GitLab

Middleware for gRPC servers

2019-10-17

Amsterdam Go Meetup

Jacob Vosmaer <jacob@gitlab.com>

About me

- Jacob Vosmaer
- staff backend engineer at GitLab, Inc.
- working for GitLab since 2013
- daily work is on Gitaly, the Git RPC backend of GitLab

Introduction

1. What is gRPC again?
2. What does a grpc-go server look like?
3. Example 1: panic recovery
4. Example 2: logging
5. Middleware all the things
6. Example 3: authentication

gRPC and Gitaly

gRPC

- Remote Procedure Call framework
- Idiomatic libraries for 10 languages (including Go)
- Based on http/2 and Protobuf

Caveat

- My experience with gRPC is based on working on Gitaly, GitLab's Git RPC backend
- Gitaly started over 2 years ago and some of my knowledge may be outdated. If you have other experiences, please share during question time!

What does a grpc-go server look like? (1)

A grpc-go server starts with a protocol definition:

```
package helloworld;

service Greeter {
     rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
     string name = 1;
}

message HelloReply {
     string message = 1;
}
```

What does a grpc-go server look like? (2)

The grpc code generator then writes a .go file for you. Messages become structs:

```
type HelloRequest struct {
     Name                    string
}
```

And your server becomes an interface:

```
type GreeterServer interface {
     SayHello(context.Context, *HelloRequest) (*HelloReply, error)
}
```

What does a grpc-go server look like? (3)

```go
type server struct {
	pb.UnimplementedGreeterServer
}

func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, error)
{
	return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}

func main() {
	lis, err := net.Listen("tcp", port)
	if err != nil {
		log.Fatalf("failed to listen: %v", err)
	}
	s := grpc.NewServer()
	pb.RegisterGreeterServer(s, &server{})
	if err := s.Serve(lis); err != nil {
		log.Fatalf("failed to serve: %v", err)
	}
}
```

Handling panics

What happens if the handler panics?

```
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply,
error) {
	panic("something went wrong")
	return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}
```

Oops: your grpc-go server will crash!

Surprising, if you're used to Go's net/http server, which recovers from panics in HTTP handlers.

Handling panics (2)

Solution: add middleware!

From **grpc-ecosystem**: https://godoc.org/github.com/grpc-ecosystem/go-grpc-middleware/recovery

```
opts := []grpc.ServerOption{
     grpc.StreamInterceptor(grpc_recovery.StreamServerInterceptor()),
     grpc.UnaryInterceptor(grpc_recovery.UnaryServerInterceptor()),
}
s := grpc.NewServer(opts...)
```

Now a handler panic gets converted into an error response.

Note that there are two kinds of "interceptor" (Stream and Unary) and you usually want both.

Logging

Wouldn't it be nice to have an access log? Add more middleware!

```
logEntry := logrus.NewEntry(logrus.StandardLogger())
opts := []grpc.ServerOption{
	grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
		grpc_logrus.StreamServerInterceptor(logEntry),
		grpc_recovery.StreamServerInterceptor(),
	)),
	grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
		grpc_logrus.UnaryServerInterceptor(logEntry),
		grpc_recovery.UnaryServerInterceptor(),
	)),
}
```

Note how we have to use `grpc_middleware.ChainXXX` to chain multiple middlewares.

Logging

Example output:

```
ERRO[0005] finished unary call with code Internal        error="rpc
error: code = Internal desc = something went wrong" grpc.code=Internal
grpc.method=SayHello grpc.request.deadline="2019-10-11T14:43:58+02:00"
grpc.service=helloworld.Greeter
grpc.start_time="2019-10-11T14:43:57+02:00" grpc.time_ms=0.22
span.kind=server system=grpc
```

Middleware all the things

- Lots of good stuff at https://github.com/grpc-ecosystem/go-grpc-middleware
- You might also want: custom log fields ("tags")
- Prometheus: https://github.com/grpc-ecosystem/go-grpc-prometheus counts all requests for all methods in your server. New methods get counted automatically

Authentication

- Authentication is mentioned as a feature on the gRPC home page but documentation is scant
- Built-in support for: (1) TLS client certificates, (2) Google OAuth2 tokens

If I understand correctly, (2) should only be used to make gRPC calls to Google services: not relevant if you are building your own gRPC service and you are not Google...

Authentication: use TLS client certificates?

So it seems the only built-in auth option for my gRPC server is TLS client certificates.

- Create 1 or 2 Certificate Authorities (CA's) for client and server (may be the same CA)
- Deploy CA certificates along with all servers and clients of our service
- Create key pairs for clients and servers
- Securely deploy key pairs
- Now our gRPC client and server each identify themselves via TLS certificates

For Gitaly, this was not realistic.

- GitLab has 100,000+ installations: that means 100,000+ organizations
- Deploying and maintaining GitLab needs to be easy

Authentication: considerations

Some more considerations when we designed authentication for Gitaly. **Don't follow this blindly, think for yourself!**

- TLS: hard to automate because our automation (Omnibus) works at host level, not "cluster" level
- Considered creative solutions network topology agnostic TLS hacks: would have been hard to sell (literally)
- Saying "Kubernetes" and "service mesh" does not magically solve the problem: puts burden on others to install / manage those things
- Note that these are GitLab-specific points: we make server software that is run and installed by organizations other than GitLab, Inc.

Authentication: considerations (2)

Some more considerations when we designed authentication for Gitaly. **Don't follow this blindly, think for yourself!**

- Our choice: static secret that gets sent on every call as a bearer token
- This later got flagged during a security review so we changed to an HMAC-timestamp scheme: a time-limited token that does not reveal the static secret
- Important lesson: include a version in your token scheme, to allow gradual transitions
- `authentication: Bearer v2.YmxhIGJsYSBibGEsIGJsYSBibGEgYmxhIGJsYQ==`

Authentication: grpc-ecosystem to the rescue

https://github.com/grpc-ecosystem/go-grpc-middleware/blob/master/auth/examples_test.go

- Once again, grpc-ecosystem/go-grpc-middleware has nice goodies.
- Middleware with pluggable `func(context.Context) (context.Context, error)`
- Incoming context includes http/2 headers of the gRPC request
- Straightforward to implement bearer token authentication

Advantages:

- Simple and well understood
- Easier to deploy than mutual TLS

Disadvantages:

- We had to write our own authentication logic

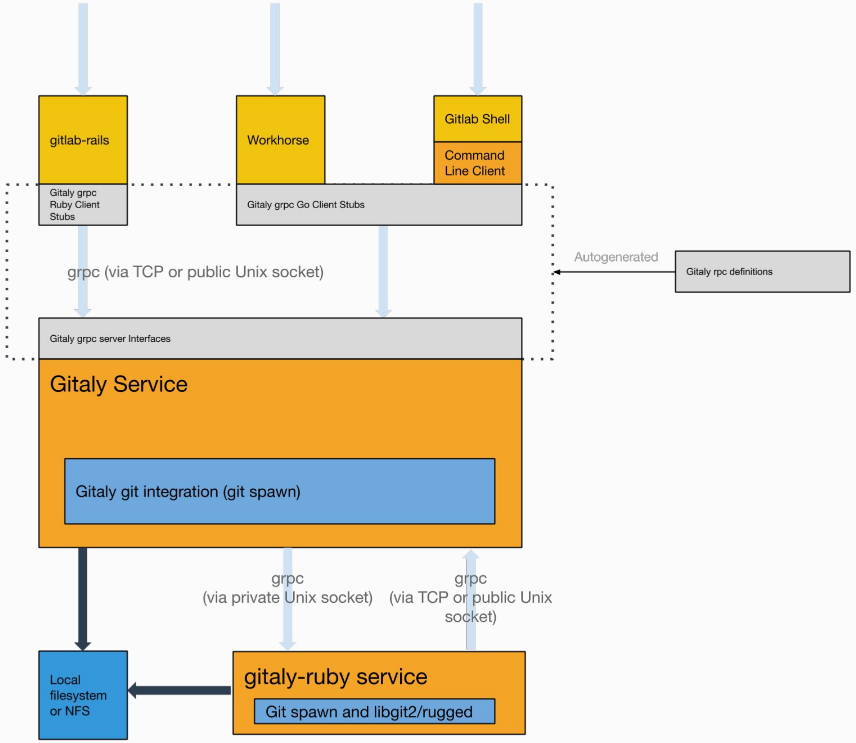
Conclusion

- There is nice gRPC middleware at https://github.com/grpc-ecosystem/
- Authentication batteries are not included with grpc-go
- (unless TLS client certificates are a good solution for you)

The end.

Questions?

Gitaly architecture diagram



gitlab-rails
Workhorse
Gitlab Shell
Command Line Client
Gitaly grpc Ruby Client Stubs
Gitaly grpc Go Client Stubs
grpc (via TCP or public Unix socket)
Autogenerated
Gitaly rpc definitions
Gitaly grpc server Interfaces
Gitaly Service
Gitaly git integration (git spawn)
grpc (via private Unix socket)
grpc (via TCP or public Unix socket)
Local filesystem or NFS
gitaly-ruby service
Git spawn and libgit2/rugged