GitLab

Gitaly: a Git gRPC service

2019-09-12

Amsterdam Go Meetup

Jacob Vosmaer <jacob@gitlab.com>

Introduction

1. GitLab
2. Go at GitLab
3. RPC and gRPC
4. Why GitLab needed a Git RPC service
5. Gitaly in numbers
6. What exactly does Gitaly do?
7. Why Go and gRPC
8. (skipped, out of time) Success: better metrics and logging thanks to middleware
9. (skipped) Challenge: RPC overhead

GitLab

"GitLab is a complete DevOps platform, delivered as a single application. From project planning and source code management to CI/CD, monitoring, and security."
https://about.gitlab.com

- Open core software product. Core is MIT licensed.
- Available both as Saas (gitlab.com) and self-managed software (run on your own server)
- High growth startup: 800+ employees, up from 600 in May 2019
- Fully remote company: there is no office, everybody is remote

Go at GitLab

Most of GitLab is written in Ruby on Rails and VueJS. Go is the "other" backend language at GitLab.

Go use cases at GitLab:

- Ease of deployment
- Better performance and concurrency compared to Ruby

Applications include:

- CI runner: gitlab-runner
- Offloading slow requests from Rails: gitlab-workhorse
- Static site hosting: gitlab-pages
- Security analysis tools
- Git RPC backend: gitaly

RPC and gRPC

What is RPC?

- RPC: Remote Procedure Call
- Use cases: mobile apps, microservices, internet of things, ...
- Example: HTTP REST API's, SOAP, gRPC

What is gRPC?

- Framework to build RPC systems: code generators and libraries
- Open source
- Supports many languages: C++, Java, Python, Go, Ruby, C#, ...
- Transport is http/2
- Binary serialization: Protobuf
- Relatively young (gRPC 1.0 was released in August 2016)

Why GitLab needed a Git RPC service

- In 2016, Git repositories on gitlab.com were sharded over 20+ NFS servers
- The application was unaware of NFS, it just looked like we had 20+ special "directories" where repositories could be stored
- This had several drawbacks, mostly around **observability** and **resilience**

GitLab and NFS: observability

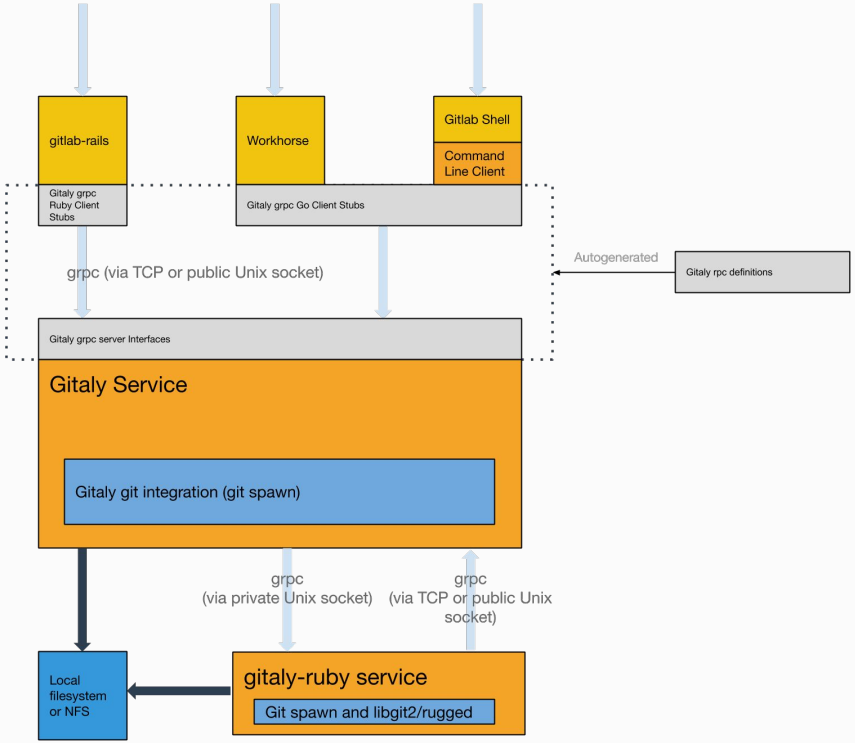
Observing behavior of GitLab was hard with NFS.

1. NFS makes it hard to attribute IO load to individual repositories or Git processes
2. Any part of the application could spin up a 'git' processes
3. Load spikes usually come from a single repository. How do we know which?

GitLab and NFS: resilience

- Particularly for gitlab.com, **resilience** of NFS was a problem.
- NFS server reboots leave client processes in "D" state (cannot be stopped, even by `kill -9`)
- NFS server reboots => almost all NFS clients go bad (many processes in D state), and also need to be rebooted

Gitaly architecture diagram



What exactly does Gitaly do?

Typical Gitaly request:

1. Some part of GitLab ("client") needs to access a Git repository
2. Client makes gRPC request to Gitaly
3. Gitaly starts `git` process on local filesystem
4. Gitaly sends result back to client as gRPC response

- Heavy lifting is done by `git` and libgit2
- Main Gitaly process never "opens" a repository
- RPC's usually map to Git commands, e.g. `git log -20 master`
- Gitaly's API is the result of migrating ~4 years worth of legacy Ruby code that accessed Git repositories

Gitaly in numbers

Gitaly is GitLab's Git backend

- RPC service written in Go (80%) and Ruby (20%)
- uses gRPC framework
- 150 RPC's
- 3200 commits on master branch in about 3 years (first commit November 2016)
- First 2 years mostly spent migrating existing GitLab code into Gitaly RPC's
- 250 tagged releases
- 4K requests/s on gitlab.com
- Team size has fluctuated between 2 and 5 engineers
- Over 100,000 installations worldwide

Why Go and gRPC

Why Go?

Because the alternative was Ruby. Advantages:

1. Smaller memory footprint
2. Better ability to handle many concurrent requests (no Global Interpreter Lock)

Why gRPC?

1. JSON was not attractive because we send lots of binary data
2. We expected the Gitaly API to grow quite big. Using a framework helped us structure the application as it grew.

The End

Thank you! Any questions?