



GitLab

Gitaly for Ruby Devs - Create Deep Dive

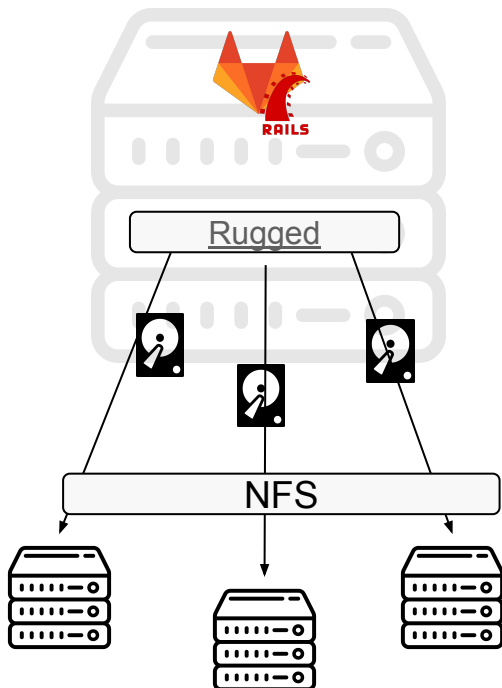
Bob Van Landuyt

2019-05-28



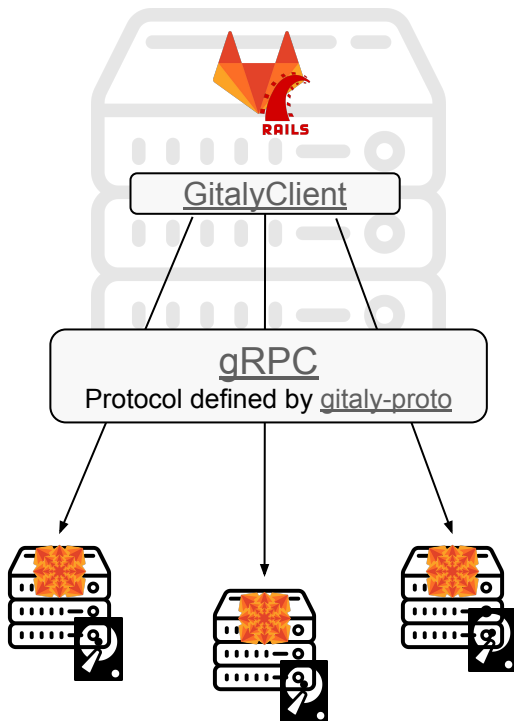
- What is Gitaly, where did it come from, why is it here?
- Case study: Applying patches (``git am``) as a user.
- Some tips, workflows

What is Gitaly: Before



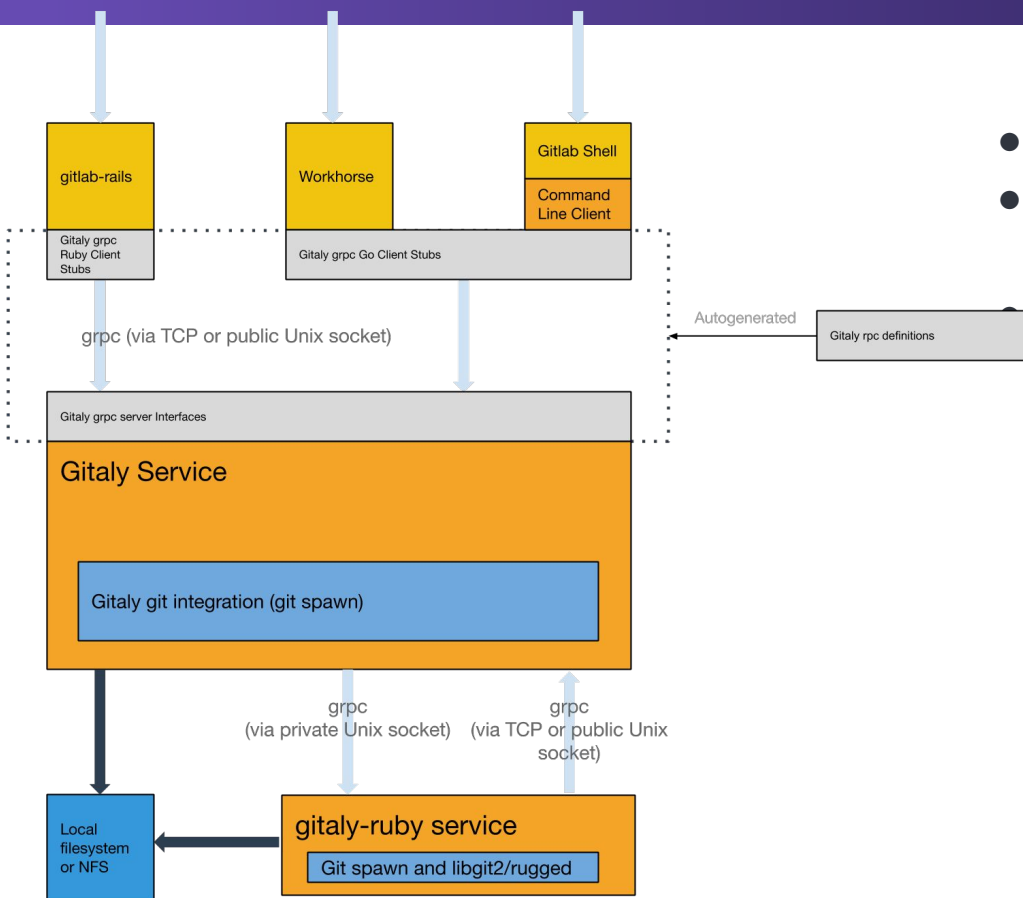
- + Rugged provides easy API
- + Adding storage was easy
- + High Availability easy because NFS
- + Rugged caches into memory mean less io than reopening the repository
- IO timeouts clogging up unicorns causing downtime

What is Gitaly: Now



- + Git commands running on the storage nodes
- + More robust
- More complex
- Overhead of a network call
- Overhead of opening the repository on each RPC call.

What is Gitaly: Really



- Several consumers including itself
- Gitaly ruby, ported code using Rugged
- Spins up multiple git-processes to do operations



- gitaly-proto
- `.proto`` files used to define protocol
 - operations.proto
 - Gitaly: service.operations package
 - Gitaly-ruby: GitalyServer::OperationService
 - Gitlab-rails: GitalyClient::OperationService
- Generate code
 - Rubygem: <https://rubygems.org/gems/gitaly-proto>
 - Go package: <https://godoc.org/gitlab.com/gitlab-org/gitaly-proto/go/gitalypb>

Case Study: Create a merge request with patches



- User emails patches
- Emails get picked up by gitlab-rails: Gitlab::Email::Handler::CreateMergeRequestHandler
- Subject contains the new branch name
- Body is merge request description
- Patches in attachments
- We need to create a branch from the HEAD of the repo
- Apply the patches from the attachments
- Create the MR



Code dive

Slides for future reference



```
rpc UserApplyPatch(stream UserApplyPatchRequest) returns (UserApplyPatchResponse) {  
  option (op_type) = {  
    op: MUTATOR  
    target_repository_field: "1.1"  
  };  
}
```

- Inside operation.proto
- UserApplyPatch: RPC name
- Stream UserApplyPatchRequest
 - Indicates the type of message sent
 - Stream means multiple messages could be included (Depending on size of the patches)
- Options: These might be needed for HA
 - Op: enum for the operation done
 - Target_repository_field: Where is the repository being modified defined



```
message UserApplyPatchRequest {
  message Header {
    Repository repository = 1;
    User user = 2;
    bytes target_branch = 3;
  }

  oneof user_apply_patch_request_payload {
    Header header = 1;
    bytes patches = 2;
  }
}
```

- Header contains the metadata for committing
 - User & Repository are defined in
- Patches will be the content of our patchfiles.
- Could be multiple requests, only the first one has the Header set

Note: Deprecating fields using `reserved` keyword



```
message UserApplyPatchResponse {  
  OperationBranchUpdate branch_update = 1;  
}
```

- Returns existing OperationBranchUpdate

```
message OperationBranchUpdate {  
  // If this string is non-empty the branch has been updated.  
  string commit_id = 1;  
  // Used for cache invalidation in GitLab  
  bool repo_created = 2;  
  // Used for cache invalidation in GitLab  
  bool branch_created = 3;  
}
```

The client: GitLab-rails



```
module Gitlab
  module Git
    module Patches
      class CommitPatches
        include Gitlab::Git::WrapsGitalyErrors

        def initialize(user, repository, branch, patch_collection)
          @user, @repository, @branch, @patches = user, repository, branch, patch_collection
        end

        def commit
          repository.with_cache_hooks do
            wrapped_gitaly_errors do
              operation_service.user_commit_patches(user, branch, patches.content)
            end
          end
        end

        private

        attr_reader :user, :repository, :branch, :patches

        def operation_service
          repository.raw.gitaly_operation_client
        end
      end
    end
  end
end
```

- Update the `gitaly-proto` gem
 - Point to git repo in Gemfile
- Wrap in a `Gitlab::Git` class:
Gitlab::Git::Patches::CommitPatches
 - Handles `GRPC::*` errors
 - Handles cache invalidation
 - Calls out to the OperationService

The client: GitLab-rails



```
def user_commit_patches(user, branch_name, patches)
  header = Gitaly::UserApplyPatchRequest::Header.new(
    repository: @gitaly_repo,
    user: Gitlab::Git::User.from_gitlab(user).to_gitaly,
    target_branch: encode_binary(branch_name)
  )
  reader = binary_io(patches)

  chunks = Enumerator.new do |chunk|
    chunk.yield Gitaly::UserApplyPatchRequest.new(header: header)

    until reader.eof?
      patch_chunk = reader.read(MAX_MSG_SIZE)

      chunk.yield(Gitaly::UserApplyPatchRequest.new(patches: patch_chunk))
    end
  end

  response = GitalyClient.call(@repository.storage, :operation_service, :user_apply_patch, chunks)

  Gitlab::Git::OperationService::BranchUpdate.from_gitaly(response.branch_update)
end
```

- GitalyClient::OperationService
- Turns our ruby object into GRPC-messages (a stream)
- Calls GitalyClient
 - Takes the service (from operations.proto)
 - And rpc (from operations.proto)
- Parses the response into a ruby object

Gitaly: The go-server (and client)



```
func (s *server) UserApplyPatch(stream gitalypb.OperationService_UserApplyPatchServer) error {
    firstRequest, err := stream.Recv()
    if err != nil {
        return err
    }

    header := firstRequest.GetHeader()
    if header == nil {
        return status.Errorf(codes.InvalidArgument, "UserApplyPatch: empty UserApplyPatch_Header")
    }

    if err := validateUserApplyPatchHeader(header); err != nil {
        return status.Errorf(codes.InvalidArgument, "UserApplyPatch: %v", err)
    }

    requestCtx := stream.Context()
    rubyClient, err := s.OperationServiceClient(requestCtx)
    if err != nil {
        return err
    }

    clientCtx, err := rubyserver.SetHeaders(requestCtx, header.GetRepository())
    if err != nil {
        return err
    }

    rubyStream, err := rubyClient.UserApplyPatch(clientCtx)
    if err != nil {
        return err
    }

    if err := rubyStream.Send(firstRequest); err != nil {
        return err
    }

    err = rubyserver.Proxy(func() error {
        request, err := stream.Recv()
        if err != nil {
            return err
        }
        return rubyStream.Send(request)
    })
    if err != nil {
        return err
    }

    response, err := rubyStream.CloseAndRecv()
    if err != nil {
        return err
    }

    return stream.SendAndClose(response)
}
```

- Request received in [operations.UserApplyPatch](#)
- Validates the header
- Passes the request on to gitaly-ruby
- Waits for gitaly-ruby to finish and passes the response back to the client



- Received in RubyServer::OperationsService
- Parses the request into ruby objects (Repository & User)
- Commits the patches using a Gitlab::Git::CommitPatches wrapper
- Builds the BranchUpdate result defined in the protocol
- Wraps any raised errors into GRPC errors

```
def user_apply_patch(call)
  stream = call.each_remote_read
  first_request = stream.next

  header = first_request.header
  user = Gitlab::Git::User.from_gitaly(header.user)
  target_branch = header.target_branch
  patches = stream.lazy.map(&:patches)

  branch_update = Gitlab::Git::Repository.from_gitaly_with_block(header.repository, call) do |repo|
    begin
      Gitlab::Git::CommitPatches.new(user, repo, target_branch, patches).commit
    rescue Gitlab::Git::PatchError => e
      raise GRPC::FailedPrecondition.new(e.message)
    end
  end

  Gitlab::UserApplyPatchResponse.new(branch_update: branch_update_result(branch_update))
end
```



```
module Gitlab
  module Git
    class CommitPatches
      attr_reader :user, :repository, :branch_name, :patches

      def initialize(user, repository, branch_name, patches)
        @user = user
        @branch_name = branch_name
        @patches = patches
        @repository = repository
      end

      def commit
        start_point = repository.find_branch(branch_name)&.target || repository.root_ref

        OperationService.new(user, repository).with_branch(branch_name) do
          repository.commit_patches(start_point, patches, extra_env: user.git_env)
        end
      end
    end
  end
end
```

- Gitlab::Git::CommitPatches performs the commits as the user
- Uses the OperationService to trigger hooks (as the user)
 - Can this user update this branch?
 - OperationService used for creating commits as auser.
- Uses the Repository
 - User in the env
 - Class executes git commands or calls out to rugged.



End of code dive? Questions?



- Gitaly does a lot already, you can probably reuse things
- Watch out for N+1s, there's a trick for counting requests in specs
- Suggest working on Gitaly inside \$GOPATH
 - ``go get gitlab.com/gitlab-org/gitaly``
- Use a local gitaly instance
 - Point gitaly-proto to your branch in the Gemfile
 - Vendor your own gitaly-proto for go
 - Symlinking or pointing the version to a branch
 - Shameless plug: GITALY_SERVER_VERSION on a fork
- Development process
 - Unit test test around your new RPC in gitlab-rails
 - Rspec for ``gitaly-ruby``
 - Go tests for others
- #g_gitaly