gRPC in Go

@AlmogBaku

Who are you?

@AlmogBaku on github

1. A serial entrepreneur
2. Developer for 12 years
3. GitHub addicted (kubernetes maintainer, etc.)
4. *Consultant/freelancer*
5. Blog about entrepreneurship and development:

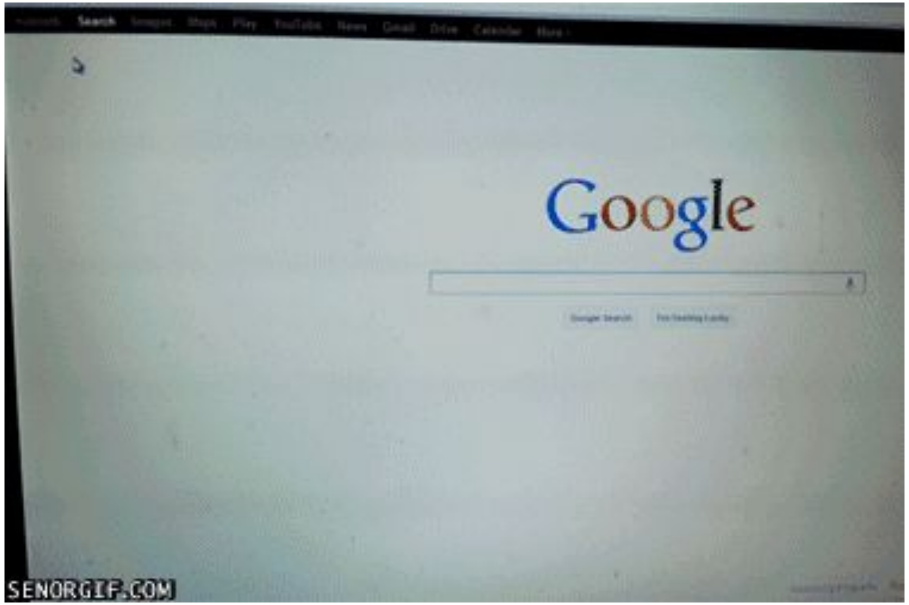
   www.AlmogBaku.com

What are we GOing to talk about?

1. What is Protocol Buffer?
2. What is gRPC?
3. How does it work (high-level)
4. How to use it w/ Go?
5. Tips and tricks

Disclaimer

You wanna know more? Google it!



Google tip: use the keyword “***golang***”

Who heard about Go?

Who heard about Protocol Buffer?

What is Protocol Buffer (protobuff)

- Interface Definition Language (IDL)
- Serializing for structured data
- Binary form
- Compact
- Fast

JSON:

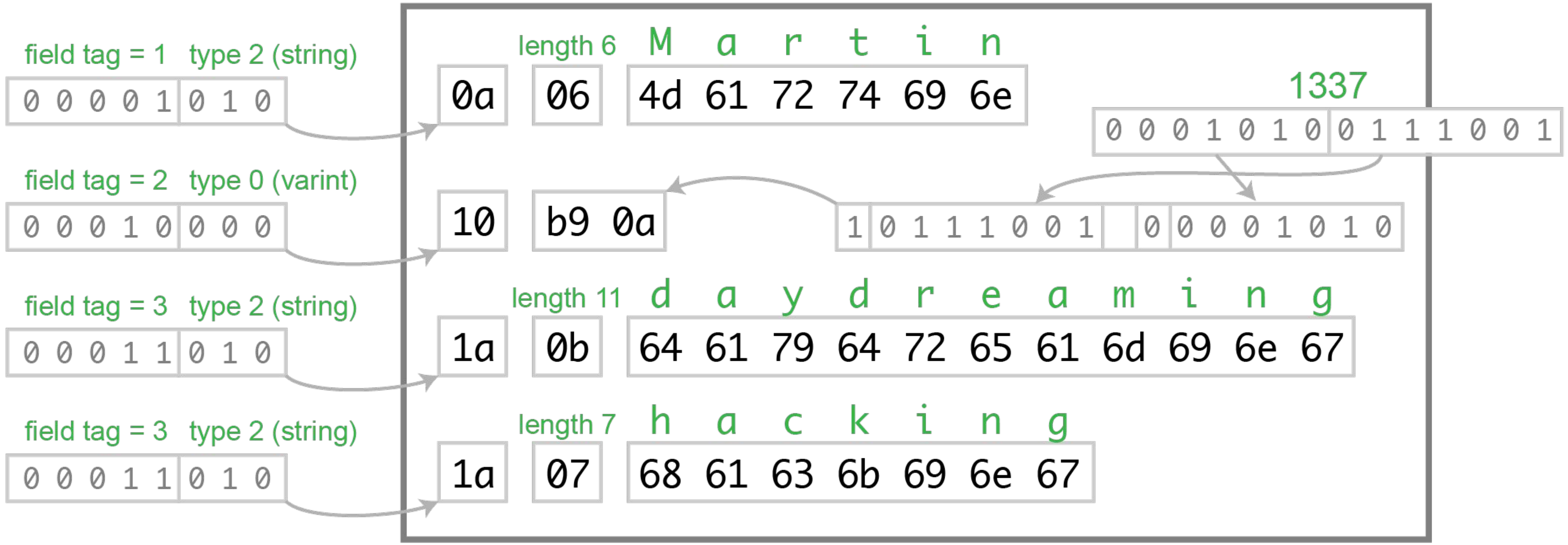
```
{
    "userName": "Martin",
    "favouriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

total: **82 bytes**

Protocol Buffer **definition**:

```
message Person {
    string user_name          = 1;
    int64  favourite_number   = 2;
    repeated string interests = 3;
}
```

Protocol Buffers



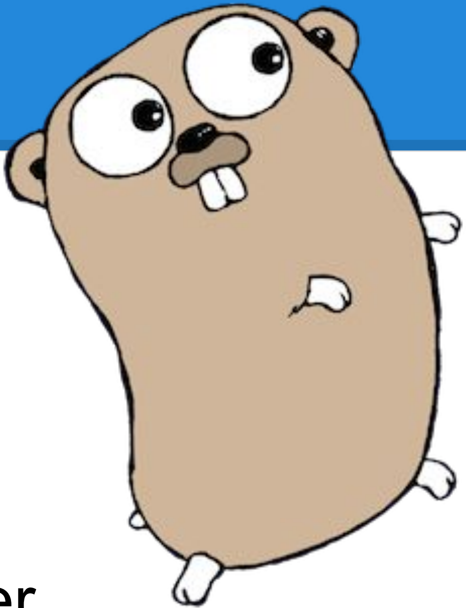
total: 33 bytes

Protocol Buffer message definition

```
syntax = "proto3";
package calculator;

message SumRequest {
   int32 a = 1;
   int32 b = 2;
}

message Result {
   int32 result = 1;
}
```

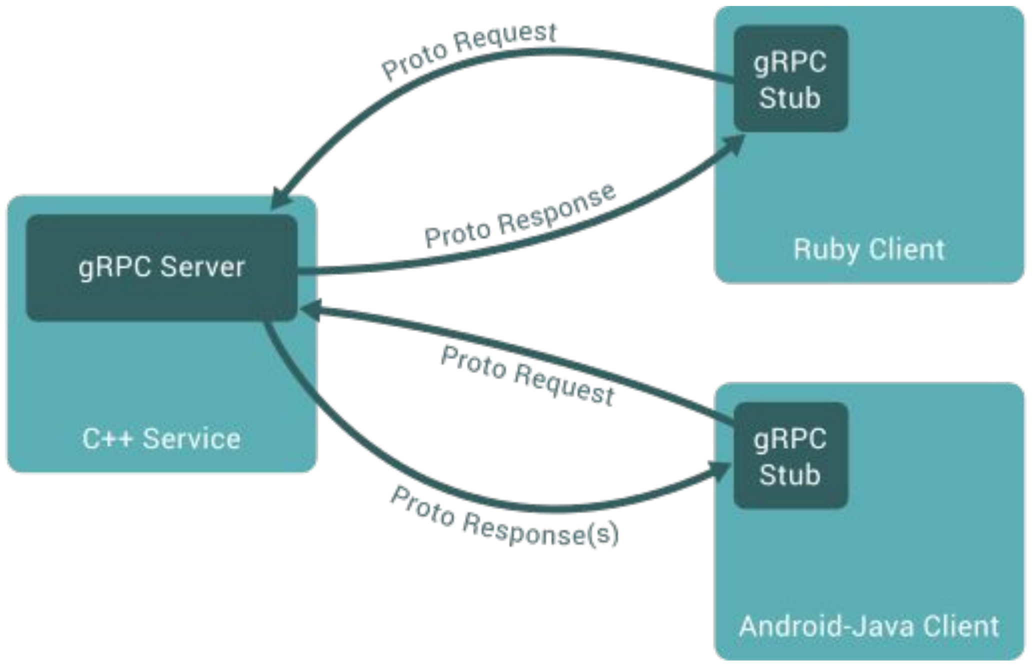
Who heard about gRPC?

What is GRPC



- **= g**RPC **R**emote **P**rocedure **C**alls
- Universal RPC framework
- Fast transportation over http2
- Messages encoded using Protocol Buffer
- Libraries in ~10 languages(native C, Go, Java)
- Layered & pluggable - bring your own monitoring, auth, load-balancing etc.

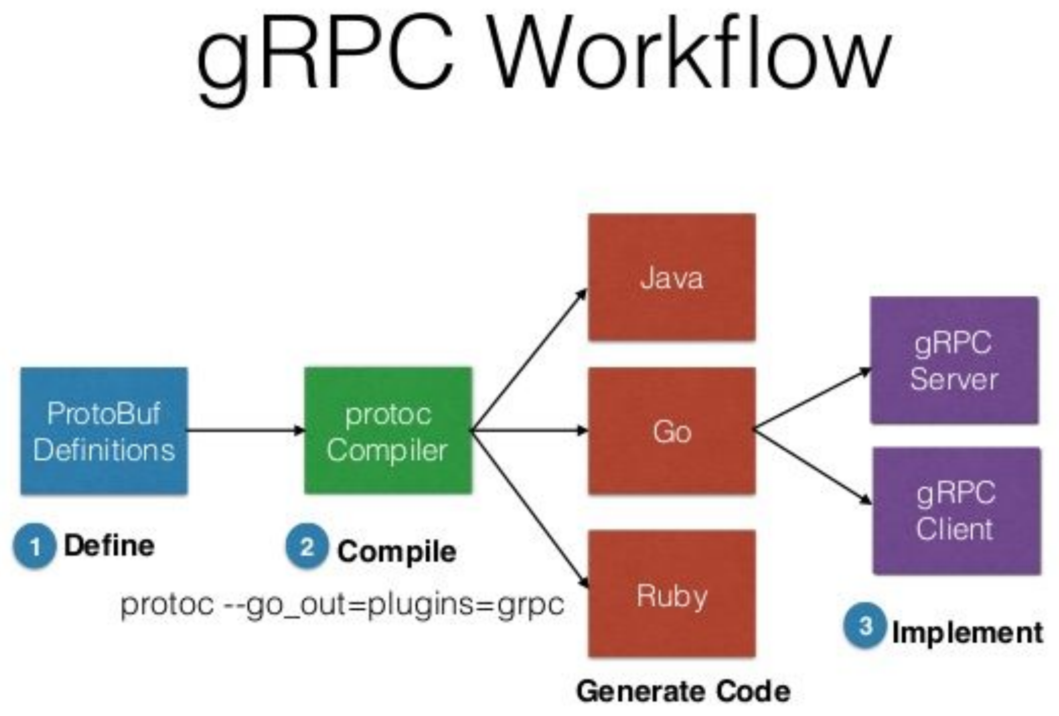
gRPC


Proto Request
gRPC Stub
Ruby Client
Proto Response
gRPC Server
C++ Service
Proto Request
gRPC Stub
Proto Response(s)
Android-Java Client

Protocol Buffer RPC definition

```
syntax = "proto3";
package calculator;

message SumRequest {
   int32 a = 1;
   int32 b = 2;
}

message Result {
   int32 result = 1;
}

service Math {
   rpc Sum (SumRequest) returns (Result);
}
```

Let’s try...


gRPC Workflow
ProtoBuf Definitions
protoc Compiler
Java
Go
Ruby
gRPC Server
gRPC Client
1 Define
2 Compile
protoc --go_out=plugins=grpc
Generate Code
3 Implement

gRPC Server implementation

```
func main() {
  //TCP Listener
  grpcListener, _ := net.Listen("tcp", ":5897")

  //Create a gRPC server
  baseServer := grpc.NewServer()
  //Bind implementation to the server
  calculator.RegisterMathServer(baseServer, &server{})

  fmt.Println("Server is running on " + grpcListener.Addr().String())

  //Bind gRPC server to the TCP
  baseServer.Serve(grpcListener)
}

type server struct{}

func (s *server) Sum(ctx context.Context, sumRequest *calculator.SumRequest) (*calculator.Result, error) {
  spew.Dump(sumRequest)
  return &calculator.Result{Result: sumRequest.A + sumRequest.B}, nil
}
```

gRPC Client implementation

```go
func main() {
  flag.Parse()

  //create the connection
  conn, _ := grpc.Dial(":5897", grpc.WithInsecure())
  defer conn.Close()

  //create a client
  client := calculator.NewMathClient(conn)

  //Create a request
  a, _ := strconv.Atoi(flag.Arg(0))
  b, _ := strconv.Atoi(flag.Arg(1))
  resp, _ := client.Sum(context.Background(), &calculator.SumRequest{ int32(a), int32(b)})

  spew.Dump(resp)
}
```

gRPC Client implementation

```
syntax = "proto3";
package calculator;

message SumRequest {
   int32 a = 1;
   int32 b = 2;
}

message Result {
   int32 result = 1;
}

service Math {
   rpc Sum (SumRequest) returns (Result);
}
```

Type of calls

- **Simple RPC / Unary call**

  The client send a request and waits for a response. Like a normal function
- **Server-side Streaming RPC**
  The client send a request and gets a stream to read a sequence of messages back until it’s over
- **Client-side Streaming RPC**
  The client send a stream with sequence of messages, once it’s over the server respond with a single response
- **Bidirectional Streaming RPC**
  Both the client and the server “chat” and stream data independently(not necessary related to the other side, or responding to each other)

Streams definition

```
syntax = "proto3";
package calculator;

message SumNumberRequest {
   int32 number = 1;
}

message Result {
   int32 result = 1;
}

service Math {
   rpc SumAll (stream SumNumberRequest) returns (Result); //client stream
   rpc Rand (Rand Request) returns (stream Result); //server stream
   rpc Chat (stream Msg) returns (stream Msg); //bi-dir streaming
}
```

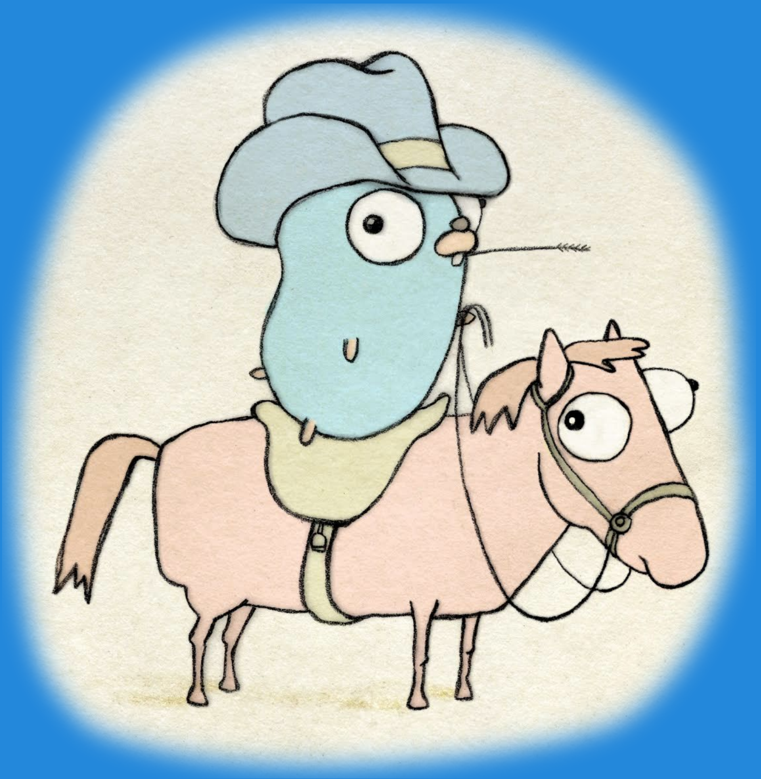
gRPC Server read stream

```go
func (s *server) SumAll(stream calculator.Math_SumAllServer) error {
  var sum int32 = 0
  for {
     numReq, err := stream.Recv()
     if err == io.EOF {
        return stream.SendAndClose(&calculator.Result{Result: sum})
     }
     if err != nil {
        return err
     }
     spew.Dump(numReq)
     sum += numReq.Number
  }
}
```

gRPC Client send stream

```go
func main() {
  flag.Parse()

  //create the connection
  conn, _ := grpc.Dial(":5897", grpc.WithInsecure())
  defer conn.Close()

  //create a client
  client := calculator.NewMathClient(conn)

  //Stream requests
  stream, _ := client.SumAll(context.Background())
  for _, num := range flag.Args() {
     n, _ := strconv.Atoi(num)
     stream.Send(&calculator.SumNumberRequest{Number: int32(n)})
  }

  resp, _ := stream.CloseAndRecv()
  spew.Dump(resp)
}
```

Tips and Tricks



Real life is a bit more complicated...

API Design

Idempotency

It should be safe to retry an RPC, without knowing whether it was processed.

BAD

```
message WireRequest {
   string from = 1;
   string to = 2;
   float amount = 3;
}

message Response {
   int64 confirmation = 1;
}
```

GOOD

```
message WireRequest {
   string from = 1;
   string to = 2;
   float amount = 3;
   string UUID = 4;
}

message Response {
   int64 confirmation = 1;
}
```

API Design

Avoid long-running operations

- The longer it takes, more likely you’ll have a retry
- Perform in background - send results async

Define default behavior

- Protobuf will “zeroize” null fields by default
- Prefer “UNKNOWN” as the default(0) option for enum

API Design

Avoid batch operations as unary

- Use stream or multiple calls instead
- Error handling become complex

Errors

Don’t panic!

- May crash the server…
- Return errors to the callers

Propagate

- Blindly return errors from libraries can be difficult to debug

Deadlines (timeouts)

Always use deadlines

Deadlines allow both client and servers to abort operations

```go
ctx, cancel := context.WithTimeout(context.Background(), 10 * time.Second)
resp, _ := client.Sum(ctx, &req)
```

Deadlines (timeouts)

Always use deadlines

On the server side, you should also care about the context

```
func (s *server) MyReqHandler(ctx context.Context, ...) (*Result, error) {
  if ctx.Err() == context.Canceled {
     return nil, status.New(codes.Canceled, "Client cancelled, abandoning.").Err()
  }
  ...
}
```

Deadlines (timeouts)

Recycle deadlines

Sometimes, your server is also a client.

Reuse the context, which carries the deadline.

```
func (s *server) MyReqHandler(ctx context.Context, ...) (*Result, error) {
  client.Call(ctx, ...)
  ...
}
```

<u>Be aware:</u> server can wait longer than necessary to fail, and not to retry!

Deadlines (timeouts)

Recycle deadlines

Define a new deadline based on the client’s.

```go
func (s *server) MyReqHandler(ctx context.Context, ...) (*Result, error) {
  ctx2, cancel := context.WithTimeout(ctx, 10 * time.Second)
  client.Call(ctx2, ...)
  ...
}
```

Protocol Buffers with Go Gadgets

https://github.com/gogo/protobuf

1. fork of golang/protobuf
2. faster
3. generating extra helper code
4. Cool □

Protocol Buffers with Go Gadgets

```
message Decision {
   Rule rule = 1;
   uint64 remain = 2; //kb

   Action do = 3;
   google.protobuf.Duration TTL = 5 [(gogoproto.stdduration) = true, (gogoproto.nullable) = false];
}
```

Go gRPC middleware

https://github.com/grpc-ecosystem/go-grpc-middleware

- Set of middlewares, helpers, interceptors for go gRPC
- Features such as:
  - retries
  - customizable auth
  - logging
  - monitoring
  - retries
  - etc

gokit

https://gokit.io

- Gokit is a great toolkit for go microservices
- Gokit offers a boilerplate and set of tools for creating microservices
- Gokit ecosystem also handles multiple issues such as logging/monitoring/load-balancing/rate-limit/etc.

Thanks.

@AlmogBaku