MySQL High Availability in the Database On Demand service

Abel Cabezas Alonso Project Associate at CERN

Date 12 October 2020

Presentation content

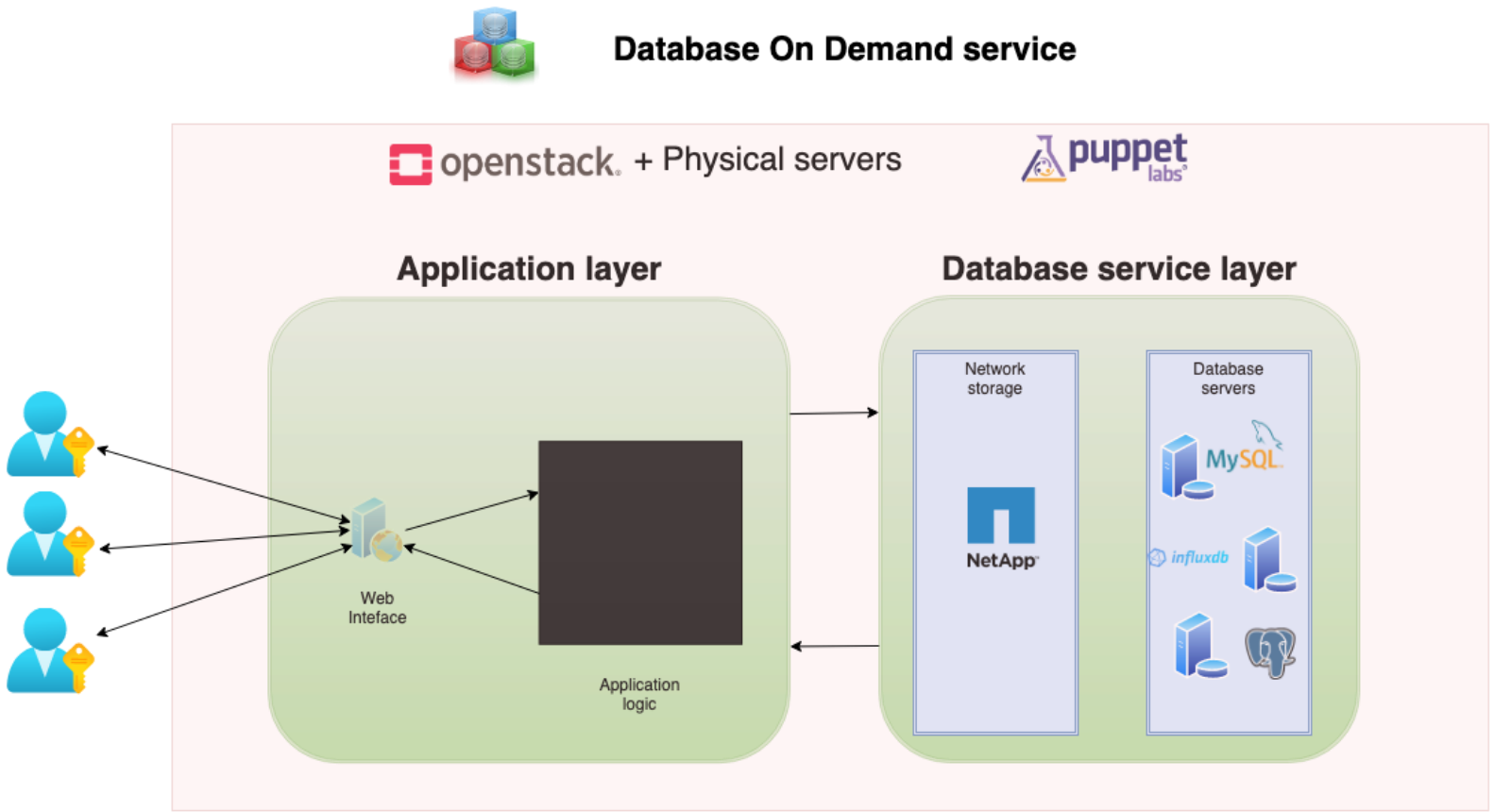
- 1. Brief introduction to the Database On Demand service.
- 2. Why High Availability?
- 3. High Availability solutions in the MySQL ecosystem.
- 4. ProxySQL + MySQL Primary/Secondary simple replication.

1. Brief introduction to the DBOD Service.

What is the DBOD Service?

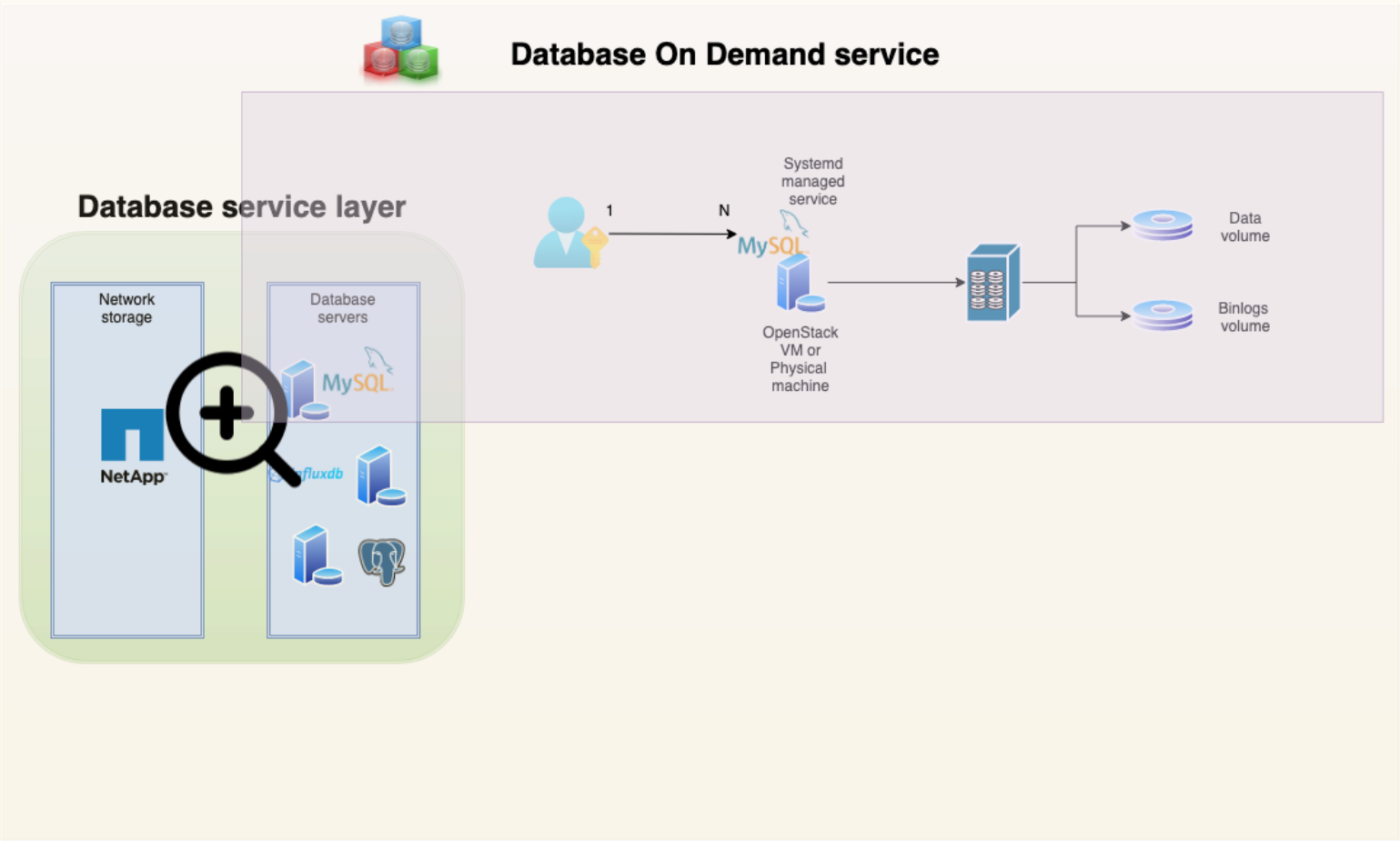
- **Born in 2012 was originally conceived as DBaaS (database as a service) with the goal of centralising and standardising procedures for the existing MySQL and PostgreSQL databases within CERN.**
- **Provide a free open source alternative to the central Oracle-based database service.**
- **The DBOD empowers users to perform certain actions that are traditionally done by DBAs granting them full DBA privileges.**
- **Any user with a CERN user account can request a DBOD instance.**

Database on Demand service architecture



- The user interacts directly with his instance as she/he would do in a regular installation.
- Web interface for managing start/stop, backups, restoring, monitoring ,configuration files, logs…
- Puppet managed nodes.
- Database servers running on CERN CentOS 7.
- Clustered storage network provided by NetApp.

Database on Demand service architecture.



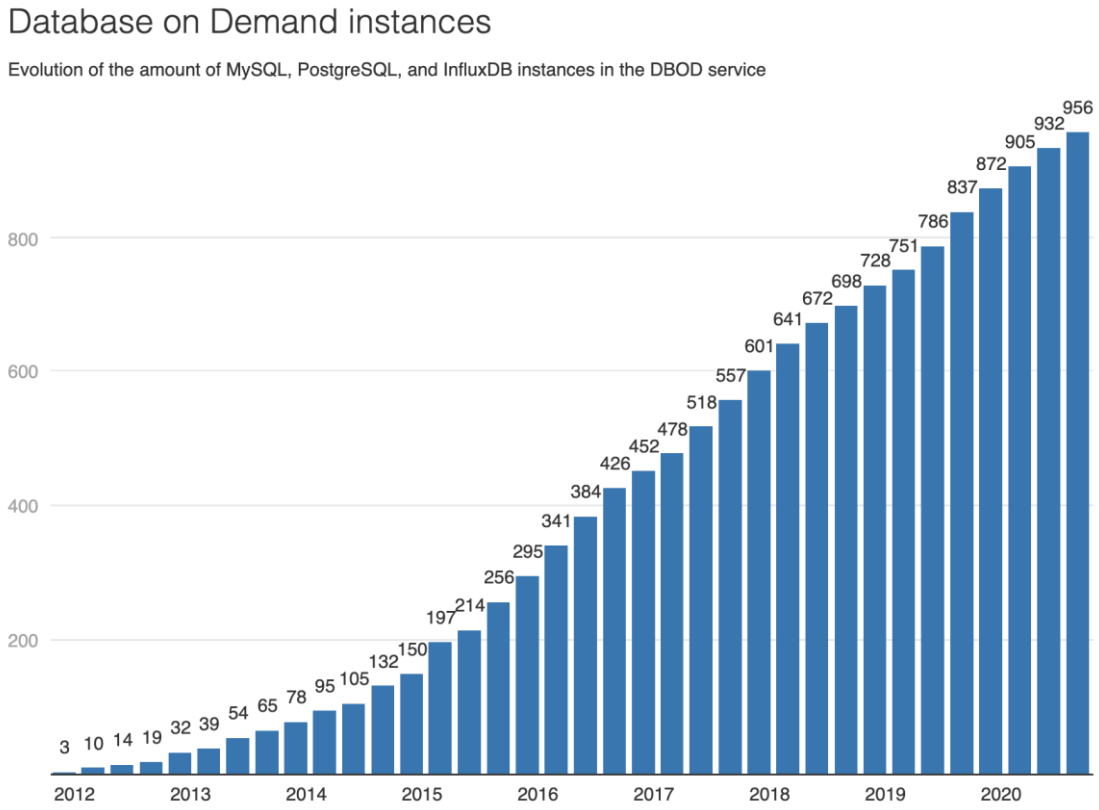
- One user can own from 1 to N instances.
- An instance can be owned by only one person but managed by all members of the designated project.
- Instances run on premise physical or cloud hosts.
- A server can host several instances and be run on 6 different Availability Zones.
- Each instance's storage is split into two separate volumes.

What we provide:

- **Automated backup and recovery services.**
- **Monitoring of instance, server and storage metrics.**
- **Cloning .**
- **Replication.**
- **Three different DBMS : MySQL, PostgreSQL and InfluxDB.**
- **Database upgrades.**
- **Warranty of service continuity.**
- **And now… High Availability (for MySQL by now).**

The DBOD in numbers

- More than 1000 instances as of October 2020.
  - 600 MySQL
  - 267 PostgreSQL
  - 152 InfluxDB
- 155 managed Virtual + Physical hosts.
- Around 36 TB of data managed.
- Around 800 users subscribed to the service.
- 8 years of operational experience.



2. Why High Availability?

What is High Availability?

- The capacity of a system to offer operational continuity during a given period of time.
- Three main principles to achieve High Availability:
  - Redundancy: if a component fails you have to have a replacement for it (no single point of failure).
  - Contingency plans: or what to do if a component fails.
  - Procedure: if a component fails you have to be able to detect it and then execute your plans.

- Reference: Charles Bell, Mats Kindahl & Lars Thalmann, (2010) MySQL High Availability: Tools for building robust data centers, O'Reilly, 1st edition.

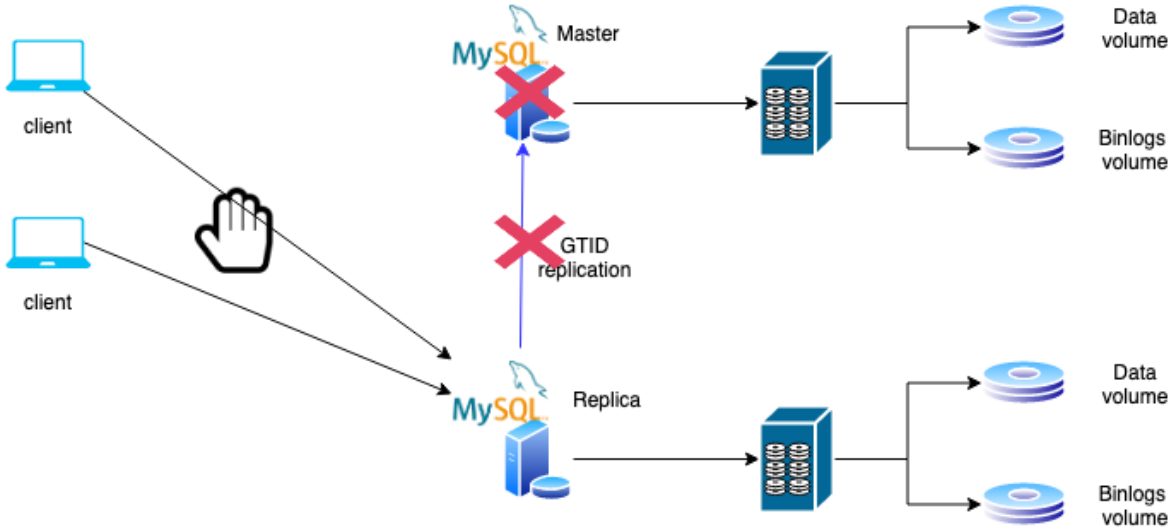
Why do we need High Availability in the DBOD service?

- **Network interventions affecting routers or switches.**
- **Storage failures.**
- **Database corruption.**
- **Host resources exhausted.**
- **Problems with the hypervisor.**
- **Regular service operations.**
  - OS upgrade
  - Scheduled database server upgrades.
  - Scheduled Storage migrations.
- **Not the rule, but many things can go wrong…**

Precedents

Master/Replica with manual switchover/failover.

- Advantages
  - In case of planned interventions, users can prepare in advance for the outage.
  - Users can still operate with the promoted replica during the major planned interventions.
- Disadvantages:
  - Requires manual intervention.
  - Requires planning.
  - No HA solution.
  - Users have to manually update the connection parameters in their code.
  - Unmanageable when the amount of replication sets is big.
  - No contingency plan for unexpected failures.
  - Failover times depend on the operator's availability and its ability to perform the failover.
- **We need to automate.**



Our requirements

- **Easily adaptable to our infrastructure.**
- **Easily deployable.**
- **Architecture easy to use and fix.**
- **Low impact in terms of resource consumption.**
- **Single point of connection for the user.**
- **Automatic failover (a replica has to automatically take over its master and start accepting writes).**
- **Prime consistency over availability (better late than nothing).**
- **Free Open Source.**

3. High availability solutions in the MySQL ecosystem.

MySQL Simple replication + MariaDB MaxScale Proxy

MariaDB MaxScale

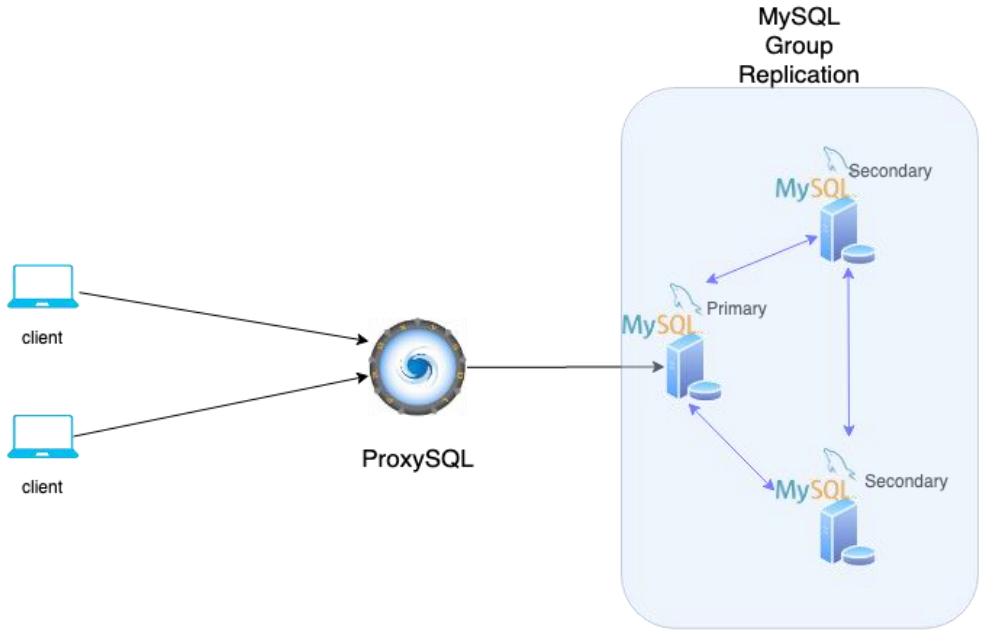
- Database proxy that forwards database statements to one or more MySQL or MariaDB database servers.
- Forwarding is performed using rules based on the semantic and the roles of the servers within the backend cluster of databases.
- Built-in automatic failover.
- Tested as a PoC.
- Problem:
  - Licensing.
  - BSL 1.0 licensed (not free).
  - Max amount of instances without license < 3.

MySQL Group Replication with ProxySQL

Group replication

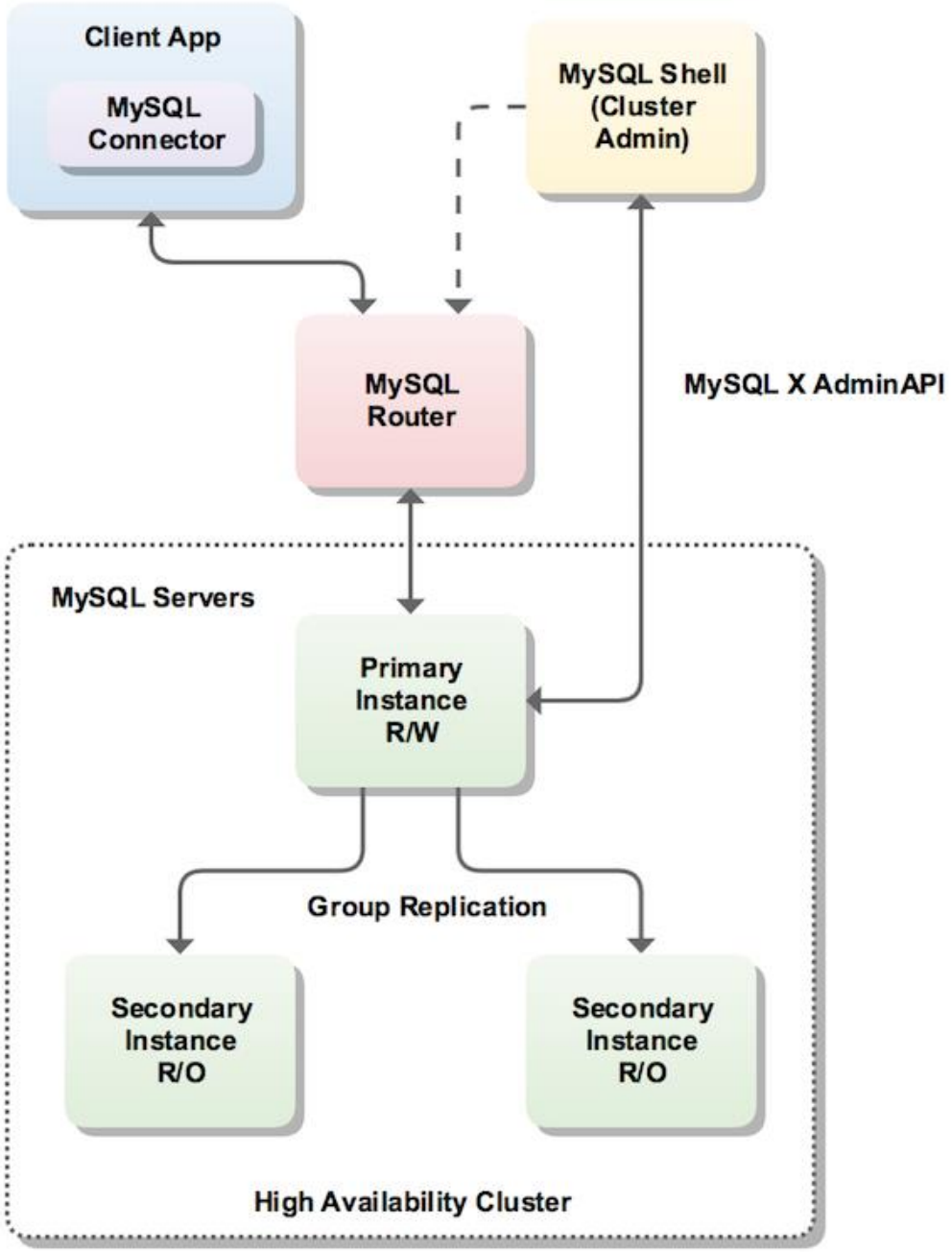
- Provided as a plugin to MySQL server.
- Distributed database.
- Automatic failover.
- Minimum number of instances to achieve consensus = 3.
- Each transaction is executed or rolled back based on the consensus of all members.
- Tested as a PoC.

- Problem:
  - We hit several bugs while testing.
  - Cluster left in inconsistent state without writable members.
  - Hard to repair.
  - Further testing envisaged.



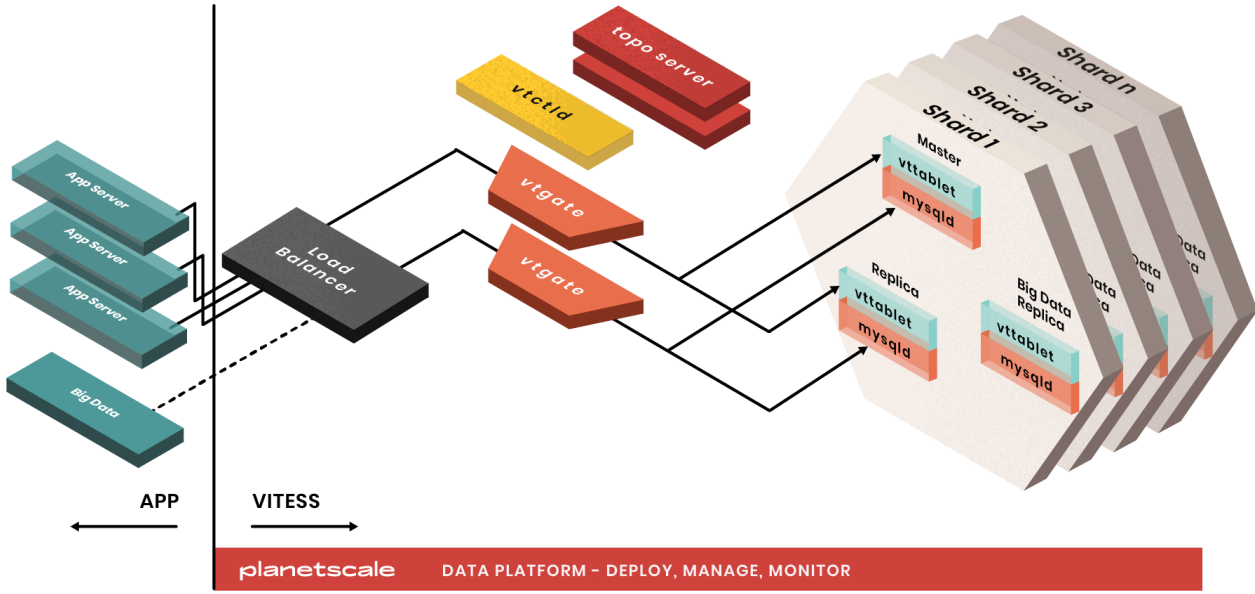
MySQL InnoDB Cluster.

- MySQL Group replication with extended functionality.
- Built-in failover.
- MySQL Shell: for easily managing and setting up the cluster.
- MySQL router as a MySQL native proxy (has no affinity with the instances of the cluster).
- Uses InnoDB storage engine (like all of our instances).
- Uses the same binaries as MySQL server.
- Problem:
  - Uses group replication.
  - Further testing envisaged.



Vitess

- Vitess is a solution that responds to the problem of scalability more than a solution for HA, although HA can be achieved integrating it with Orchestrator.
- Its use can be considered for instances bigger than 500 GB (not our case).
- A highly available Vitess Cluster requires (at least) the following components:
  - 2 VTGate Servers (Proxy).
  - A redundant topology service (e.g. 3 etcd servers).
  - 3 MySQL servers with semi-sync replication enabled.
  - 3 VTTablet process (Process attached to each MySQL instance that determines its role in the topology).
  - 1 VTctld process.



Problem

- Very complex for our use cases.
- Not all statements compatible with MySQL.

Other options discarded because of incompatibilities with our infrastructure.

- **Percona XtraDB cluster:**
  - Different storage engine (XtraDB): compatible but not the same, it may diverge over time.
  - Requires extra binaries for the deployment.
- **MySQL NDB cluster:**
  - Uses different binaries than MySQL server.
  - Use of a different storage engine (NDB instead of InnoDB) which requires a conversion.

4. ProxySQL + MySQL Primary/Secondary replication.

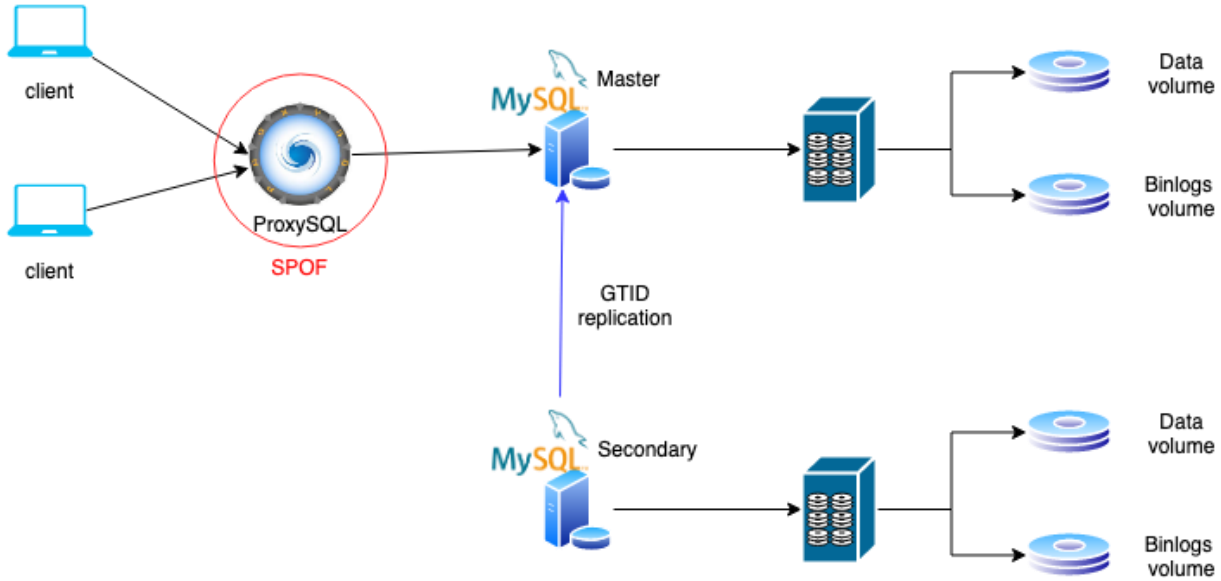
ProxySQL

- High performance, high availability, protocol aware proxy for MySQL and its forks.
- Takes decisions on where to route traffic based on the monitored value of the MySQL “read-only” / “super-read-only” variable.
  - If read-only false => We assume it is a Primary (write to it).
  - If read-only true => We assume it is a Secondary (read only).
- It is designed to not perform any specialised operation in relation to the servers it is connected to.
- **No bulit-in failover**.
- ProxySQL does not know about the topology it is connected, therefore changes in the topology have to be updated with custom scripts.
- Built-in **monitoring module** for checking the status of the connected instances.
- **Scheduler module**: Offers the user the possibility to write custom scripts that are run by ProxySQL in a cron-like fashion.

First approach

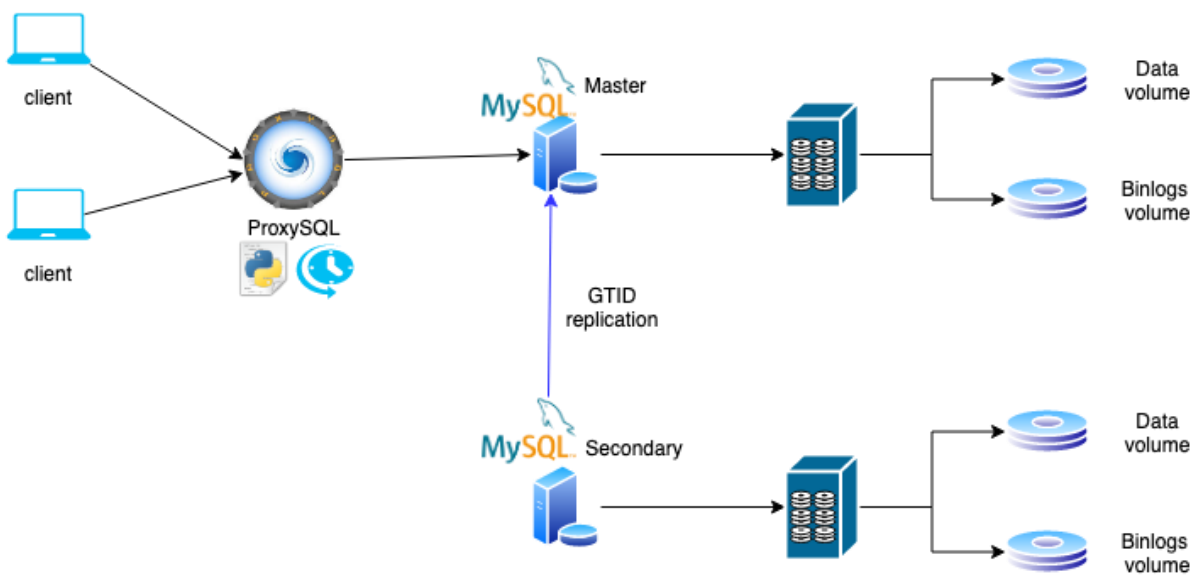
- Deploy PoxySQL as a systemd service.
- Simply routes traffic.

- Problems:
  - How to detect if a component fails?
  - Single Point of Failure.
  - No redundancy.
  - No contingency plan or procedure if a component fails.



How to detect if a node fails?

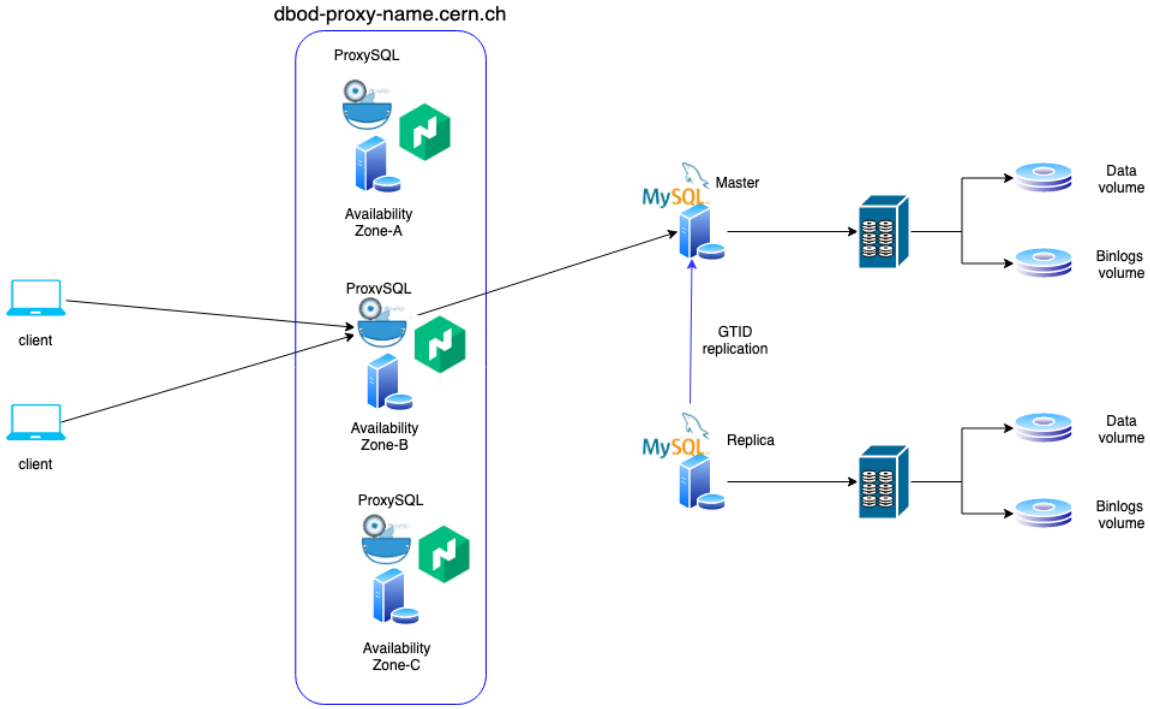
- Profiting from the built-in **scheduler** and **monitoring** modules.
- ProxySQL monitors de instances at fixed intervals of time and stores the result of the health check inside its internal SQLite3 database.
- Custom python scripts check on the status of the nodes by looking at the monitoring table.
  - If the replica is down => do not allow failover to happen and alert (prometheus).
  - If the Primary is down => wait 5 heartbeats or 25 seconds to avoid false positives and then perform the failover.
- Replicate the users that can access the backends through the proxy.



Adding redundancy

- Run ProxySQL as a docker container.
- What happens if it fails?
  - Run it on Nomad (container orchestrator similar to Kubernetes) with nodes in three different availability zones.

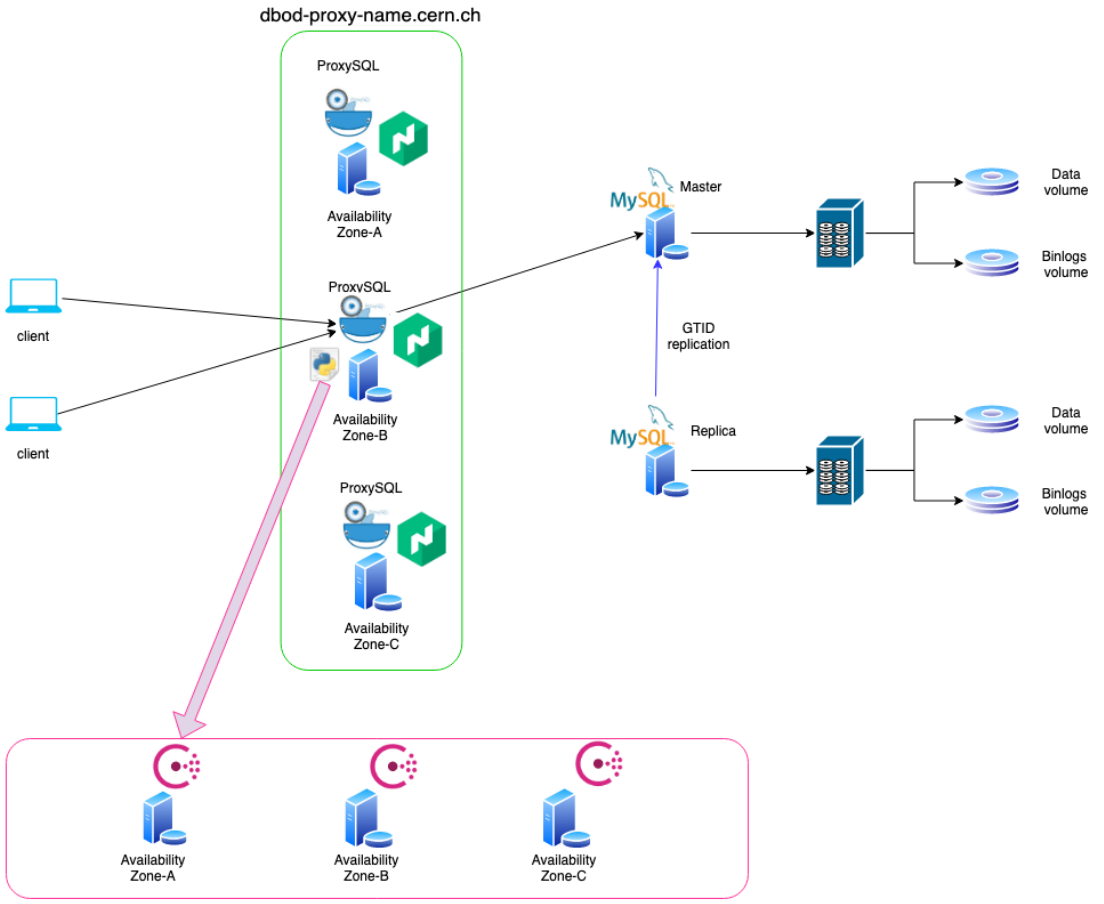
- ProxySQL cluster mode is not really usable, since it does not implement consensus algorithms between the nodes (on the road map of ProxySQL creators). Who makes the decisions?
- Solution:
  - Run a single instance and be ready to run a new instance in any of the three availability nodes => nomad provides it.
  - Respawn time of a container on any node of the cluster ≈ 5 s .



- New problem:
  - Which is our source of truth?
  - If the new container is started in a different node how does it know about the previous topology ?

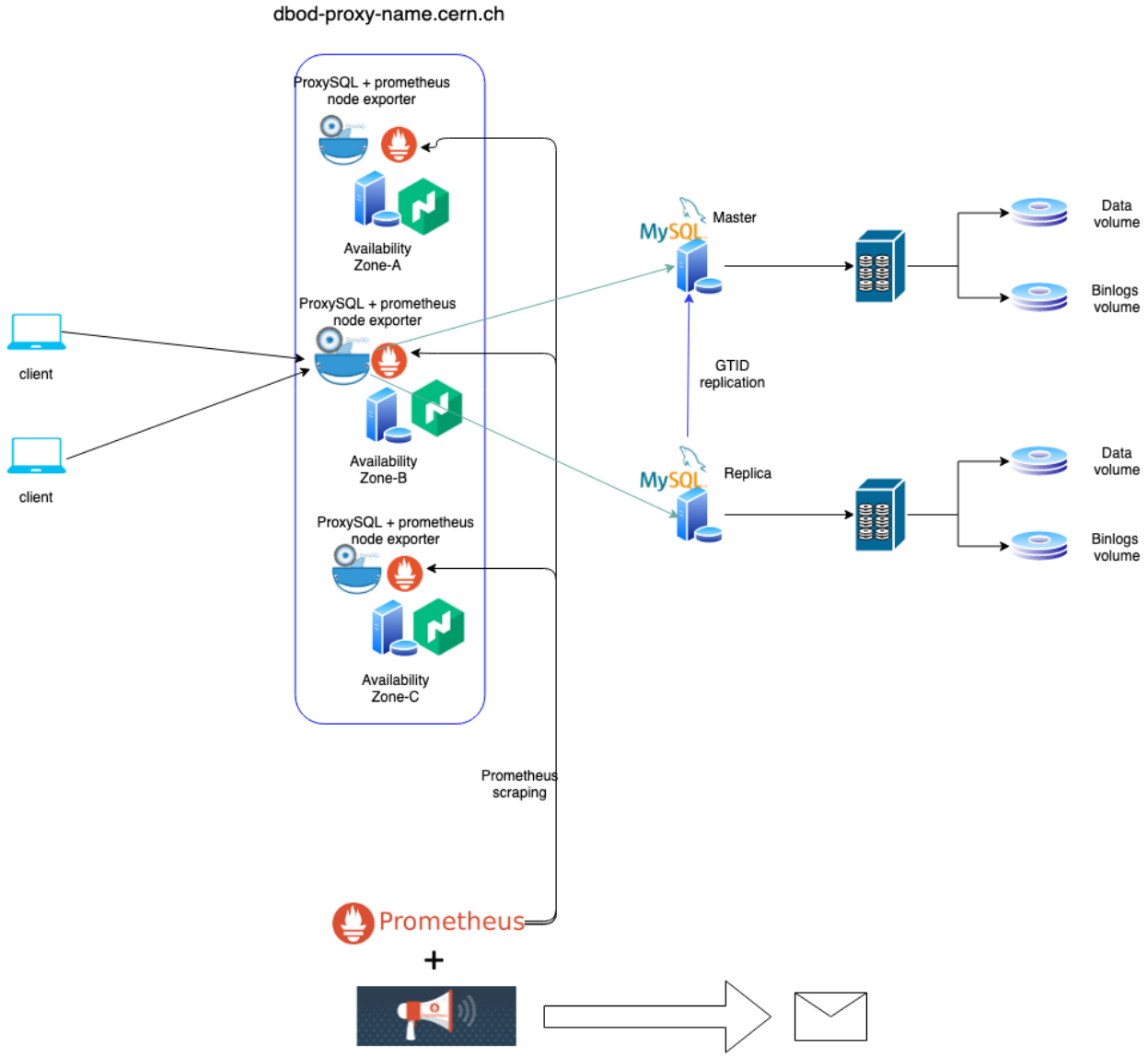
Adding Consul

- If the new container is started in a different node how does it know about the previous topology ?
- Consul:
  - K-V high available cluster that keeps information of the topology.
  - If the ProxySQL is restarted, it will first look at the legitimate roles registered in Consul.
- More problems can happen:
  - The primary may go down while the proxy was down.
  - If an old master is restarted having two instances with each “read-only” value set to false, who is the legitimate master?
- Solution:
  - More python scripts in the scheduler module, to resolve the conflicts:
    - Two primary conflict.
    - Two secondary conflict.
    - No primary on the topology.



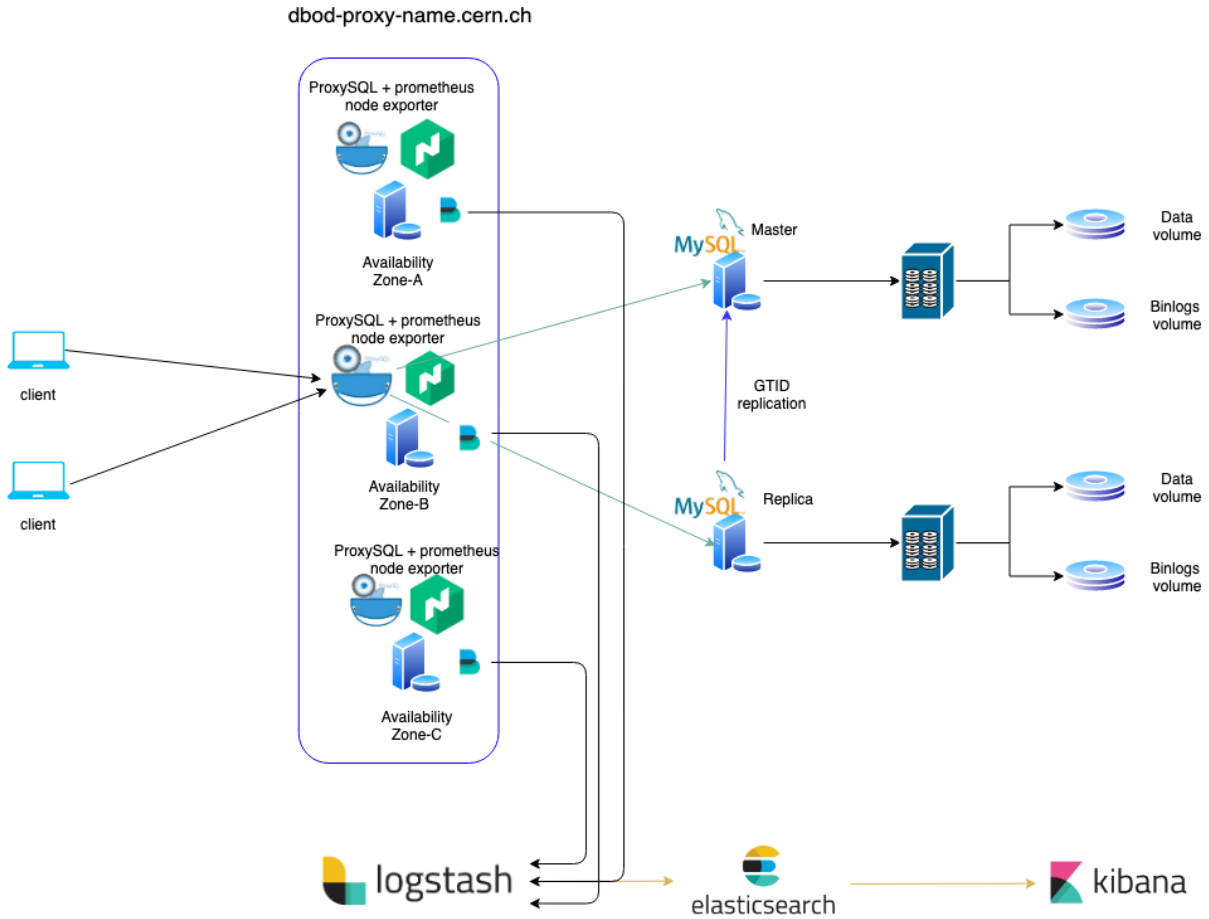
Alerting with Prometheus

- Each ProxySQL container runs with a prometheus node exporter sidecar container.
- Targets (ProxySQL node exporters) are registered in prometheus server.
- Prometheus server scrapes metrics at fixed intervals of time.
- Prometheus evaluates its alert rules and if any is triggered it passes it to the alert manager.
- Notifications by email.
- Different types of alerts:
  - Container not running.
  - ProxySQL down (container running but ProxySQL main process down).
  - No Client connections
  - Replication Lag > X s
  - No Primary running.
  - No Secondary running.



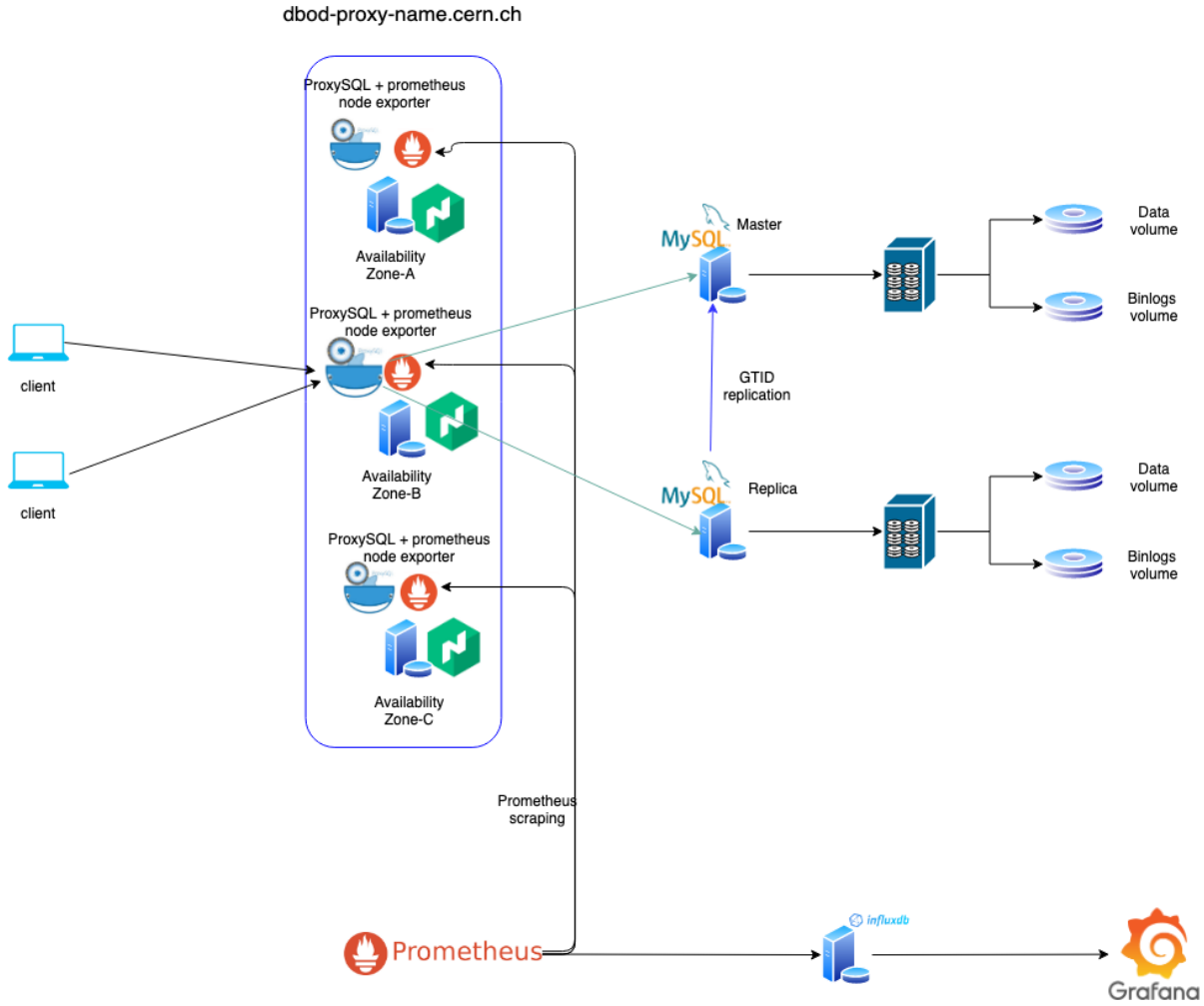
Logging

- ELK Stack.
  - Filebeat + Logstash + elasticsearch + Kibana.



Monitoring

- The same prometheus node exporters that we use for alerting.
- Prometheus server to scrape the metrics.
- Prometheus sends metrics to a DBOD InfluxDB instance.
- Metrics are visualised in Grafana.



Results

- Two CERN critical services already using the architecture.
- Downtime during failover: 60 - 90 seconds although it can be reduced using Traefik for the DNS resolution.
- Evolving architecture to improve its quality.
- HA achieved:
  - Redundancy at all levels.
  - Contingency plans defined.
  - Procedure: monitoring (detect) + python scripts to execute the plans.

Road Map

- **Implement graceful failover.**
- **Measure the availability of the solution in number of nines (currently expected to provide from 1 to 3 nines). Also MTBF, MTTR.**
- **Add redundancy ProxySQL level. Either by taking the decision making out of ProxySQL (i.e. Orchestrator) or implementing the consensus algorithms ourselves.**
- **Integrate with Orchestrator for better detection of false positives and true positives.**
- **Test again InnoDB Cluster to see if the technology is mature enough.**

Thank you for your attention. Questions?