Back to GitHub.com

The GitHub Blog

June 20, 2018

Engineering

MySQL High Availability at GitHub

Shlomi Noach

GitHub uses MySQL as its main datastore for all things non- `git` , and its availability is critical to GitHub's operation. The site itself, GitHub's API, authentication and more, all require database access. We run multiple MySQL clusters serving our different services and tasks. Our clusters use classic master-replicas setup, where a single node in a cluster (the *master*) is able to accept writes. The rest of the cluster nodes (the *replicas*) asynchronously replay changes from the master and serve our read traffic.

The availability of master nodes is particularly critical. With no master, a cluster cannot accept writes: any writes that need to be persisted cannot be persisted. Any incoming changes such as commits, issues, user creation, reviews, new repositories, etc., would fail.

To support writes we clearly need to have an available writer node, a master of a cluster. But just as important, we need to be able to identify, or *discover*, that node.

On a failure, say a master box crash scenario, we must ensure the existence of a new master, as well as be able to quickly advertise its identity. The time it takes to detect a failure, run the failover and advertise the new master's identity makes up the total outage time.

This post illustrates GitHub's MySQL high availability and master service discovery solution, which allows us to reliably run a cross-data-center operation, be tolerant of data center isolation, and achieve short outage times on a failure.

High availability objectives

The solution described in this post iterates on, and improves, previous high availability (HA) solutions implemented at GitHub. As we scale, our MySQL HA strategy must adapt to changes. We wish to have similar HA strategies for our MySQL and for other services within GitHub.

When considering high availability and service discovery, some questions can guide your path into an appropriate solution. An incomplete list may include:

- How much outage time can you tolerate?
- How reliable is crash detection? Can you tolerate false positives (premature failovers)?
- How reliable is failover? Where can it fail?
- How well does the solution work cross-data-center? On low and high latency networks?
- Will the solution tolerate a complete data center (DC) failure or network isolation?
- What mechanism, if any, prevents or mitigates split-brain scenarios (two servers claiming to be the master of a given cluster, both independently and unknowingly to each other accepting writes)?
- Can you afford data loss? To what extent?

To illustrate some of the above, let's first consider our previous HA iteration, and why we changed it.

Moving away from VIP and DNS based discovery

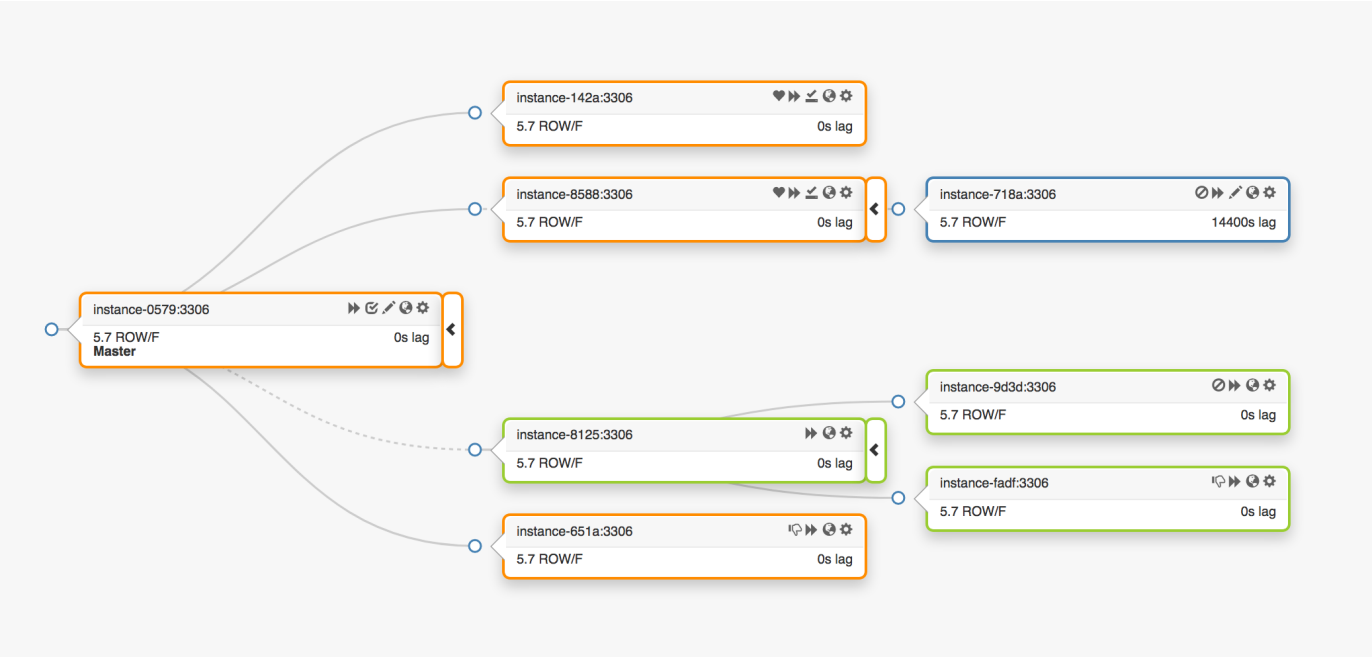
In our previous iteration, we used:

- orchestrator for detection and failover, and
- VIP and DNS for master discovery.

In that iteration, clients discovered the writer node by using a name, e.g. `mysql-writer-1.github.net`. The name resolved to a Virtual IP address (VIP) which the master host would acquire.

Thus, on a normal day, clients would just resolve the name, connect to the resolved IP, and find the master listening on the other side.

Consider this replication topology, spanning three different data centers:



In the event of a master failure, a new server, one of the replicas, must be promoted in its place.

`orchestrator` will detect a failure, promote a new master, and then act to reassign the name/VIP. Clients don't actually know the identity of the master: all they have is a *name*, and that name must now resolve to the new master. However, consider:

VIPs are cooperative: they are claimed and owned by the database servers themselves. To acquire or release a VIP, a server must send an ARP request. The server owning the VIP must first release it before the newly promoted master acquires it. This has some undesired effects:

- An orderly failover operation will first contact the dead master and request that it release the VIP, and then contact the newly promoted master and request that it grab the VIP. What if the old master cannot be reached or refuses to release the VIP? Given that there's a failure scenario on that server in the first place, it is not unlikely that it would fail to respond in a timely manner, or indeed respond at all.
  - We can end up with a split-brain: two hosts claiming to have the same VIP. Different clients may connect to either of those servers, depending on the shortest network path.
  - The source of truth here depends on the cooperation of two independent servers, and this setup is unreliable.
- Even if the old master does cooperate, the workflow wastes precious time: the switch to the new master waits while we contact the old master.
- And even as the VIP changes, existing client connections are not guaranteed to disconnect from the old server, and we may still experience a split-brain.

In parts of our setup VIPs are bound by physical location. They are owned by a switch or a router. Thus, we can only reassign the VIPs onto co-located servers. In particular, in some cases we cannot assign the VIP to a server promoted in a different data center, and must make a DNS change.

- DNS changes take longer to propagate. Clients cache DNS names for a preconfigured time. A cross-DC failover implies more outage time: it will take more time to make all clients aware of the identity of the new master.

These limitations alone were enough to push us in search of a new solution, but for even more consideration were:

- Masters were self-injecting themselves with heartbeats via the `pt-heartbeat` service, for the purpose of lag measurement and throttling control. The service had to be kicked off on the newly promoted master. If possible, the service would be shut down on the old master.
- Likewise, Pseudo-GTID injection was self-managed by the masters. It would need to kick off on the new master, and preferably stop on the old master.
- The new master was set as writable. The old master was to be set as `read_only`, if possible.

These extra steps were a contributing factor to the total outage time and introduced their own failures and friction.

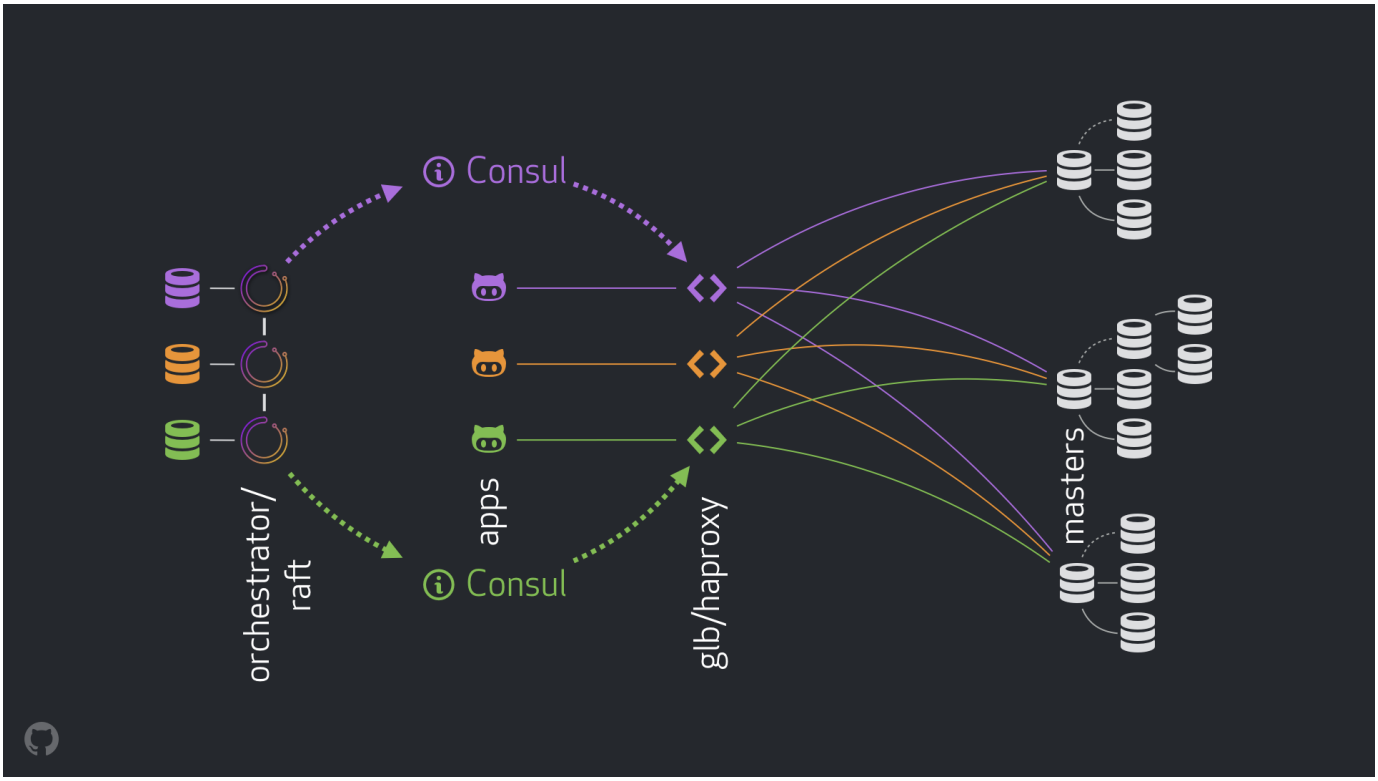
The solution worked, and GitHub has had successful MySQL failovers that went well under the radar, but we wanted our HA to improve on the following:

- Be data center agnostic.
- Be tolerant of data center failure.
- Remove unreliable cooperative workflows.
- Reduce total outage time.
- As much as possible, have lossless failovers.

GitHub's HA solution: orchestrator, Consul, GLB

Our new strategy, along with collateral improvements, solves or mitigates much of the concerns above. In today's HA setup, we have:

- orchestrator to run detection and failovers. We use a cross-DC orchestrator/raft setup as depicted below.
- Hashicorp's Consul for service discovery.
- GLB/HAProxy as a proxy layer between clients and writer nodes. Our GLB director is open sourced.
- `anycast` for network routing.



The new setup removes VIP and DNS changes altogether. And while we introduce more components, we are able to decouple the components and simplify the task, as well as be able to utilize solid and stable solutions. A breakdown follows.

A normal flow

On a normal day the apps connect to the write nodes through GLB/HAProxy.

The apps are never aware of the master's identity. As before, they use a name. For example, the master for `cluster1` would be `mysql-writer-1.github.net`. In our current setup, however, this name gets resolved to an anycast IP.

With `anycast`, the name resolves to the same IP everywhere, but traffic is routed differently based on a client's location. In particular, in each of our data centers we have GLB, our highly available load balancer, deployed on multiple boxes. Traffic to `mysql-writer-1.github.net` always routes to the local data center's GLB cluster. Thus, all clients are served by local proxies.

We run GLB on top of HAProxy. Our HAProxy has *writer pools*: one pool per MySQL cluster, where each pool has exactly one backend server: the cluster's *master*. All GLB/HAProxy boxes in all DCs have the exact same pools, and they all indicate the exact same backend servers in these pools. Thus, if an app wishes to write to `mysql-writer-1.github.net`, it matters not which GLB server it connects to. It will always get routed to the actual `cluster1` master node.

As far as the apps are concerned, discovery ends at GLB, and there is never a need for re-discovery. It's all on GLB to route the traffic to the correct destination.

How does GLB know which servers to list as backends, and how do we propagate changes to GLB?

Discovery via Consul

Consul is well known as a service discovery solution, and also offers DNS services. In our solution, however, we utilize it as a highly available key-value (KV) store.

Within Consul's KV store we write the identities of cluster masters. For each cluster, there's a set of KV entries indicating the cluster's master `fqdn`, port, ipv4, ipv6.

Each GLB/HAProxy node runs consul-template: a service that listens on changes to Consul data (in our case: changes to clusters masters data). `consul-template` produces a valid config file and is able to reload HAProxy upon changes to the config.

Thus, a change in Consul to a master's identity is observed by each GLB/HAProxy box, which then reconfigures itself, sets the new master as the single entity in a cluster's backend pool, and reloads to reflect those changes.

At GitHub we have a Consul setup in each data center, and each setup is highly available. However, these setups are independent of each other. They do not replicate between each other and do not share any data.

How does Consul get told of changes, and how is the information distributed cross-DC?

orchestrator/raft

We run an `orchestrator/raft` setup: `orchestrator` nodes communicate to each other via raft consensus. We have one or two `orchestrator` nodes per data center.

`orchestrator` is charged with failure detection, with MySQL failover, and with communicating the change of master to Consul. Failover is operated by the single `orchestrator/raft` leader node, but the *change*, the news that a cluster now has a new master, is propagated to all `orchestrator` nodes through the `raft` mechanism.

As `orchestrator` nodes receive the news of a master change, they each communicate to their local Consul setups: they each invoke a KV write. DCs with more than one `orchestrator` representative will have multiple (identical) writes to Consul.

Putting the flow together

In a master crash scenario:

- The `orchestrator` nodes detect failures.
- The `orchestrator/raft` leader kicks off a recovery. A new master gets promoted.
- `orchestrator/raft` advertises the master change to all `raft` cluster nodes.
- Each `orchestrator/raft` member receives a leader change notification. They each update the local Consul's KV store with the identity of the new master.
- Each GLB/HAProxy has `consul-template` running, which observes the change in Consul's KV store, and reconfigures and reloads HAProxy.
- Client traffic gets redirected to the new master.

There is a clear ownership of responsibilities for each component, and the entire design is both decoupled as well as simplified. `orchestrator` doesn't know about the load balancers. Consul doesn't need to know where the information came from. Proxies only care about Consul. Clients only care about the proxy.

Furthermore:

- There are no DNS changes to propagate.
- There is no TTL.
- The flow does not need the dead master's cooperation. It is largely ignored.

Additional details

To further secure the flow, we also have the following:

- HAProxy is configured with a very short `hard-stop-after` . When it reloads with a new backend server in a writer-pool, it automatically terminates any existing connections to the old master.
  - With `hard-stop-after` we don't even require cooperation from the clients, and this mitigates a split-brain scenario. It's noteworthy that this isn't hermetic, and *some time*

passes before we kill old connections. But there's then a point in time after which we're comfortable and expect no nasty surprises.

- We do not strictly require Consul to be available at all times. In fact, we only need it to be available at failover time. If Consul happens to be down, GLB continues to operate with the last known values and makes no drastic moves.
- GLB is set to validate the identity of the newly promoted master. Similarly to our context-aware MySQL pools, a check is made on the backend server, to confirm it is indeed a writer node. If we happen to delete the master's identity in Consul, no problem; the empty entry is ignored. If we mistakenly write the name of a non-master server in Consul, no problem; GLB will refuse to update it and keep running with last known state.

We further tackle concerns and pursue HA objectives in the following sections.

orchestrator/raft failure detection

`orchestrator` uses a holistic approach to detecting failure, and as such it is very reliable. We do not observe false positives: we do not have premature failovers, and thus do not suffer unnecessary outage time.

`orchestrator/raft` further tackles the case for a complete DC network isolation (aka DC fencing). A DC network isolation can cause confusion: servers within that DC can talk to each other. Is it *they* that are network isolated from other DCs, or is it *other DCs* that are being network isolated?

In an `orchestrator/raft` setup, the `raft` leader node is the one to run the failovers. A leader is a node that gets the support of the majority of the group (quorum). Our `orchestrator` node deployment is such that no single data center makes a majority, and any `n-1` DCs do.

In the event of a complete DC network isolation, the `orchestrator` nodes in that DC get disconnected from their peers in other DCs. As a result, the `orchestrator` nodes in the isolated DC cannot be the leaders of the `raft` cluster. If any such node did happen to be the leader, it steps down. A new leader will be assigned from any of the other DCs. That leader will have the support of all the other DCs, which are capable of communicating between themselves.

Thus, the `orchestrator` node that calls the shots will be one that is outside the network isolated data center. Should there be a master in an isolated DC, `orchestrator` will initiate the failover to replace it with a server in one of the available DCs. We mitigate DC isolation by delegating the decision making to the quorum in the non-isolated DCs.

Quicker advertisement

Total outage time can further be reduced by advertising the master change sooner. How can that be achieved?

When `orchestrator` begins a failover, it observes the fleet of servers available to be promoted. Understanding replication rules and abiding by hints and limitations, it is able to make an educated decision on the best course of action.

It may recognize that a server available for promotion is also an *ideal candidate*, such that:

- There is nothing to prevent the promotion of the server (and potentially the user has hinted that such server is preferred for promotion), and
- The server is expected to be able to take all of its siblings as replicas.

In such a case `orchestrator` proceeds to first set the server as writable, and immediately advertises the promotion of the server (writes to Consul KV in our case), even while asynchronously beginning to fix the replication tree, an operation that will typically take a few more seconds.

It is likely that by the time our GLB servers have been fully reloaded, the replication tree is already intact, but it is not strictly required. The server is good to receive writes!

Semi-synchronous replication

In MySQL's semi-synchronous replication a master does not acknowledge a transaction commit until the change is known to have shipped to one or more replicas. It provides a way to achieve lossless failovers: any change applied on the master is either applied or waiting to be applied on one of the replicas.

Consistency comes with a cost: a risk to availability. Should no replica acknowledge receipt of changes, the master will block and writes will stall. Fortunately, there is a timeout configuration, after which the master can revert back to asynchronous replication mode, making writes available again.

We have set our timeout at a reasonably low value: `500ms`. It is more than enough to ship changes from the master to local DC replicas, and typically also to remote DCs. With this timeout we observe perfect semi-sync behavior (no fallback to asynchronous replication), as well as feel comfortable with a very short blocking period in case of acknowledgement failure.

We enable semi-sync on local DC replicas, and in the event of a master's death, we expect (though do not strictly enforce) a lossless failover. Lossless failover on a complete DC failure is costly and we do not expect it.

While experimenting with semi-sync timeout, we also observed a behavior that plays to our advantage: we are able to influence the identity of the *ideal candidate* in the event of a master failure. By enabling semi-sync on designated servers, and by marking them as *candidates*, we are able to reduce total outage time by *affecting* the outcome of a failure. In our experiments we observe that we typically end up with the *ideal candidates*, and hence run quick advertisements.

Heartbeat injection

Instead of managing the startup/shutdown of the `pt-heartbeat` service on promoted/demoted masters, we opted to run it everywhere at all times. This required some patching so as to make `pt-heartbeat` comfortable with servers either changing their `read_only` state back and forth or completely crashing.

In our current setup `pt-heartbeat` services run on masters and on replicas. On masters, they generate the heartbeat events. On replicas, they identify that the servers are `read-only` and routinely recheck their status. As soon as a server is promoted as master, `pt-heartbeat` on that server identifies the server as writable and begins injecting heartbeat events.

orchestrator ownership delegation

We further delegated to orchestrator:

- Pseudo-GTID injection,
- Setting the promoted master as writable, clearing its replication state, and
- Setting the old master as `read_only`, if possible.

On all things new-master, this reduces friction. A master that is just being promoted is clearly expected to be alive and accessible, or else we would not promote it. It makes sense, then, to let `orchestrator` apply changes directly to the promoted master.

Limitations and drawbacks

The proxy layer makes the apps unaware of the master's identity, but it also masks the apps' identities from the master. All the master sees are connections coming from the proxy layer, and we lose information about the actual source of the connection.

As distributed systems go, we are still left with unhandled scenarios.

Notably, on a data center isolation scenario, and assuming a master is in the isolated DC, apps in that DC are still able to write to the master. This may result in state inconsistency once network is brought back up. We are working to mitigate this split-brain by implementing a reliable STONITH from within the very isolated DC. As before, *some time* will pass before bringing down the master, and there could be a short period of split-brain. The operational cost of avoiding split-brains altogether is very high.

More scenarios exist: the outage of Consul at the time of the failover; partial DC isolation; others. We understand that with distributed systems of this nature it is impossible to close all of the loopholes, so we focus on the most important cases.

The results

Our orchestrator/GLB/Consul setup provides us with:

- Reliable failure detection,
- Data center agnostic failovers,
- Typically lossless failovers,
- Data center network isolation support,
- Split-brain mitigation (more in the works),
- No cooperation dependency,
- Between `10 and 13 seconds` of total outage time in most cases.
  - We see up to `20 seconds` of total outage time in less frequent cases, and up to `25 seconds` in extreme cases.

Conclusion

The orchestration/proxy/service-discovery paradigm uses well known and trusted components in a decoupled architecture, which makes it easier to deploy, operate and observe, and where each component can independently scale up or down. We continue to seek improvements as we continuously test our setup.

Related posts

March 5, 2019

Engineering

Vulcanizer: a library for operating Elasticsearch

GitHub Engineering

February 24, 2019

Community

Highlights from Git 2.21

Taylor Blau

February 19, 2019

Engineering

Five years of the GitHub Bug Bounty program

philipturnbull

GitHub

Product

Features

Security

Enterprise

Case Studies

Pricing

Resources

Platform

Developer API

Partners

Atom

Electron

GitHub Desktop

Support

Help

Community Forum

Training

Status

Contact

Company

About

Blog

Careers

Press

Shop