Almost Perfect
Service Discovery and Failover
with ProxySQL and Orchestrator

Jean-François Gagné
System and MySQL Expert at HubSpot
jgagne AT hubspot DOT com / @jfg956
MySQL Service Discovery at MessageBird
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Summary of part #1
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

- MySQL Primary High Availability
- Failover to a Replica War Story
- MySQL at MessageBird (Percona Server)
- MySQL Service Discovery at MessageBird (ProxySQL)
- Orchestrator integration and the Failover Process
MySQL Primary High Availability [1 of 4]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Failing-over the primary to a replica is my favorite high availability method
  • But it is not as easy as it sounds, and it is hard to automate well
  • An example of complete failover solution in production:
    https://github.blog/2018-06-20-mysql-high-availability-at-github/

The five considerations for primary high availability:
  • Plan how you are doing primary high availability
  • Decide when you apply your plan (Failure Detection – FD)
  • Tell the application about the primary change (Service Discovery – SD)
  • Protect against the limit of FD and SD for avoiding split-brains (Fencing)
  • Fix your data if something goes wrong
Failure detection (*FD*) is the 1st part (and 1st challenge) of failing-over:
- It is a very hard problem: partial failure, unreliable network, partitions, …
- It is impossible to be 100% sure of a failure, and confidence needs time
  - quick FD is unreliable, relatively reliable FD implies longer downtime
  - Quick FD for short downtime generates false positive

Repointing is the 2nd part of failing-over to a replica:
- Relatively easy with the right tools: GTID, Pseudo-GTID, Binlog Servers, …
- Complexity grows with the number of direct replicas of the primary
- Some software for repointing:
  - Orchestrator, Ripple Binlog Server, Replication Manager, MHA, Cluster Control, MaxScale
MySQL Primary High Availability [3 of 4]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

What do I mean by repointing:
• In below configuration and when the primary fails, once one of the replica as been chosen as the new primary the other replica needs to be re-sourced (re-slaved) to the new primary
Service Discovery (SD) is the 3rd part (and 2nd challenge) of failover:
• If centralized → SPOF; if distributed → impossible to update atomically
• SD will either introduce a bottleneck (including performance limits)
• Or it will be unreliable in some way (pointing to the wrong primary)
• Some ways to implement Service Discovery: DNS, ViP, Proxy, Zookeeper, …

Ø Unreliable FD and unreliable SD is a recipe for split-brains!

Protecting against split-brains (Fencing): Adv. Subject – not many solutions
(proxies and semi-synchronous replication might help)

Fixing your data in case of a split-brain: only you can know how to do this!
tip on this in the war story)
Some infrastructure context:

- Service Discovery is DNS (and failure detector is Orchestrator)
- The databases are behind a firewall in two data centers
Failover War Story [1 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Some infrastructure context:
- Service Discovery is DNS (and failure detector is Orchestrator)
- The databases are behind a firewall in two data centers
- And we have a failure of the firewall in the zone of the primary
Failover War Story [2 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Things went as planned: failed-over from Zone1 to Zone2
  • New primary in zone 2: stop replication, set it read-write, update DNS, …
  • Everything was ok…
Failover War Story [2 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Things went as planned: failed-over from Zone1 to Zone2

• New primary in zone 2: stop replication, set it read-write, update DNS, …
• Everything was ok… until the firewall came back up
Failover War Story [3 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Once the firewall came back up, no detectable problems
  • But some intuition made me checked the binary logs of the old primary
  • And I found new transactions with timestamp after the firewall recovery
    (and obviously this is after the failover to zone 2)
These new transactions are problematic:

- They are in the databases in zone 1, but not in zone 2
- They share common auto-increments with data in zone 2
- Luckily, there are only a few transactions, so easy to fix, but what happened?
The infrastructure is a little more complicated than initially presented:

- There are web servers and local DNS behind the firewalls (fw)
- The DNS update of the failover did not reach zone 1 (because of the fw failure)
- When the firewall came back up, the web servers received traffic and because the DNS was not yet updated, they wrote on the old primary
- Once updated (a few seconds later), writes went to the new primary in zone 2
Failover War Story[6 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

This war story was a decentralized Service Discovery causing problems

Remember that it is not a matter of “if” but “when” things will go wrong

Please share your war stories so we can learn from each-others’ experience

• GitHub has a MySQL public Post-Mortem (great of them to share this):
• I also have another MySQL Primary Failover war story in another talk:
  https://www.usenix.org/conference/srecon19emea/presentation/gagne

Tip for easier data-reconciliation: use UUID instead of auto-increments

• But store UUID in an optimized way (in primary key order)
  https://www.percona.com/blog/2014/12/19/store-uuid-optimized-way/
  http://mysql.rjweb.org/doc.php/uuid
MySQL at MessageBird

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

MessageBird is using MySQL 5.7 (more precisely Percona Server).

These are hosted in many Google Cloud Regions.

There are three types of MySQL deployments:

1. Multi-region primary:
   replicas in many regions and primary potentially on many regions
2. Single-region primary with replicas in many regions
3. Primary and replicas all in a single region
This is a multi-region primary (regions are color-coded)
Requirements for MySQL Service Discovery:
  • Being able to route traffic to local replicas
  • Embed some sort of fencing mechanism

This led to a multi-layer solution using ProxySQL:
  • Three layers for multi-region primary:
    1. Collect
    2. Master-Gateway (mgw)
    3. Fencing

For single-region, mgw and fencing are merged in local-fencing (locfen)
MySQL Service Discovery at MessageBird
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
The *collect* layer is a standard entry-point design.

The *fencing* layer is a natural HA way to route traffic to the primary (a single node would not be HA).

The *master-gateway* layer is the glue between collect and fencing (more about this later in the talk).

The *local-fencing* layer is the *mgw* and the *fencing* layers merged for single-region primary databases because routing to a single region does not need the *mgw* glue (three nodes for N+2 HA, more about this later in the talk).
MySQL Service Discovery at MessageBird
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

The collect is the entry-point of the MySQL Service Discovery:

- It starts with a load-balancer sending traffic to ProxySQL
- We have at least 3 instances of ProxySQL for N+2 high availability

- From here, read-only traffic is sent directly to replicas
- Primary traffic (read-write) is sent to mgw or locfen
Routing from *collect* to *locfen* is either local or crossing a region boundary.

For *mgw*, it is biased to local when a *mgw* is on the same region as *collect*.

- ProxySQL routing is weight-based (no easy fallback routing).
The master-gateway is deployed on all regions potentially hosting a primary

The same way fencing (or locfen) is designed as small as possible to reduce the update-scope of failover (to a single region) …

… mgw bounds the update-scope of moving the primary to another region to a continent (in this case, the three mgw regions are in Europe) (it avoids a Planet-Scale reconfiguration of collect on failover)

… and the way mgw routes traffic to fencing protects against writing to the primary in case of network partitions
The `mgw` routing is as follow:

- If the primary is in a remote region, traffic is routed to *fencing* in that region.
- If the primary is in the same region, traffic is routed to the other `mgw`.
- No path to the primary not crossing a region boundary.
No path to the primary that is not crossing a region boundary
• That might sound sub-optimal, but it is an interesting tradeoff
• It makes the best vs worse case round-trip ratio to the primary closer

Without crossing a region boundary:
• Best case (local access): ~1 ms round-trip to the primary
• Worse case (remote access): ~20 ms round-trip to the primary
  ➢ Ratio of 1 to 20

With crossing a region boundary in mgw:
• Best case (remote access): ~20 ms round-trip to the primary
• Worse case (local access): ~40 ms round-trip to the primary
• Even worse case (remote routed to mgw of the primary) 60 ms
  ➢ Ratio of 1 to 2 (3 in the worse case)
The worse case (which could be avoided with smarter collect routing)
Avoiding low round-trip variance over optimal best-case

- With region-remote primary accesses, a high latency is unavoidable: when having 20 ms latency, having 40 or 60 should not be problematic.

- Moving the average closer to the median avoids problems.

- And in the case where most writes are local, it avoids surprises when the database becomes remote.

- And it prevents writing to the primary in case of a network partition.
And therefore, I claim this design has interesting tradeoff

- But it might not fit everyone’s requirements
Why only two fencing nodes and three locfen nodes:

- I like high availability N+2
- This allows a single failure to not need an immediate fix
- If a locfen node fails on Friday evening, it can wait until Monday

- A failure of one of the two fencing node looks problematic
- But we can failover to another region having two healthy nodes
- And updating two ProxySQL in case of a failover is easier than three (needing three locfen is something I dislike as it makes failover more fragile)
Failing-over is a multi-step process:

1. Detecting a failure
2. **Fencing the primary**: setting it as OFFLINE_HARD in ProxySQL
3. Regrouping replicas under the new primary
4. Waiting for replication to catch-up on the new primary
5. Making the new primary ready: stop replication, set writable, start HB, …
6. **Updating ProxySQL to point to the new primary**
7. Re-configure fencing and master-gateway if needed
Reconfiguring Fencing and MGW [1 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Reconfiguring Fencing and MGW [3 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Reconfiguring Fencing and MGW [4 of 6]

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Reconfiguring Fencing and MGW [5 of 6]
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Reacting to a network partition is a similar operation
Orchestrator integration
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Orchestrator integration

• Pre-failover hook: fence the primary in fencing (or locfen)
• Post-failover hook: update fencing (or locfen) to the new primary

• List of ProxySQL nodes (fencing or locfen) and the ProxySQL hostgroup are store in each databases which make this information available to the Orchestrator hooks
MySQL Service Discovery at MessageBird
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Almost Perfect Service Discovery and Failover with ProxySQL and Orchestrator

Art van Scheppingen
Senior Database Engineer at MessageBird
art.vanscheppingen AT messagebird DOT com
How does it work far?
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Has been successfully in production for over two years now
  • Most of our workload is on single-region primary (e.g. locfen)
  • We have one cluster on multi-region primary

ProxySQL has been very stable for us
  • No big issues on 1.4.x, 2.0.x and 2.1.x
How does it work far?
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Easier for devs to setup connections to primary
• Point connection to ip address
• No failover handling necessary

Easier for devs to scale out reads
• Point connection to the same ip address
• Uses a different user for RO
• We can differentiate reads (read-only, read-only replica-only, etc)
Multiplexing in ProxySQL turned out to be tricky

Without Multiplexing:
- 1 on 1 number of connections between Collect and Locfen
- Application makes a connection for each shard (8 at the moment)
- Application only uses one actively
- High number of connections means high load

With multiplexing
- High number of connections on Collect
- Theoretically only 1/8th of the connections on Locfen
- Even lower due to less overhead on establishing connections
Mastering multiplexing [2 of 4]

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Prior to multiplexing
We enabled multiplexing in our configuration but not much happened

ProxySQL disabled multiplexing due to an auto increment
- Bugreport on ORM that didn’t handle multiplexing well (Hibernate)
- Parsing of the OK packet for auto increments
- Whenever encountered: multiplexing is affected
- mysql-auto_increment_delay_delay_multiplex set to 5 by default
- This means for 5 consecutive queries multiplexing is disabled
After mastering multiplexing

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
We tried reusing collect and locfen as much as possible
- Centralized configuration (hostgroups and users)
- Less complexity

Expansion was inevitable
- Noisy neighbors (hostgroups)
- Reducing risk
- Better tuning for certain workloads
- Easier for maintenance
- Cascading effect to other hostgroups

Currently running 3 vertical stacks of collect and locfen
Above 50% CPU usage ProxySQL will show increased latency
  • Lower CPU usage by multiplexing
  • Lower CPU usage by idle-threads

When a certain hostgroup in Locfen is under stress (1000x normal workload)
  • CPU usage can get above 50%
  • Latency will increase on all hostgroups
  • Latency will cascade upstream to the collect layer
Separation of stacks: Noisy neighbors
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Above 50% CPU usage ProxySQL will show increased latency
For us ProxySQL scales up to about 12K of connections per host
  • After this we will hit the limits of TCP

Our Collect layer reached 7.8K connections
  • If one collect host fails two remain
  • Two remaining hosts will have to do an additional 3.9K connections
  • Very close to our 12K limit
  • Replacing a failed host now becomes an emergency operation
ProxySQL keeps count of connection errors (MySQL + TCP shared)
- Will shun a host if a backend becomes “less responsive” (e.g. high load)
- Happens on hostgroups with any number of hosts

Hitting the limits of TCP
- Errors to locfen or MGW will increase
- Locfen and MGW backends will be shunned
- Established connections will also be closed
Separation of stacks: Reducing risk

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Separation of stacks: Reducing risk

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
ProxySQL keeps count of connection errors (MySQL + TCP shared)

- Will shun a host if a backend becomes “less responsive” (e.g. high load)
- Happens on hostgroups with any number of hosts

Shunning a primary for 1 second will cause another torrent of connections

- Client gets a timeout and will reconnect immediately
- No available backend: new connection is “paused” up to 10 seconds
- After 1 second primary become available again
- ProxySQL has thousands of connections waiting
- Rinse and repeat…
How to detect shunned hosts?

- ProxySQL will log a shunned host in the proxysql log
- This includes server name, error rate and duration of shun.

How do we get them in our graphs?

- ProxySQL log tailer
- Looks for: connection timeouts, shunned hosts, shunned due to replication lag
- Exports metrics to Prometheus every minute
Shunning a primary

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Shunning a primary for 1 second will cause an avalanche connections

- Normal latency is 10ms to 50ms
- Added latency of 1 second will decrease application throughput
- Decreased application throughput means k8s scale up workers
- k8s scale up means more incoming connections
- Rinse and repeat…

How we dealt with this:

- During some incidents we throttle down workers
- Counter intuitive: throttling down works increases throughput
- Some application workers now have a fixed ceiling
Most ProxySQL tuning is done on a global level

Some examples:

- mysql-connection_delay_multiplex_ms
- mysql-free_connections_pct
- mysql-wait_timeout

Having a separate stack allows

- Fine tuned multiplexing configuration
- Earlier or later closing of connections
- Separate handling of (end) user connections
Separation of stacks: maintenance
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Maintenance on Collect is scary
  • Draining from GLB gracefully is "closing after X-minutes"
  • Near capacity means we run a risk when performing maintenance

Maintenance on MGW/Fencing/Locfen is scary
  • Draining a host takes ages to happen
  • Aggressive reuse of connections by connection pool
  • Connection timeout (wait_timeout) is 8 hours
  • Some applications don’t handle closing of a connection well
Warstory: instability on one cluster swiped out many others

The instable cluster
  • MySQL back_log set too low
  • TCP listen overflows on primary
  • ProxySQL started to shun primary

The effect
  • Continuous shunning happened
  • TCP listen overflows started to happen on ProxySQL
  • Affected stability on other hostgroups
Shunning of primary causes endless shunning loop

(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)
Listen overflows on ProxySQL hosts make it even worse
In effect Collect layer shuns Locfen layer hosts
Other quirks: Connection contamination
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

After a client-connection closes, the connection will be reused
  • Collect → Locfen
  • Locfen → database

ProxySQL resets connection to initial connection parameters

What if new connection doesn’t match settings (e.g. UTF8mb4 or CET timezone?)

Will the connection between Locfen → database also change?
Other quirks: Uneven distribution
(Service Discovery and Failover with ProxySQL and Orchestrator – PL May 2021)

Uneven distribution of connections/queries
- Weight influences the distribution of connections
- Reuse of existing connections is favored by ProxySQL
- Influenced by `mysql-free_connections_pct`

The variable `mysql-free_connections_pct` is a global variable
- Percentage of maximum allowed connections of a hostgroup
- Some hostgroups allow up to 3000 incoming connections
- 2% of 3000 is 60 connections, actual usage is 10 to 15
- More connections are kept open in connection pool than necessary
Thanks !

Almost Perfect Service Discovery and Failover with ProxySQL and Orchestrator

*by* Jean-François Gagné

*and* Art van Scheppingen

Presented at Percona Live Online, May 2021