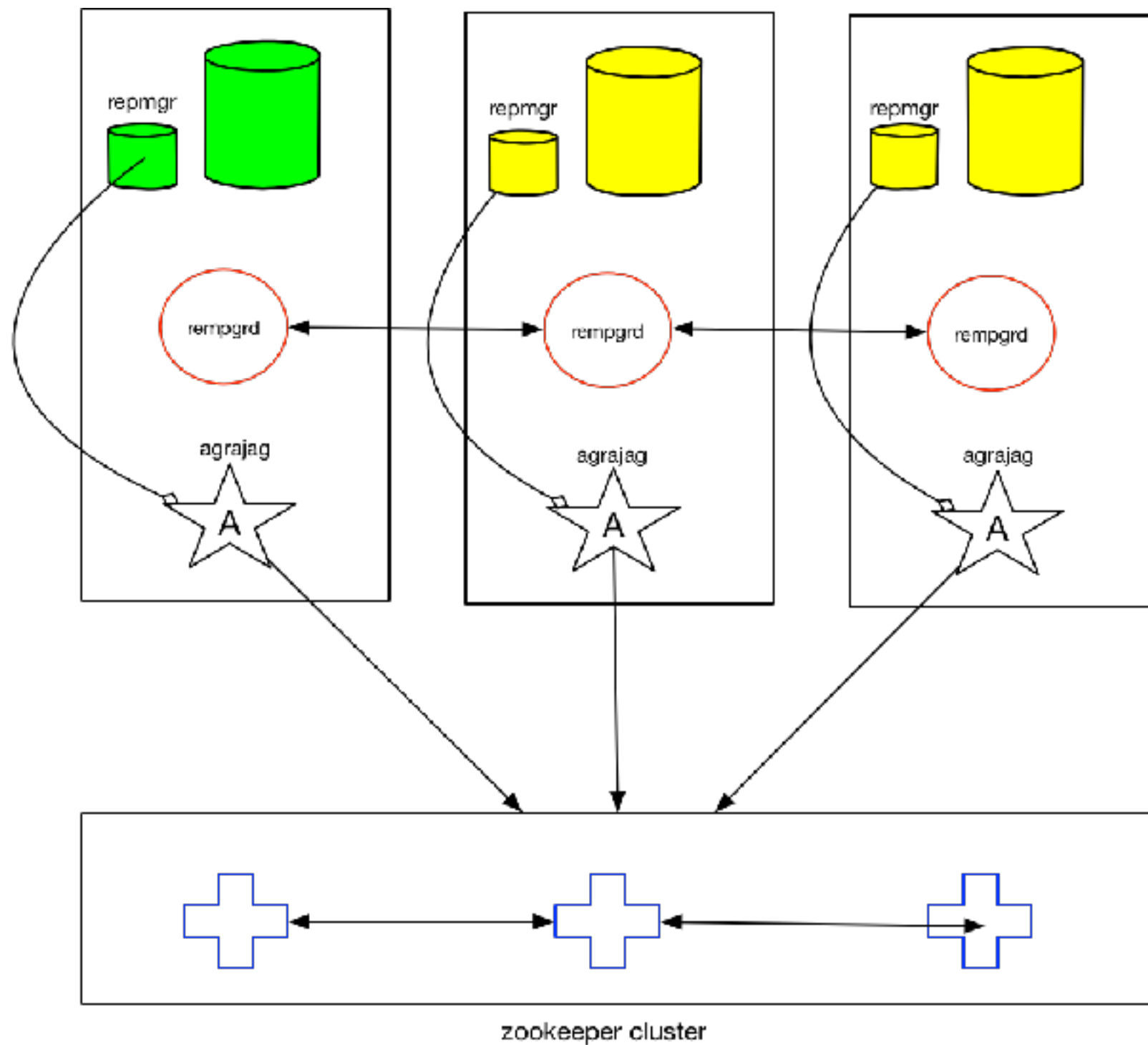
Agrajag

<https://github.com/staples-sparx/Agrajag>

Agrajag



the orchestrator



repmgr cluster show

```
$ repmgr -f /etc/repmgr.conf cluster show
```

Role	Name	Upstream	Connection String
* master	node1		host =db_node1 dbname=repmgr user=repmgr
standby	node2	node1	host =db_node2 dbname=repmgr user=repmgr
standby	node3	node2	host =db_node3 dbname=repmgr user=repmgr

repmgr repl_events

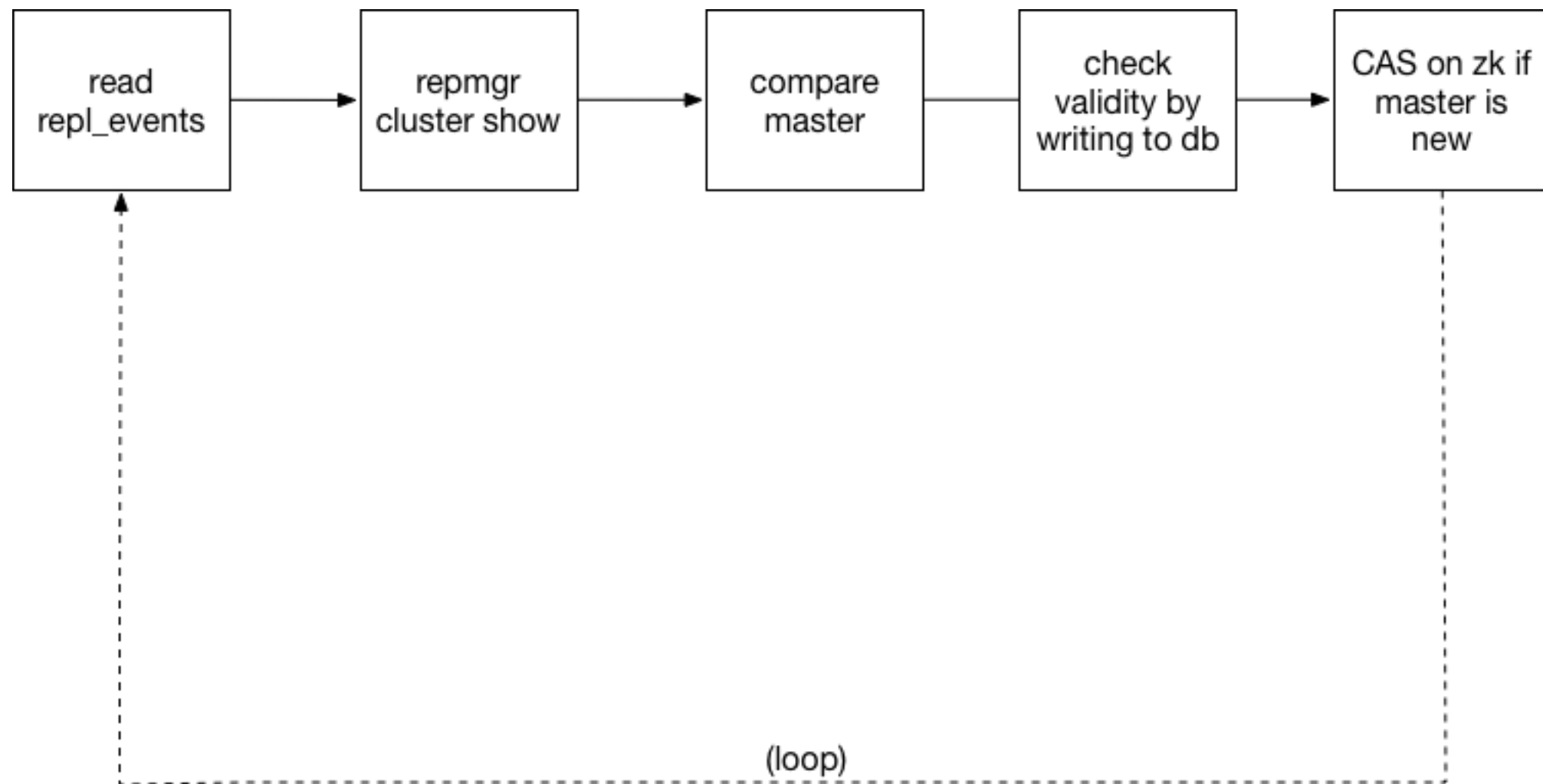
```
repmgr=# SELECT * from repmgr_test.repl_events;
```

node_id	event	successful	event_timestamp	details
1	master_register	t	2016-01-08	
2	standby_clone	t	2016-01-08	
2	standby_register	t	2016-01-08	

zookeeper CAS

```
1 (defn compare-and-set
2   [client path predicate? new-data & args]
3   (try (let [zk-data (get-data path)
4             deserialized (when-let [bytes (:data zk-data)]
                              (deserializer bytes))
5             version (-> zk-data :stat :version)]
6     (when (predicate? deserialized)
7       (log/debug "Pubilshing data to zk for" path)
8       (zk/set-data client path (serializer new-data) version)))
9   (catch KeeperException$BadVersionException bve
10    (log/warn "Trying to do stale write" bve))))
```

Agrajag



the orchestrator

- `repmgrd` is dead, do a manual master check
- second line of defence built into the orchestrator

back-of-the-envelope concerns

- stateful reads from zookeeper
- dependency on orchestrator
- network partitions
- app knows about the upstream datastore
- directly update HaProxy

reads from zookeeper

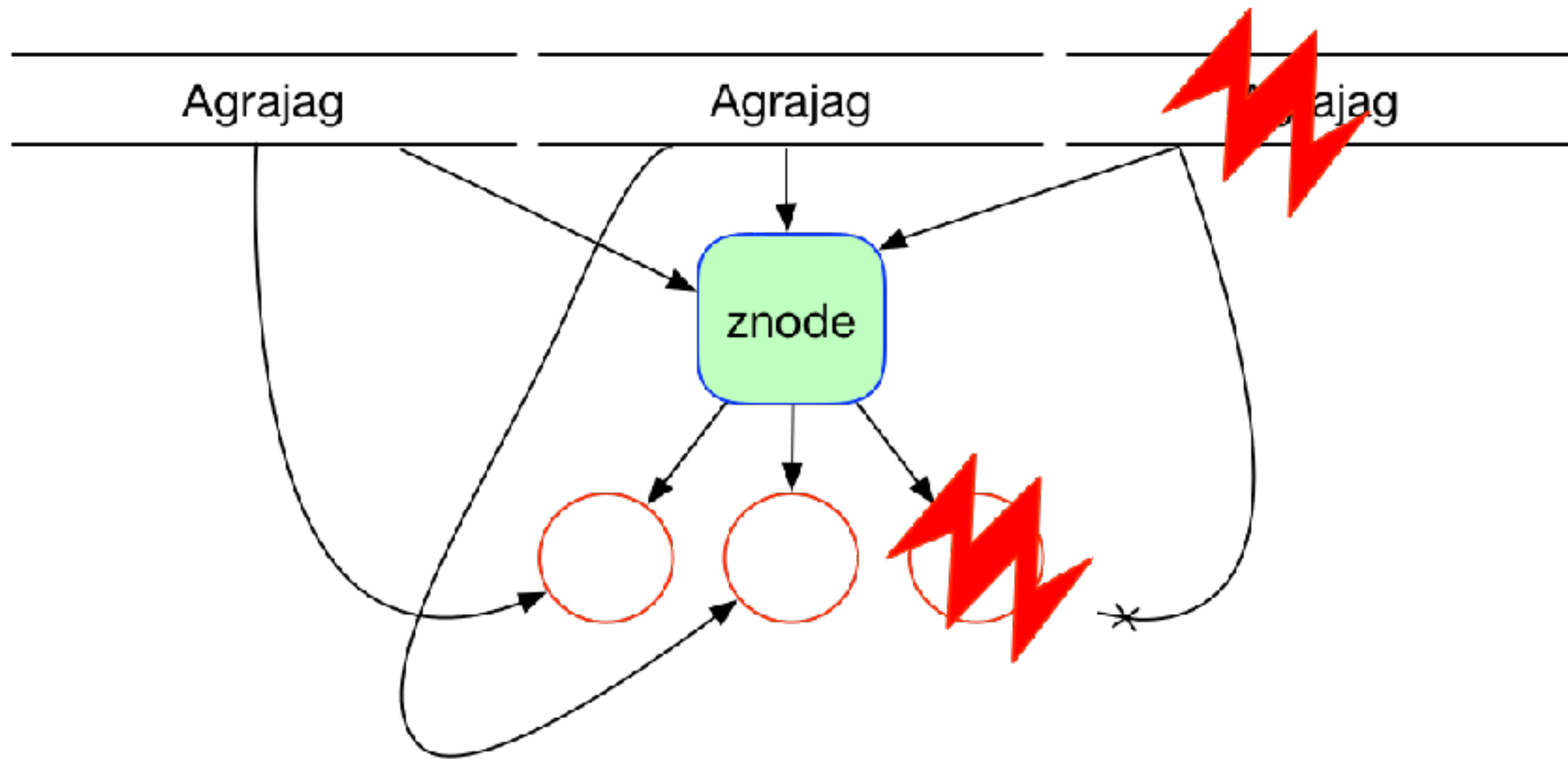
- the application / bouncer still needs to read from zookeeper
- is it really stateless?

zookeeper watchers

- allow stateless reads from zookeeper

```
1 (defn get-data
2   ([path]
3     (get-data path false nil))
4   ([path watch? watcher-fn]
5     (-> (zk/data @client path :watch? watch? :watcher watcher-fn)
6         :data
7         deserializer)))
```


zookeeper ephemeral nodes



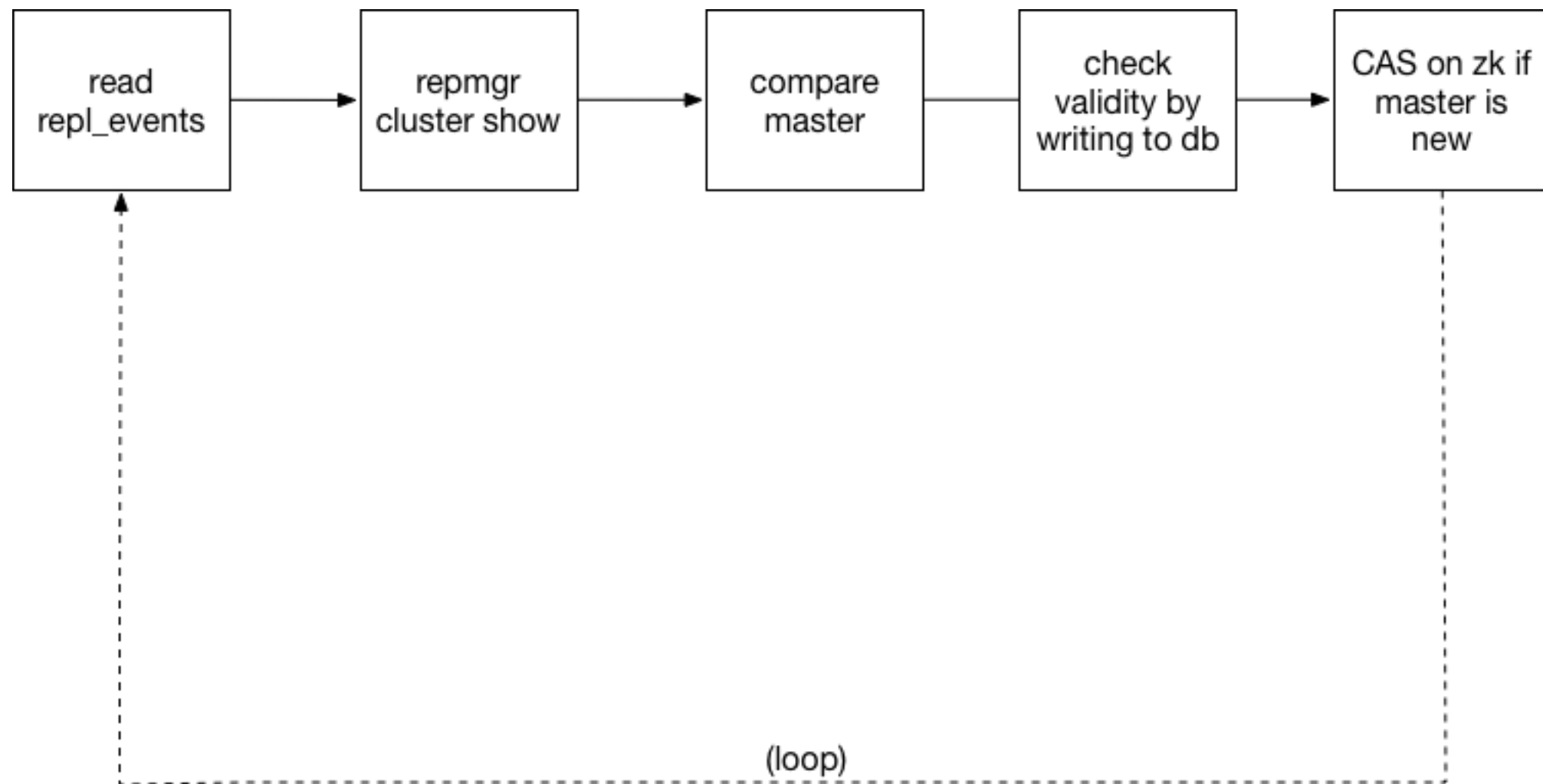
zookeeper ephemeral nodes

- proceed only if master is present
- ephemeral nodes > 0 for the master znode

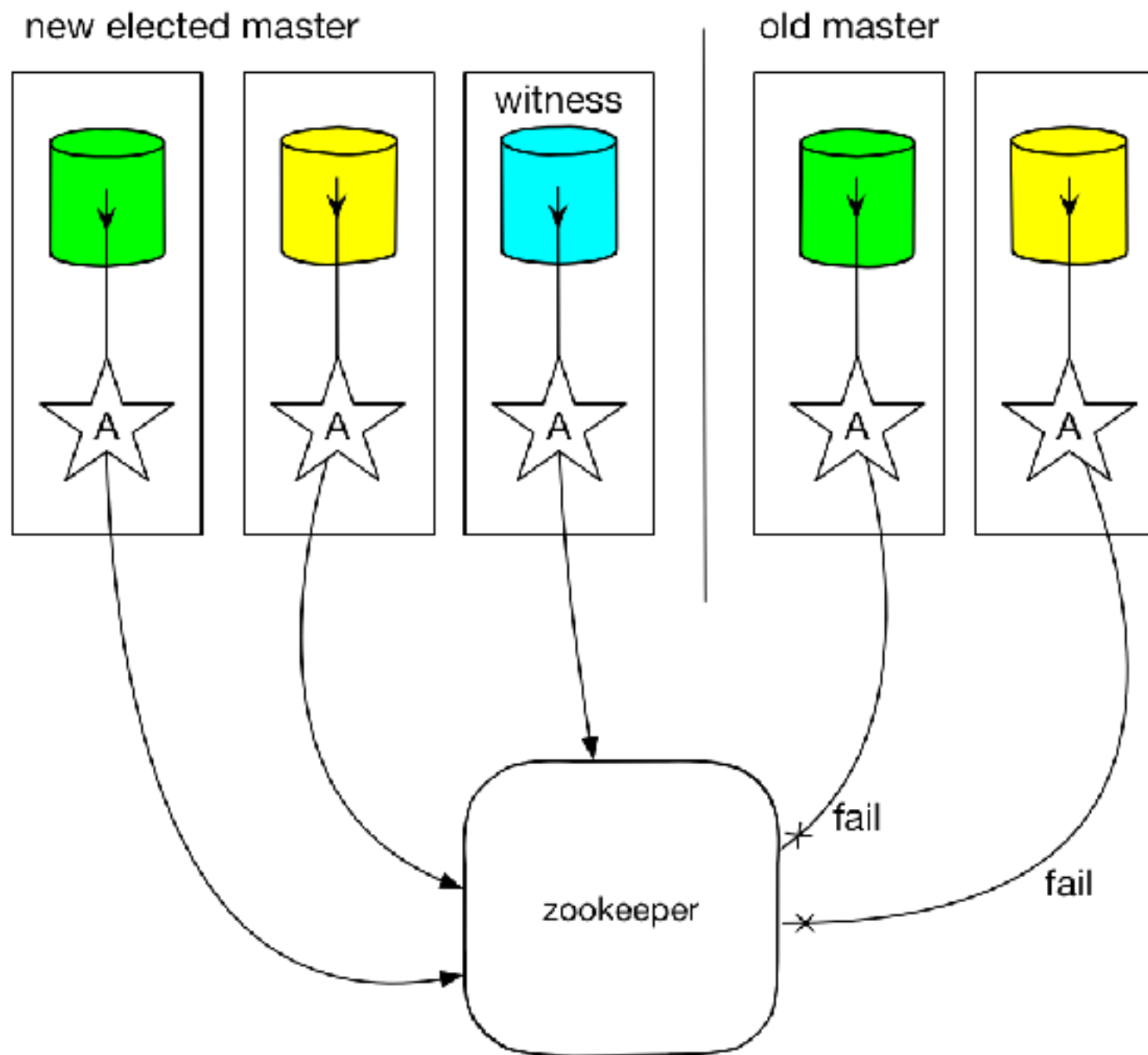
split-brain

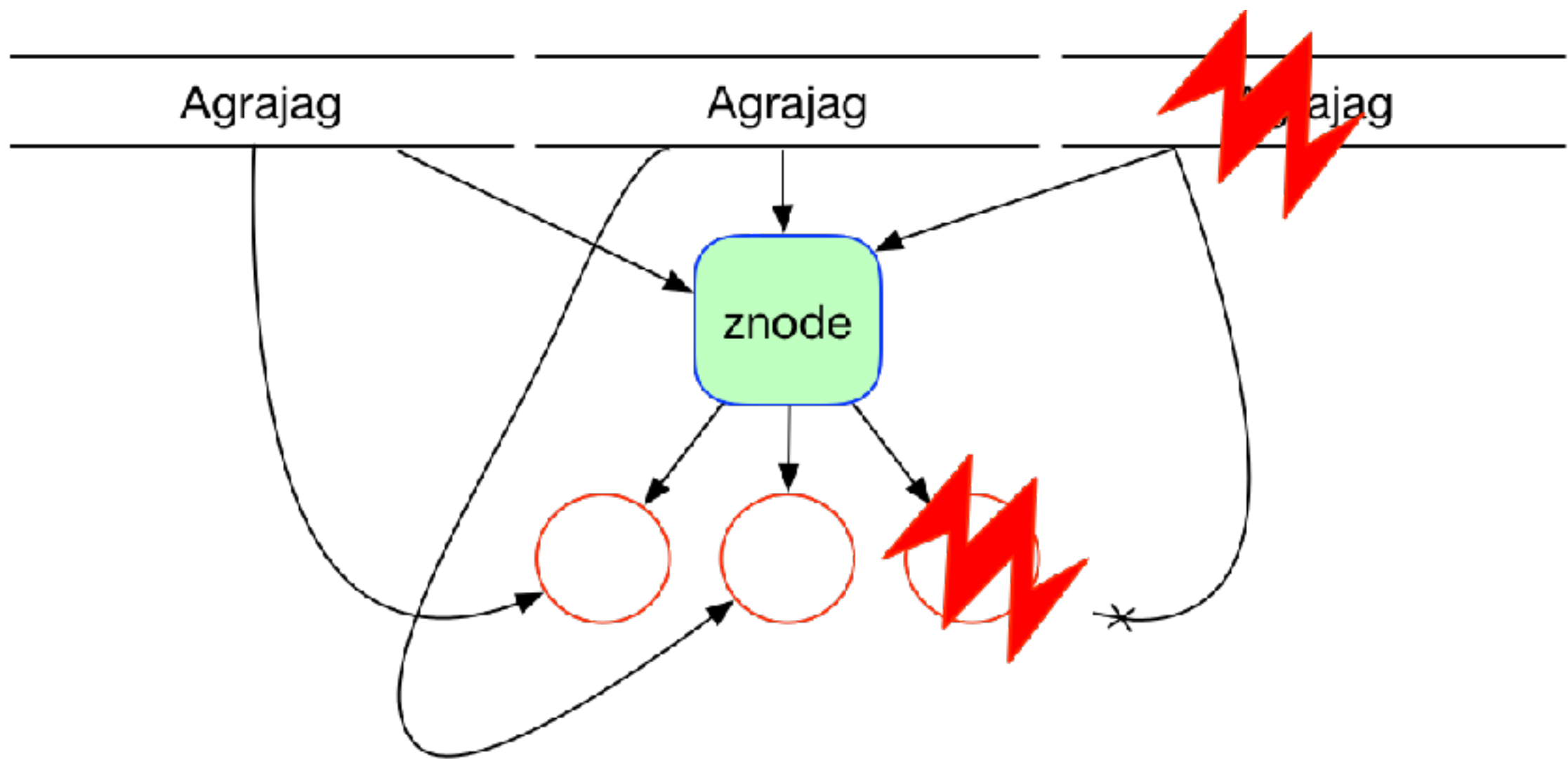
- have a witness server setup with repmgr
- will implicitly fence the old master based on promotion times

Agrajag

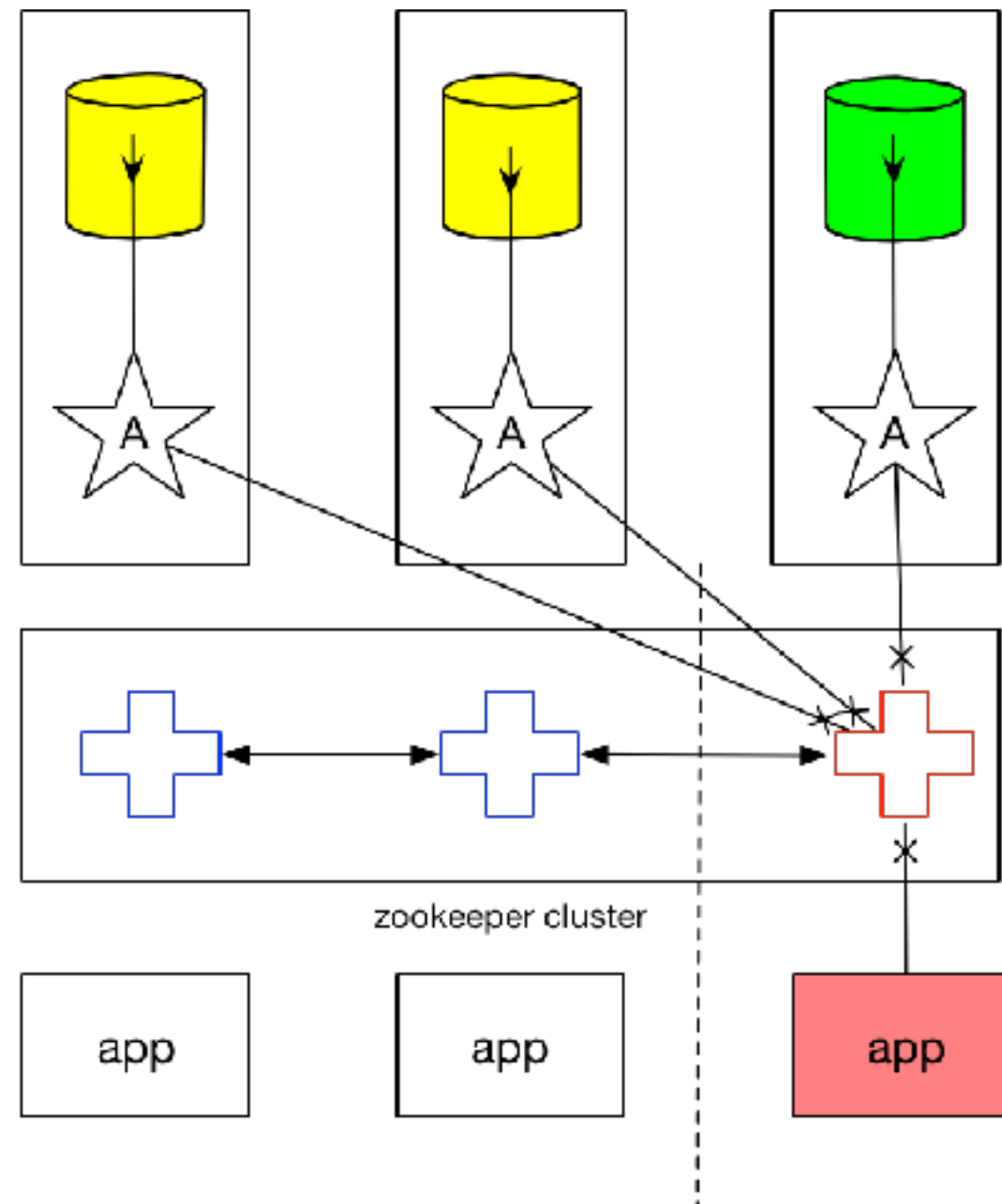


split-brain





network split in zookeeper



app still has information about
an upstream dependency

- move it to `conf.d`
- write your own Agrajag client

but now the watchers are prone
to failures

- but they read from one central place we can trust
- multiple nodes get a chance to update that same place
- client libraries for zookeeper already exist, tried and tested

“just directly update HAProxy”

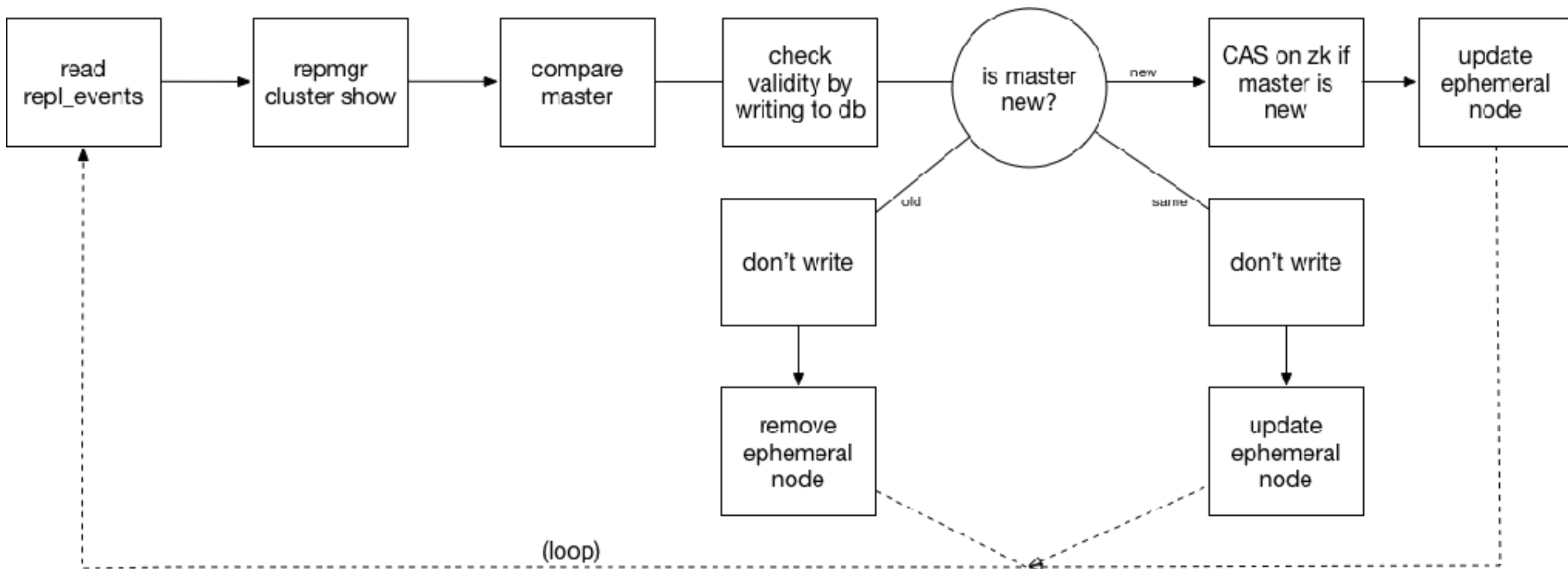
- good idea
- does not have compare-and-swap

speed

```
reconnect_attempts=6  
reconnect_interval=10
```

Lines (15 slots)	542 Bytes
1	{:frequency-ms 5000
2	:repmgr {:config-file
3	:zookeeper {:connect "
4	:master-pa
5	:db-spec {:auto-commit
6	:connection-
-	-

Agrajag



re-invent?
re-write?
plug holes?

- slapping HA on top of HA
- where do you draw the line?

network is reliable

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

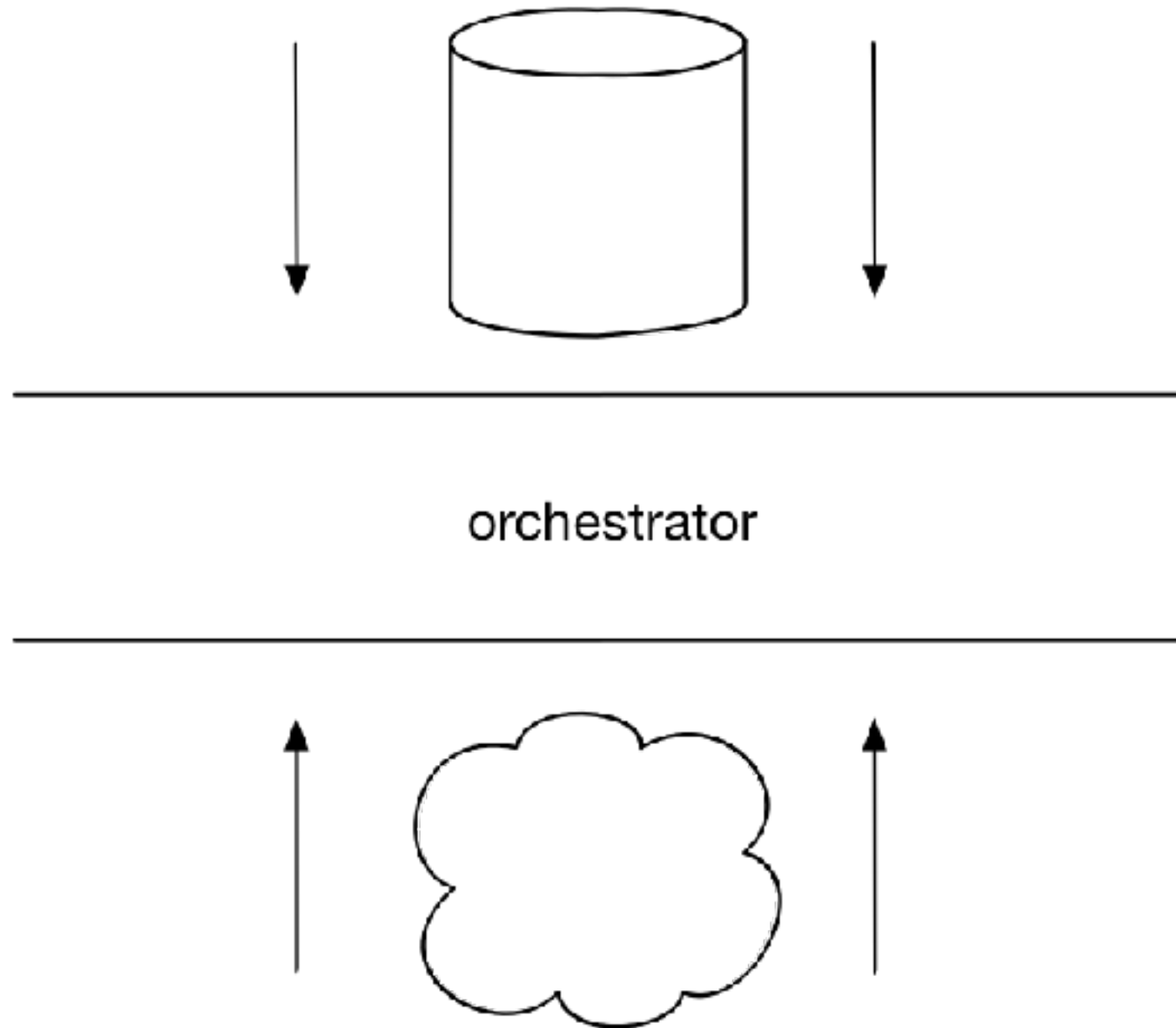
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

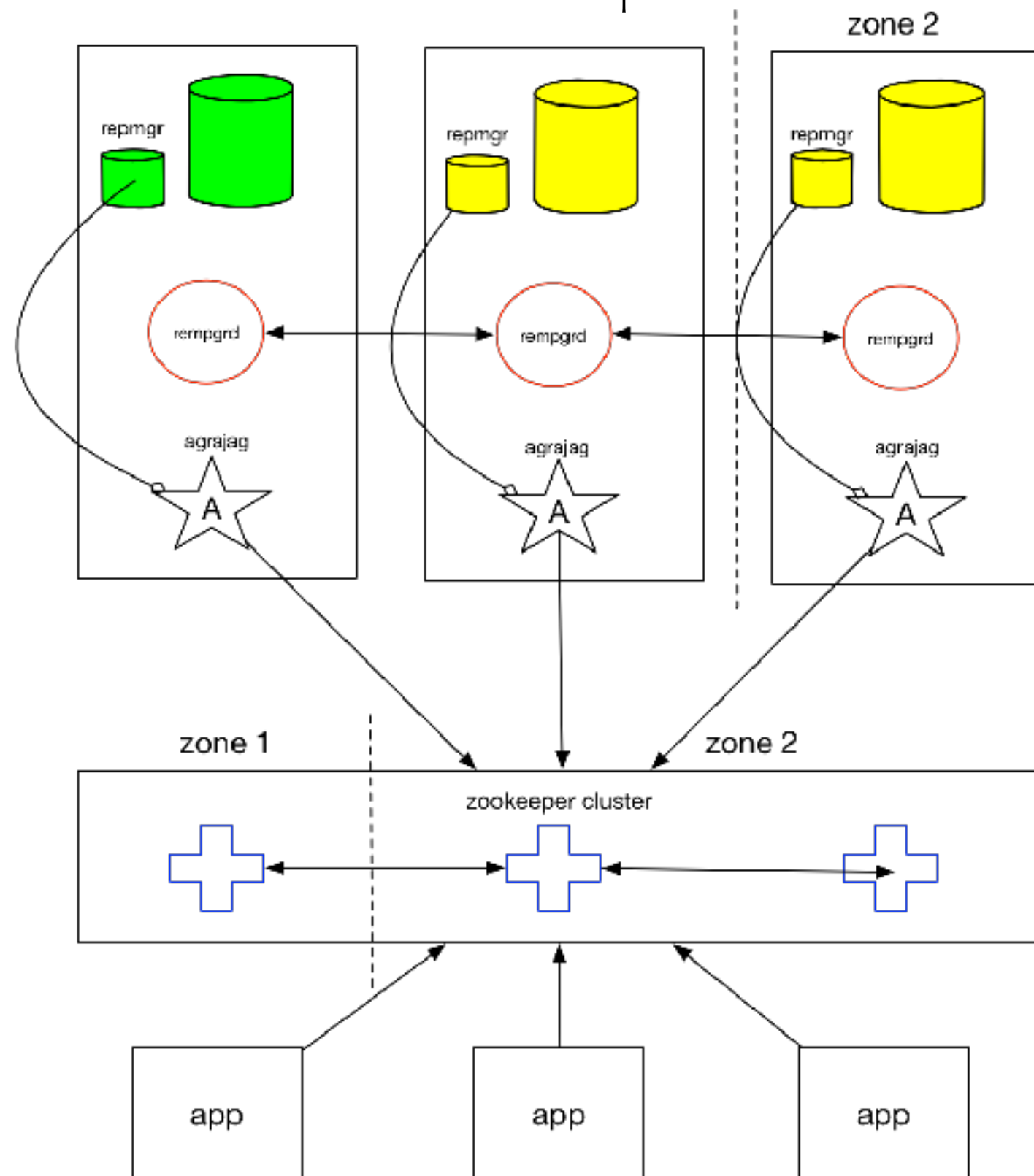
avoid consensus if possible

- consensus is done, plug holes
- only add new facts to the system
- avoid co-ordination

the orchestrator



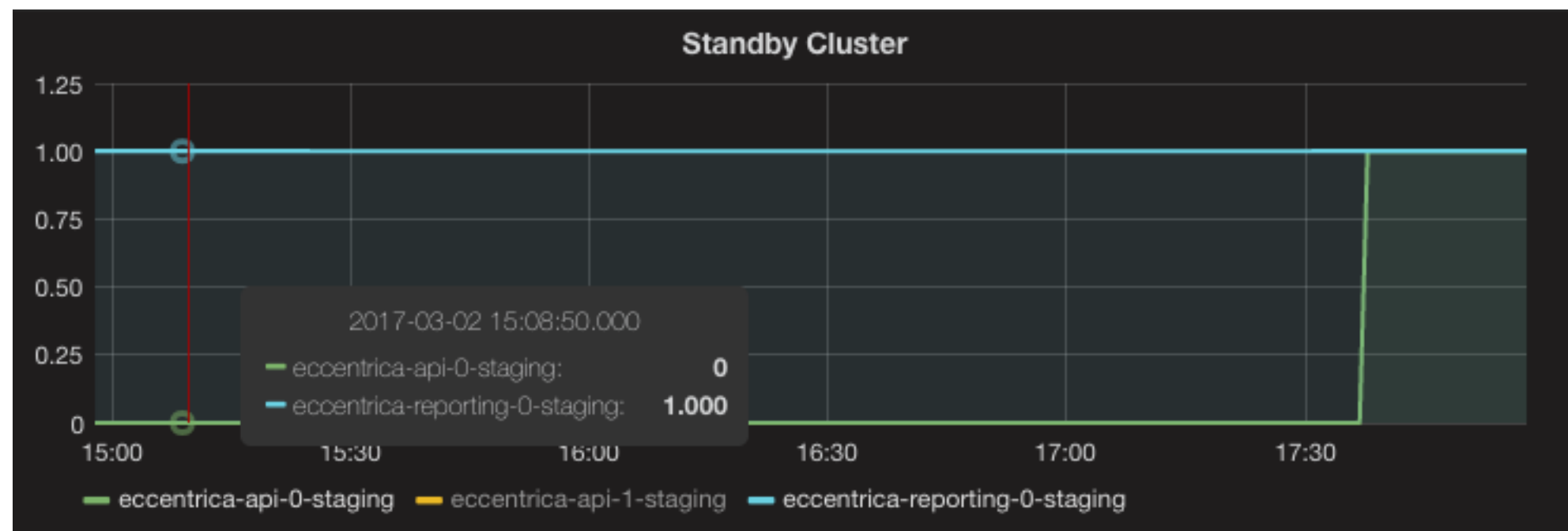
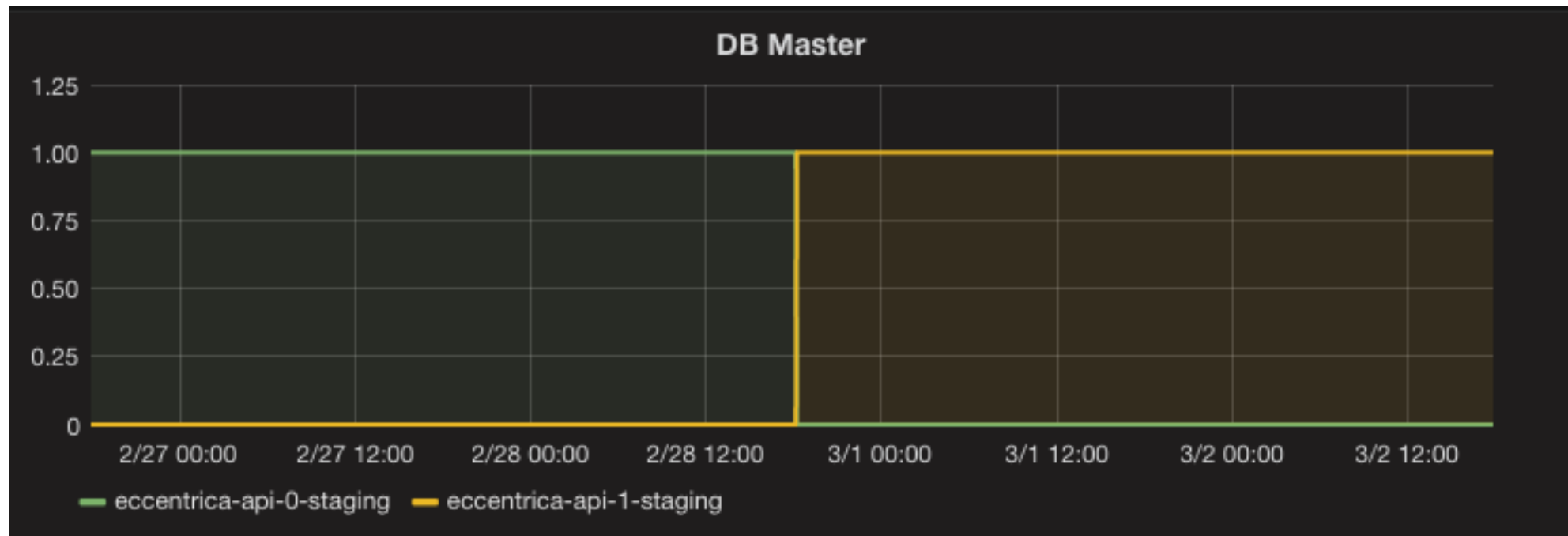
recap



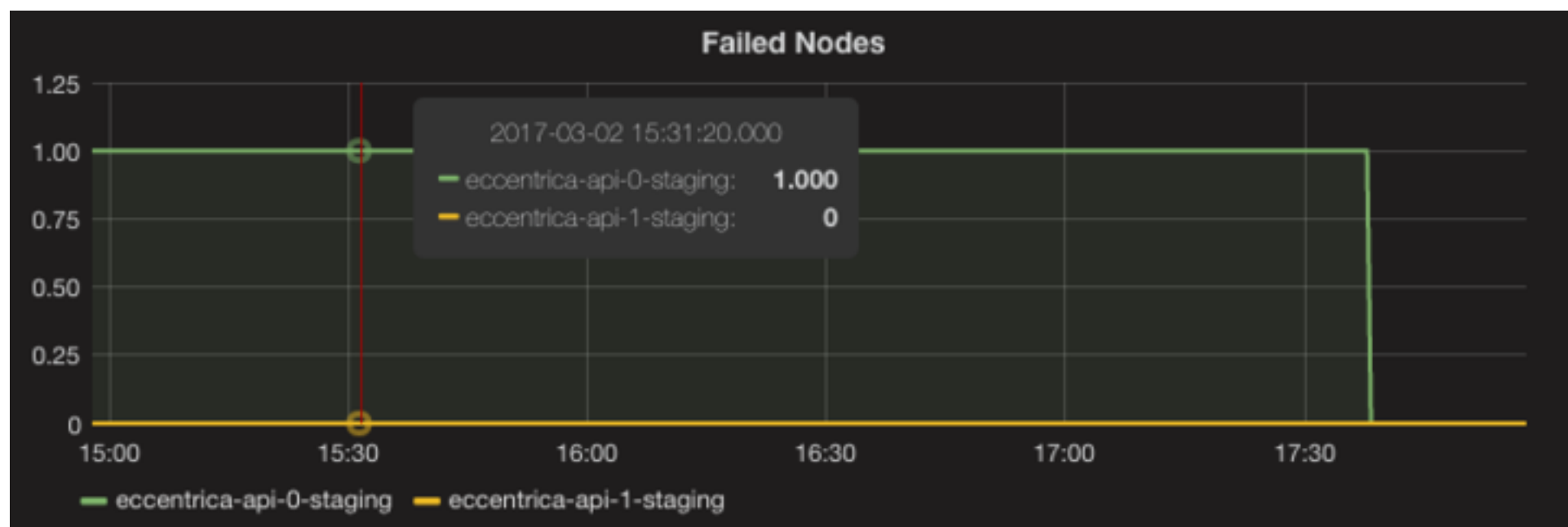
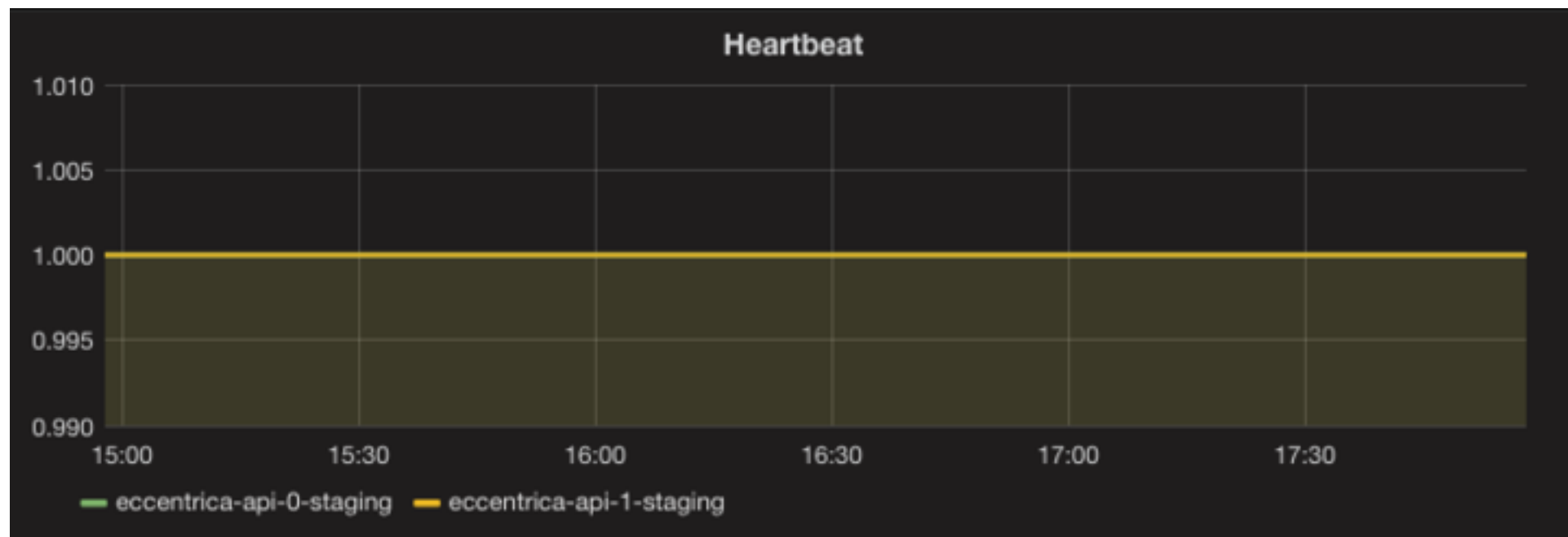
future

- /failover API endpoint
- another line of defence
- push and then poll and then broadcast

monitor all the things



monitor all the things



caveats and hedges

- repmgrd might not be the best at consensus
- Agrajag is very beta
- adding another layer of dependency
- not one-size-fits-all

caveats and hedges - what about PgBouncer?

- the most appropriate use-case for this orchestration layer is when you have an application-level connection pool
- when PgBouncer is sitting on the application layer

plz help

Who is the master? #259



kitallis opened this issue on 19 Dec 2016 • 0 comments



kitallis commented on 19 Dec 2016



I've been working on this project <https://github.com/staples-sparx/repmgr-to-zk> that will help post the current master status to ZK/Kafka, so that HAProxy / the application can read the current master and refresh the connections. It's similar to <https://github.com/compose/governor>, except that it relies on repmgr for cluster management and failovers.

I'm parsing the `cluster show` command (from all nodes) to get the correct master, but I'm not sure if that's the latest source of truth. I'm also not entirely sure why the CSV output doesn't post the master information.

What is the best way to find out the latest master from repmgr?

takeaway

- apply these axioms / properties to your current system
- test thoroughly
- tell me I'm wrong 🙈

open questions

- what about Patroni and Stolon?
- what about Heartbeat / Pacemaker?
- what about PgBouncer?



references

- <https://github.com/aphyr/distsys-class>
- <https://aphyr.com/posts/291-call-me-maybe-zookeeper>
- https://wiki.postgresql.org/images/1/1e/QuanHa_PGConf_US_2016_-_Doc.pdf
- <http://www.formilux.org/archives/haproxy/1003/3259.html>
- www.interdb.jp/pg/pgsql11.html
- <https://github.com/staples-sparx/Agrajag/blob/dev/doc/tradeoffs.md>