Implementing MySQL Database-as-a-Service using Open Source tools

Matthias Crauwels

SCaLE 18x, Pasadena, CA, USA
March 6, 2020

Pythian

Who am I?

Matthias Crauwels

- Living in Ghent, Belgium
- Bachelor Computer Science
- ~20 years Linux user / admin
- ~10 years PHP developer
- ~8 years MySQL DBA
- 3rd year at Pythian
- Currently Lead Database Consultant
- Father of Leander

AGENDA

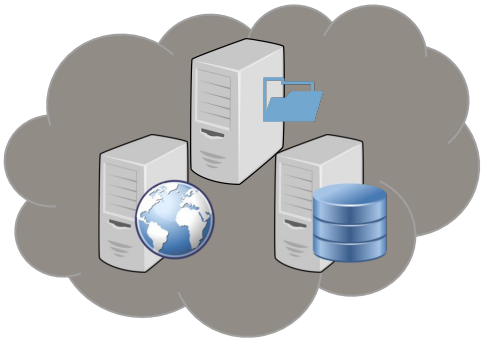
Introduction and history
DBaaS: frontend
DBaaS: backend
Communication

Let's get started!

History

You start a new application, in many cases on a LAMP stack

- Linux
- Apache
- MySQL
- PHP



Everything on a single server!

History

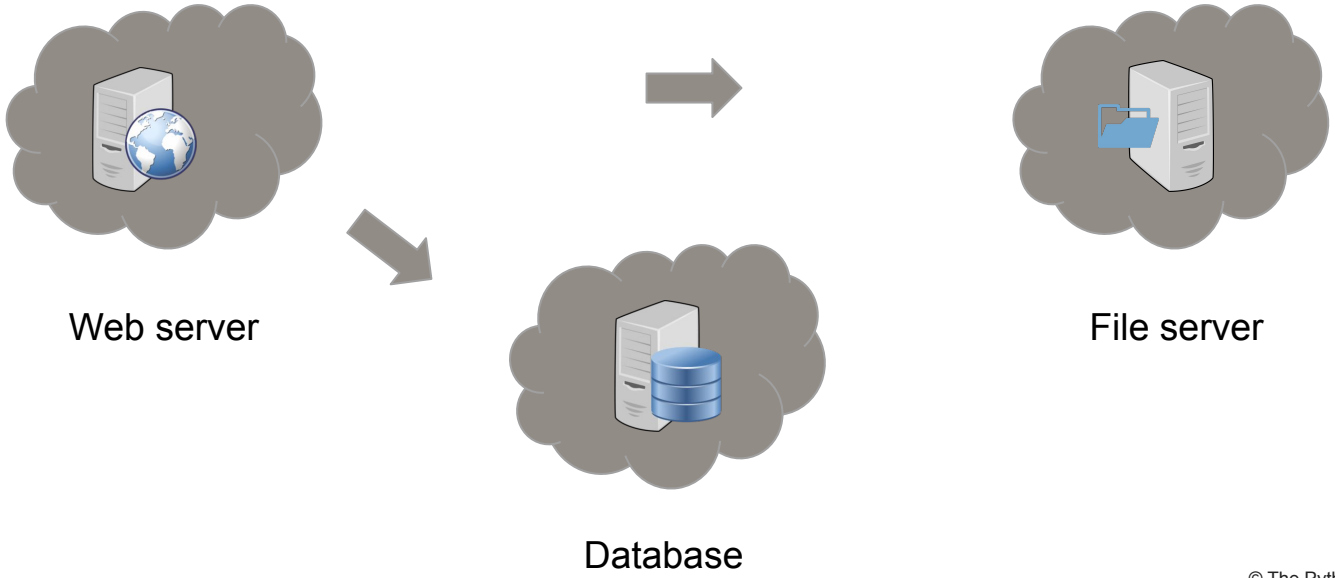
Your application grows… What do you do?

You buy a bigger server!

History

You application grows even more. Yay!

You buy **more** servers and **split** your infrastructure.



History

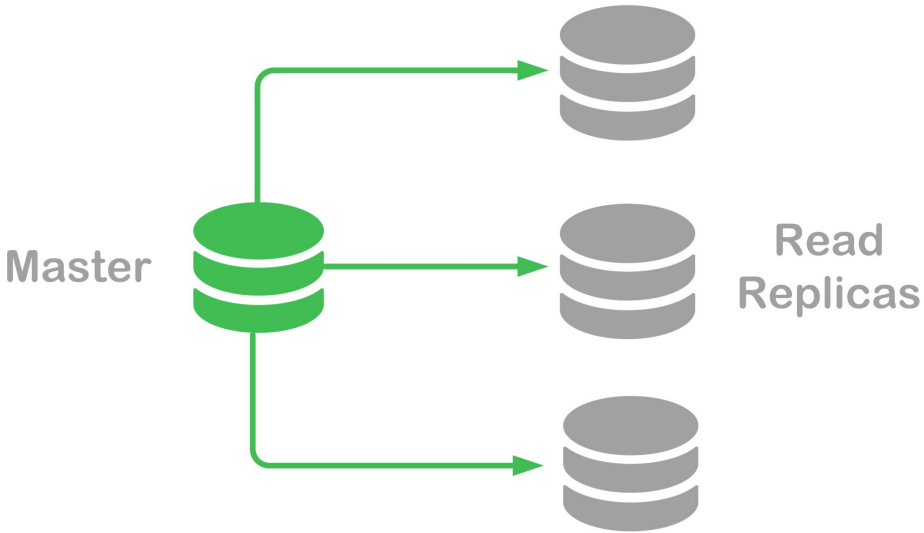
Your application grows even more!

- You scale up the components
  - Web servers are easy, just add more and load balance
  - File servers are easy, get more/bigger disks, implement RAID solutions, ...
  - What about the database??
    - More servers?
      - Ok but what about the data?
    - I want all my web servers to see the same data.
      - Writing it on all the servers? Overhead!

History

MySQL replication

- Writing to master
- Reading from replica's (slaves)



History

Million dollar questions

- How do we know what server is the master?
- How do we know which servers are the replica’s?
- How do we manage this replication topology?
- What if the master goes down?
- What about maintenance?
- …

Database-as-a-Service

Frontend Solution

ProxySQL

ProxySQL

ProxySQL: What?

ProxySQL

ProxySQL is a high performance layer 7 proxy application for MySQL.

- It provides ‘intelligent’ load balancing of application requests onto multiple databases
- It understands the MySQL traffic that passes through it, and can split reads from writes.
- It understands the underlying database topology, whether the instances are up or down
- It shields applications from the complexity of the underlying database topology, as well as any changes to it
- ...

ProxySQL: terminology

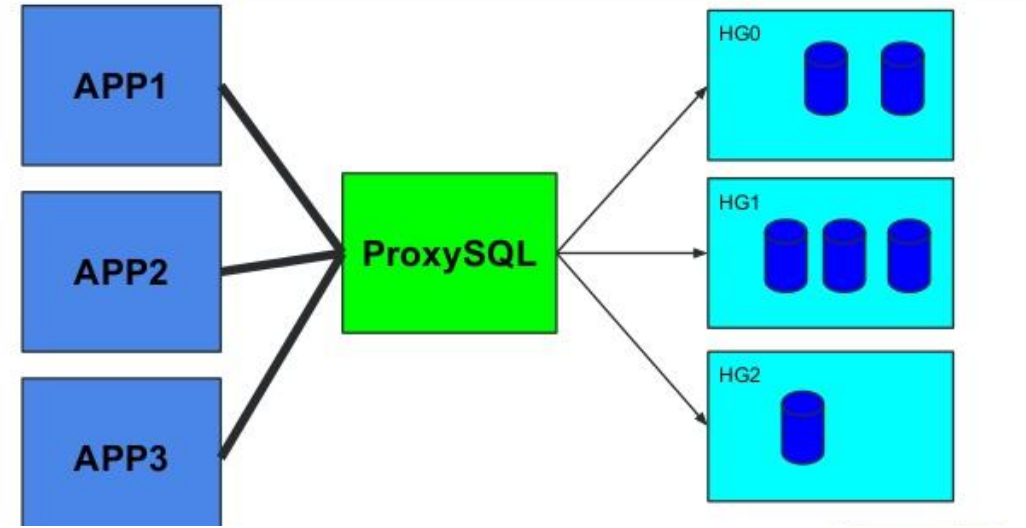
- Hostgroup

  All backend MySQL servers are grouped into hostgroups. These “hostgroups” will be used for query routing.
- Query rules

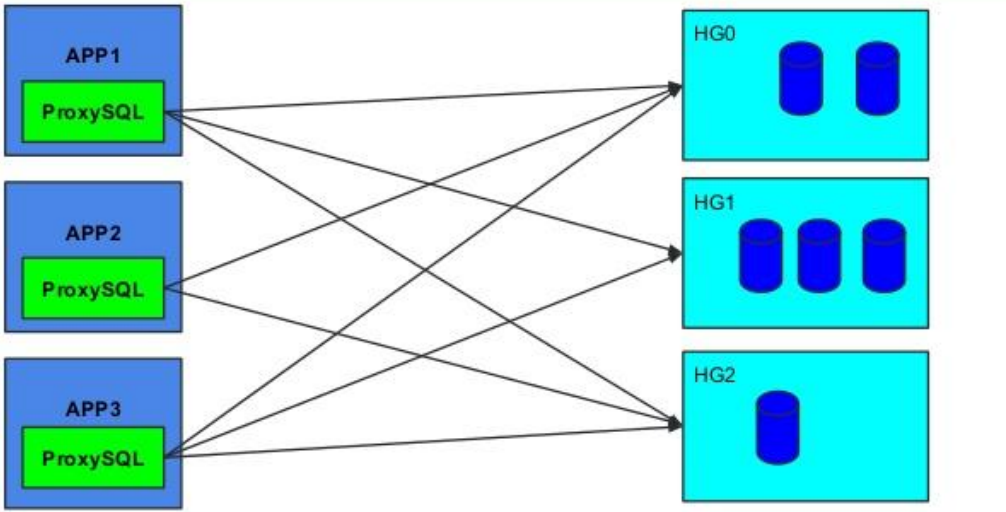
  Query rules are used for routing, mirroring, rewriting or blocking queries. They are at the heart of ProxySQL’s functionalities
- MySQL users and servers

  These are configuration items which the proxy uses to operate

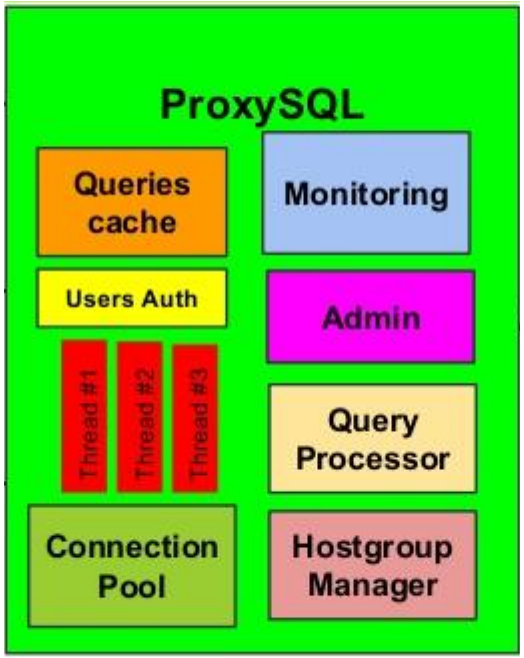
ProxySQL: Basic design (1)



ProxySQL: Basic design (2)



ProxySQL: Internals



ProxySQL: Clustering

ProxySQL will be configured to share configuration values with its peers. Currently, all instances are equal and can be used to reconfigure, there is no “master” or “leader”. This is a feature on the roadmap (https://github.com/sysown/proxysql/wiki/ProxySQL-Cluster#roadmap).

Helps to:

- Avoid your ProxySQL instance to be the single point of failure
- Avoid having to reconfigure every ProxySQL instance on the application server
- Helps to (auto-)scale the ProxySQL infrastructure

ProxySQL: Conclusions

ProxySQL exists between the application and the database.

- It hides the complexity of the database topology to the application
- It knows which server is the master and which are the slaves
- It **will not** make changes to the topology so topology management is not solved with this product.
- It has support for gracefully taking a server out of service
- It is easy to configure
- It can be clustered for not being a single-point-of-failure

Question about ProxySQL?

Database-as-a-Service

Backend management

Orchestrator

Orchestrator: What?

Orchestrator is a High Availability and replication management tool.

Originally developed by Shlomi Noach, currently developed by the Database Infrastructure Team at GitHub

It can be used for:

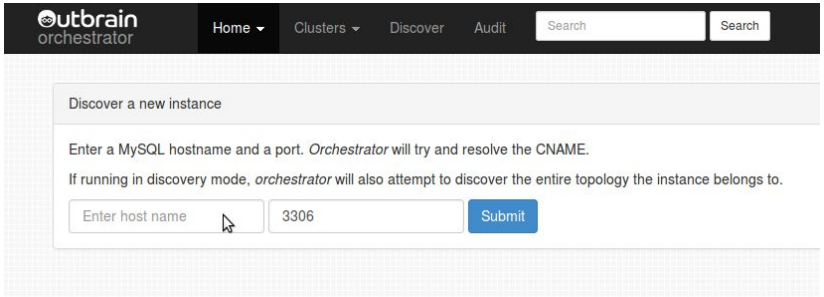
- Discovery of a topology
- Visualisation of a topology
- Refactoring of a topology
- Recovery of a topology

Orchestrator: Discovery

Orchestrator can (and will) discover your entire replication technology as soon as you connect it to a single server in the topology.

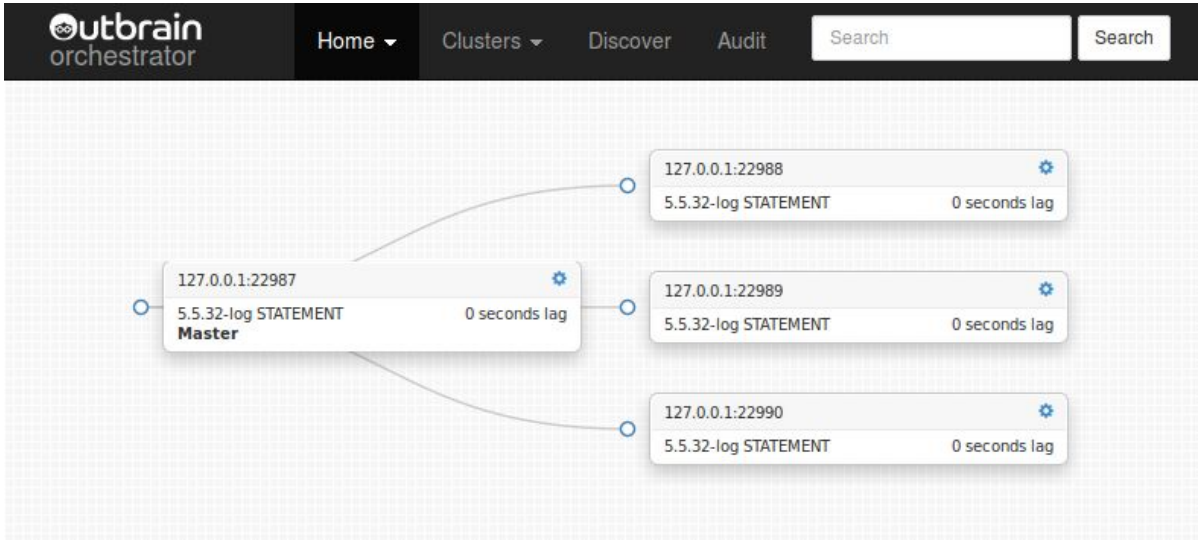
It will use SHOW SLAVE HOSTS, SHOW PROCESSLIST, SHOW SLAVE STATUS to try and connect to the other servers in the topology.

Requirement: the `orchestrator_topology_user`needs to be created on every server in the cluster so it can connect.



Orchestrator: Visualization

Orchestrator comes with a web interface that visualizes the servers in the topology.



Orchestrator: Refactoring

Orchestrator can be used to refactor the topology.

This can be done from the command line tool, via the API or even via the web interface by dragging and dropping.

You can do things like

- Repoint a slave to a new master
- Promote a server to a (co-)master
- Start / Stop slave
- ...

Orchestrator: Recovery

All of these features are nice, but they still require a human to execute them. This doesn't help you much when your master goes down at 3AM and you get paged to resolve this.

Orchestrator can be configured to automatically recover your topology from an outage.

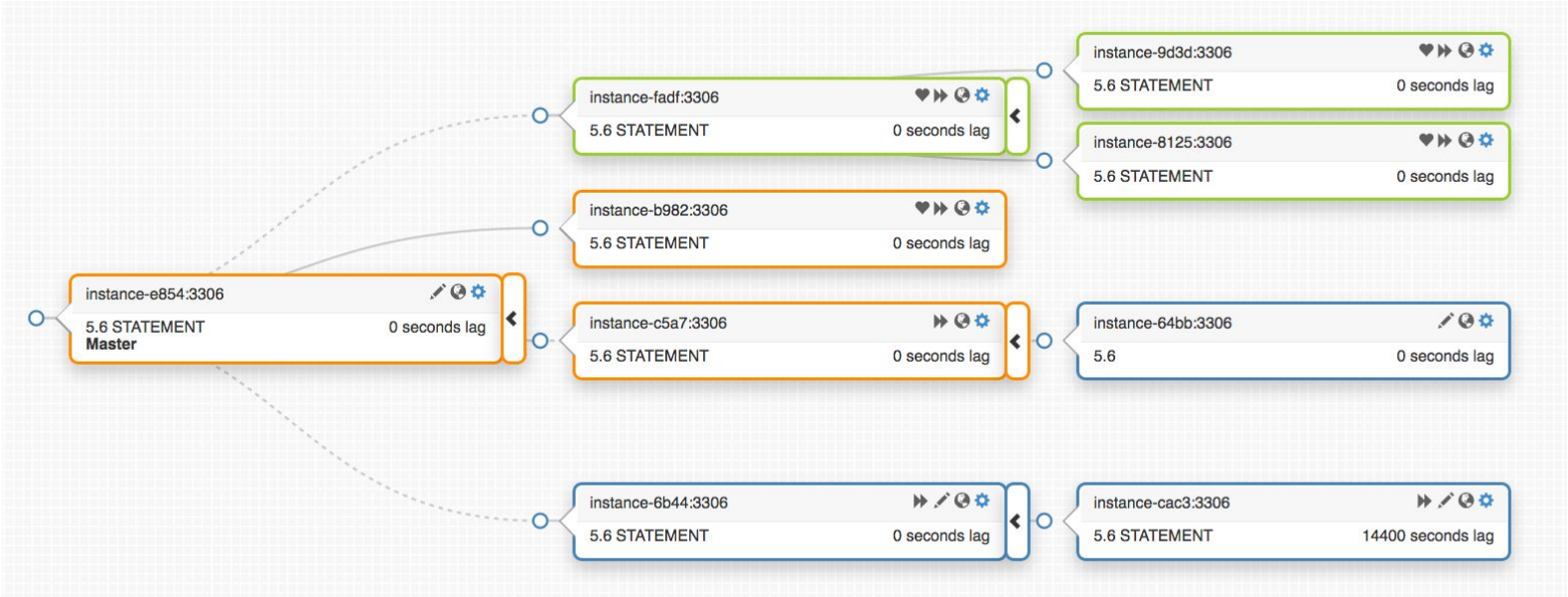
Orchestrator: How recovery works?

To be able to perform a recovery, Orchestrator first needs to detect a failure.

As indicated before Orchestrator connects to every server in the topology and gathers information from each of the instances.

Orchestrator uses this information to make decisions on the best action to take. They call this the holistic approach.

Orchestrator: Failure detection example



Orchestrator High Availability

Orchestrator was written with High Availability as a basic concept.

You can easily run multiple Orchestrator instances with a shared MySQL backend. All instances will collect all information but they will allow only one instance to be the “active node” and to make changes to the topology.

To eliminate a single-point-of-failure in the database backend you can use either master-master replication (2 nodes) or Galera synchronous replication (3 nodes).

Orchestrator High Availability

Since version 3.x of Orchestrator there is “Orchestrator-on-Raft”.

Orchestrator now implements the ‘raft consensus protocol’. This will

- Ensure that a leader node is elected from the available nodes
- Ensure that the leader node has a quorum (majority) at all times
- Allow to run Orchestrator without a shared database backend
- Allow to run without a MySQL backend but use a sqlite backend

Orchestrator High Availability

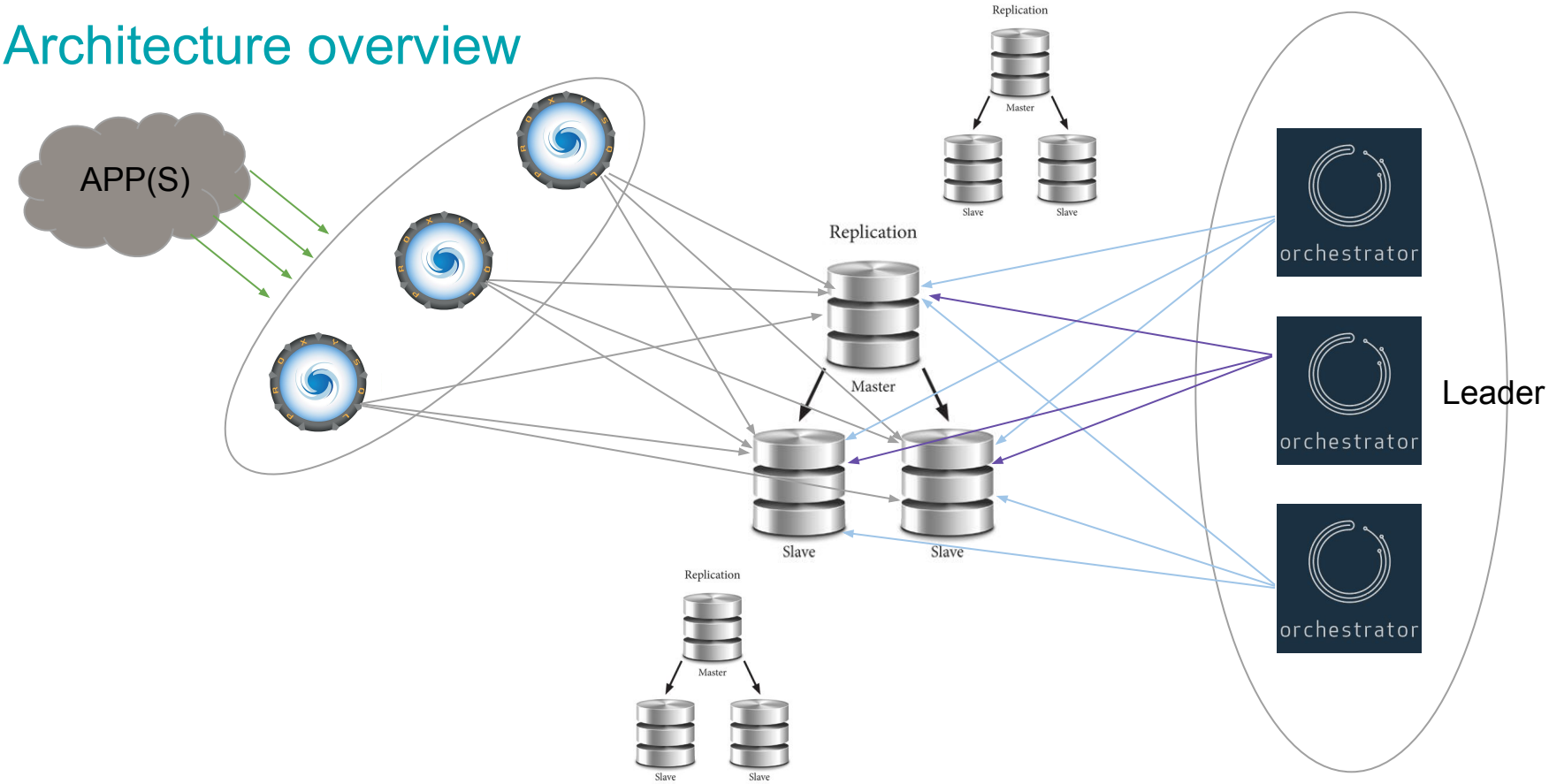
A common example of a High Availability setup

- 3 Orchestrator nodes in different DC's
  - Often one primary DC, one backup DC and one "arbitrator" node in a cloud DC.
  - Orchestrator developers have made changes to raft protocol to allow
    - leader to step down
    - other nodes to yield to a certain node to become the leader
- Shlomi Noach from GitHub will definitely go into more detail on how they implemented this at GitHub.

Questions about Orchestrator?

Overview

Architecture overview



Communication

Default behaviour

ProxySQL read only flag monitoring

- Using the read only flag monitoring in ProxySQL by adding a hostgroup-pair to `mysql_replication_hostgroups` table

```
Admin> SHOW CREATE TABLE mysql_replication_hostgroups\G
*************************** 1. row ***************************
       table: mysql_replication_hostgroups
Create Table: CREATE TABLE mysql_replication_hostgroups (
  writer_hostgroup INT CHECK (writer_hostgroup>=0) NOT NULL PRIMARY KEY,
  reader_hostgroup INT NOT NULL CHECK (reader_hostgroup<>writer_hostgroup AND reader_hostgroup>0),
  comment VARCHAR,
  UNIQUE (reader_hostgroup)
)

1 row in set (0.00 sec)
```

- Requires monitoring user to be configured correctly

Orchestrator ApplyMySQLPromotionAfterMasterFailover

- Orchestrator will flip the read-only flag on master failover
- Setting `ApplyMySQLPromotionAfterMasterFailover`
  - default value was `false` (Orchestrator version < 3.0.12)
  - since 3.0.12 default is `true`
- Recommendation has always been to enable this.
- Configure MySQL to be read-only by default (best practise)

Default behaviour: Caveats

- What happens on network partitions?
  - Orchestrator sees master being unavailable and promotes a new
  - Old master still is writeable (Orchestrator can not reach it to toggle the flag)
  - ProxySQL will move the new master (writable) to the writer hostgroup
  - ProxySQL will place old master as `SHUNNED`.
  - When network partition gets resolved it will still be writable so it will return to `ONLINE`.
  - this will lead to **split brain**

Default behaviour: Caveats / workarounds

- Solutions to prevent this split brain scenario
  - STONITH (shoot the other node in the head)
  - Run script in ProxySQL scheduler that deletes any `SHUNNED` writers from the configuration (both from the writer and reader hostgroups)

Communication

Orchestrator hooks

Orchestrator hooks: What?

- Orchestrator implements hooks on various stages of the recovery process
- These "hooks" are like events that will be called and you can configure your own scripts to run
- This makes Orchestrator highly customisable and scriptable
- Default (naive) configuration will echo text to `/tmp/recovery.log`
- **Use the hooks!** If not for scripting then for alerting / notifying you that something happened

Orchestrator hooks: Why?

- Instead of relying on ProxySQL's monitoring of the read-only flag we can now actively push changes to ProxySQL using the hooks.
- Whenever a planned or unplanned master change takes place we will update the ProxySQL.
  - Pre-failover:
    - Remove `{failedHost}` from the writer hostgroup
  - Post-failover:
    - If the recovery was successful: Insert `{successorHost}` in the writer hostgroup
- WARNING: test test test test test !!!!!
  (before enabling automated failovers in production)

Communication

Decouple communication (between ProxySQL and Orchestrator)

The problem

- Orchestrator hooks are great but...
- ... what happens if there is no communication possible between Orchestrator and ProxySQL?
  - Hooks are only fired once
  - What if ProxySQL is not reachable? Stop failover?
  - You need ProxySQL admin credentials available on Orchestrator

The solution

- Decouple Orchestrator and ProxySQL
- Use Consul as key-value store in between both
- Orchestrator has built-in support to update master coordinates in the K/V store (both for Zookeeper and Consul)
- Configuration settings
  - `"KVClusterMasterPrefix": "mysql/master",`
  - `"ConsulAddress": "127.0.0.1:8500",`
  - `"ZkAddress": "srv-a,srv-b:12181,srv-c",`

Which keys and values?

- `KVClusterMasterPrefix` is the prefix to use for master discovery entries. As example, your cluster alias is `mycluster` and the master host is `some.host-17.com` then you will expect an entry where:
  - The **Key** is `mysql/master/mycluster`
  - The **Value** is `some.host-17.com:3306`
- Additionally following key/values will be available automatically
  - `mysql/master/mycluster/hostname` , value is **some.host-17.com**
  - `mysql/master/mycluster/port` , value is **3306**
  - `mysql/master/mycluster/ipv4` , value is **192.168.0.1**
  - `mysql/master/mycluster/ipv6` , value is **<whatever>**

Avoiding single-point-of-failures (1)

- Recommended setup for Orchestrator is to run 3 nodes with their own local datastore (MySQL or SQLite)
- Communication between nodes happens using the RAFT protocol.
- This is also the preferred setup for the Consul K/V store
- We install Consul "server" on each Orchestrator nodes
- Consul "server" comes also with an "agent"
- We let the Orchestrator leader send it's updates to the local Consul agent.
- Consul agent updates the Consul leader node and the leader distributes the data to all 3 nodes using the RAFT protocol.

Avoiding single-point-of-failures (2)

- We now have our HA for Orchestrator and Consul.
- We have avoided network partitioning
  - Majority vote is required to be the leader on both applications
  - If our local Consul agent is unable to reach the Consul leader node, then Orchestrator will not be able to reach its peers and thus not be the Leader node.
- Optional: Orchestrator extends RAFT to implement a yield option to yield to a specific leader. We could implement a cronjob for Orchestrator to always yield Orchestrator leadership to the Consul leader for faster updates but this not a requirement.

What about the slaves?

- Orchestrator really doesn't care all that much for slaves
- Masters are important for HA
  - the native support for the K/V store ends with updating the masters to it (`"KVClusterMasterPrefix": "mysql/master"`)
- API to the rescue!
  - We can create a fairly simple script that runs in a cron
  - pull ALL the servers from the API (get JSON response)
  - compare the slave entries with values in Consul (for example keys starting with `mysql/slaves`)
  - update Consul if needed

How to configure ProxySQL?

- Now Orchestrator is updating Consul K/V (master via native support, slaves via our script)
- Let's install a Consul "agent" on every ProxySQL machine.
- We can now query Consul data via this local agent

```
root@proxysql-1:~ $ consul members
Node              Address         Status  Type    Build  Protocol  DC       Segment
orchestrator-1    10.0.1.2:8301   alive   server  1.4.3  2         default  <all>
orchestrator-2    10.0.2.2:8301   alive   server  1.4.3  2         default  <all>
orchestrator-3    10.0.3.2:8301   alive   server  1.4.3  2         default  <all>
proxysql-1        10.0.1.3:8301   alive   client  1.4.3  2         default  <default>
proxysql-2        10.0.2.3:8301   alive   client  1.4.3  2         default  <default>
```

How to configure ProxySQL?

- First option is to use the scripted approach
- Run a script in a cronjob or in the ProxySQL scheduler
- Crawl the Consul K/V store
- Update ProxySQL config

Pro	Con
Fairly easy	A lot of wasted CPU cycles
Fairly quick (ProxySQL scheduler works on a millisecond base)	

How to configure ProxySQL?

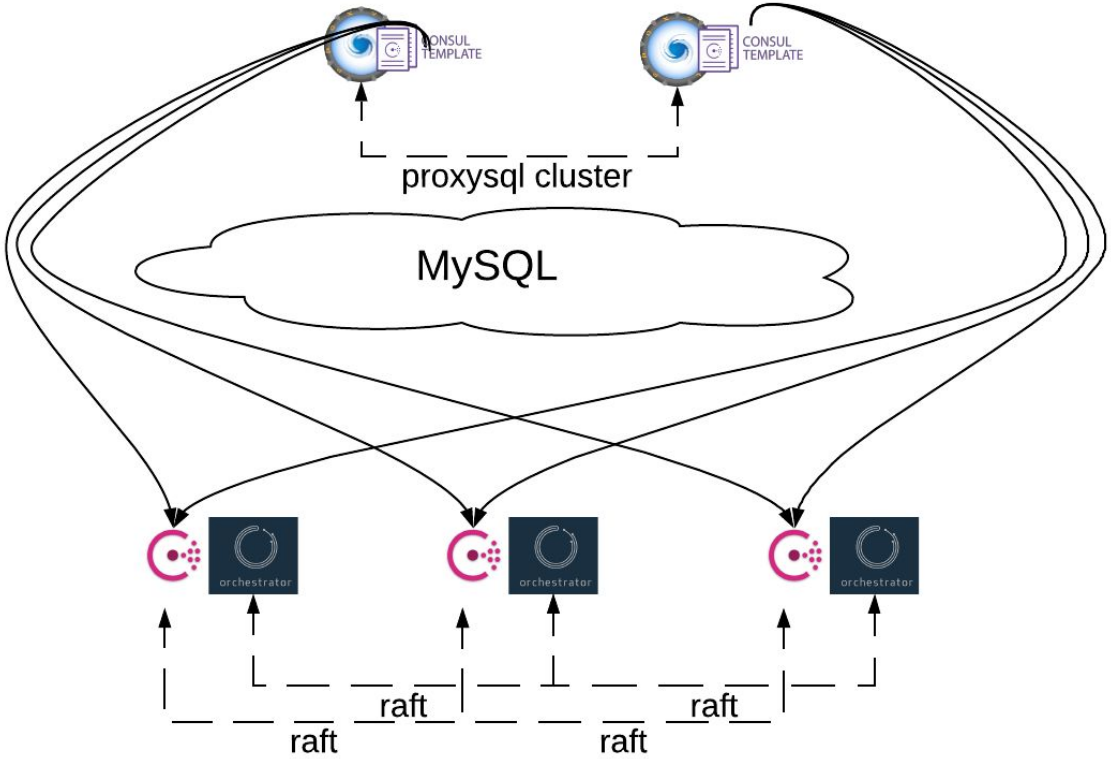
- Use consul-template
- Registers as listener to the Consul values
- Every time a value is changed it will re-generate a file from a template
- Example:

```
{{ if keyExists "mysql/master/testcluster/hostname" }}
DELETE FROM mysql_servers where hostgroup_id = 0;
REPLACE into mysql_servers (hostgroup_id, hostname) values ( 0, "{{ key
"mysql/master/testcluster/hostname" }}" );
{{ end }}

{{ range tree "mysql/slave/testcluster" }}
REPLACE into mysql_servers (hostgroup_id, hostname) values ( 1, "{{ .Key }}{{ .Value }}" );
{{ end }}

LOAD MYSQL SERVERS TO RUNTIME;
SAVE MYSQL SERVERS TO DISK;
```

Architecture



Questions?

We're hiring!!

https://pythian.com/careers

Contact

Matthias Crauwels

crauwels@pythian.com

+1 (613) 565-8696 ext. 1215

Twitter @mcrauwel