High Performance gRPC
(in Go)

FOSDEM 2025

FOSDEM'25

Aurélien Deroide

What is gRPC ?

- RPC protocol
- Schema first approach with spec written in protobuf files
- Easy cross-language support
- Based on http2
- Binary encoding
- Supports unary or streaming RPCs

Basic example: a Unary requ

```
type CreateUserRequest struct {
    state         protoimpl.MessageState `protogen:"open.v1"`
    Id            uint64                 `protobuf:"varint,1,opt,
    Name          string                 `protobuf:"bytes,2,opt,n
    Age           uint64                 `protobuf:"varint,3,opt,
    PaidPlan      bool                   `protobuf:"varint,4,opt,
    CreatedAt     uint64                 `protobuf:"varint,5,opt,
    Status        string                 `protobuf:"bytes,6,opt,n
    Substriptions []string               `protobuf:"bytes,7,rep,n
    Email         string                 `protobuf:"bytes,8,opt,n
    unknownFields protoimpl.UnknownFields
    sizeCache     protoimpl.SizeCache
}
```

myservice.pb.go

```
message CreateUserRequest{
    uint64 id = 1;
    string name = 2;
    uint64 age = 3;
    bool paid_plan = 4;
    uint64 createdAt = 5;
    string status = 6 ;
    repeated string subsriptions = 7;
    string email = 8;
}

message CreateUserResponse{
    uint64 id = 1;
    string name = 2;
    uint64 age = 3;
    bool paid_plan = 4;
    uint64 createdAt = 5;
    string status = 6 ;
    repeated string subsriptions = 7;
    string email = 8;
}

service MyService {
    rpc CreateUser (CreateUserRequest) returns (CreateUserResponse) {}
}
```

myservice.proto

```
type BasicImpl struct {
    pb.UnimplementedMyServiceServer
}

func (BasicImpl) CreateUser(_ context.Context, req *pb.CreateUserRequest,
) (*pb.CreateUserResponse, error) {
    // Dumb echo
    return &pb.CreateUserResponse{
        Id:           req.Id,
        Name:         req.Name,
        Age:          req.Age,
        PaidPlan:     req.PaidPlan,
        CreatedAt:    req.CreatedAt,
        Status:       req.Status,
        Subsriptions: req.Subsriptions,
        Email:        req.Email,
    }, nil
}
```

main.go

```go
func BenchmarkUnary(b *testing.B) { run benchmark
    // Error handling elided
    //Setup server
    lis, _ := net.Listen("tcp", "localhost:1234")
    defer lis.Close()
    grpcServer := grpc.NewServer()
    pb.RegisterMyServiceServer(grpcServer, BasicImpl{})
    go func() {
        grpcServer.Serve(lis)
    }()

    //Setup client
    cc, _ := grpc.NewClient("localhost:1234",
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    client := pb.NewMyServiceClient(cc)

    // benchmark
    ctx := context.Background()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        _, _ = client.CreateUser(ctx, &pb.CreateUserRequest{
            Id:          1234,
            Name:        "my-name",
            Age:         42,
            PaidPlan:    true,
            CreatedAt:   567153,
            Status:      "PREMIUM",
            Subsriptions: []string{"foo", "bar", "baz"},
            Email:       "name@gmail.com",
        })
    }
}
```

1. Client marshalls request
2. Client sends request
3. Server receives request
4. Server unmarshalls request
5. Server executes handler
6. Server marshalls response
7. Server sends response
8. Client receives response
9. Client unmarshalls response

```
❯ go test -bench=BenchmarkUnary$ .
goos: linux
goarch: amd64
pkg: github.com/aureliar8/high-perf-grpc
cpu: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
BenchmarkUnary     32554              36327 ns/op
PASS
ok      github.com/aureliar8/high-perf-grpc     1.709s
```

1. Client marshalls request
2. Client sends request
3. Server receives request
4. Server unmarshalls request
5. Server executes handler
6. Server marshalls response
7. Server sends response
8. Client receives response
9. Client unmarshalls response

```go
// Codec defines the interface gRPC uses to encode and decode messages.  Note
// that implementations of this interface must be thread safe; a Codec's
// methods can be called from concurrent goroutines.
type Codec interface {
    // Marshal returns the wire format of v.
    Marshal(v any) ([]byte, error)
    // Unmarshal parses the wire format into v.
    Unmarshal(data []byte, v any) error
```

vtprotobuf plugin: faster marshalling & unmarshalling

github.com/planetscale/vtprotobuf/

```yaml
version: v1
plugins:
  - plugin: buf.build/protocolbuffers/go
    out: ./
    opt: module=github.com/aureliar8/high-perf-grpc
  - plugin: buf.build/grpc/go
    out: ./
    opt: module=github.com/aureliar8/high-perf-grpc
  - plugin: buf.build/community/planetscale-vtprotobuf
    out: ./
    opt:
      - module=github.com/aureliar8/high-perf-grpc
```

buf.gen.yaml

```
❯ tree pb
pb
├── myservice_grpc.pb.go
├── myservice.pb.go
└── myservice_vtproto.pb.go
```

Using a custom codec

```go
// Codec defines the interface gRPC uses to encode and decode messages.  Note
// that implementations of this interface must be thread safe; a Codec's
// methods can be called from concurrent goroutines.
type Codec interface {
    // Marshal returns the wire format of v.
    Marshal(v any) ([]byte, error)
    // Unmarshal parses the wire format into v.
    Unmarshal(data []byte, v any) error
```

```go
func init() {
    if os.Getenv("VTCODEC") != "" {
        encoding.RegisterCodec(VTCodecV1{})
    }
}
```

```go
type VTCodecV1 struct{}

type vtprotoMessage interface {
    MarshalVT() ([]byte, error)
    UnmarshalVT([]byte) error
}

func (VTCodecV1) Marshal(v interface{}) ([]byte, error) {
    vt, ok := v.(vtprotoMessage)
    if !ok {
        return nil, fmt.Errorf("failed to marshal, message is %T", v)
    }
    return vt.MarshalVT()
}

func (VTCodecV1) Unmarshal(data []byte, v interface{}) error {
    vt, ok := v.(vtprotoMessage)
    if !ok {
        return fmt.Errorf("failed to unmarshal, message is %T", v)
    }
    return vt.UnmarshalVT(data)
}
```

```
❯ go test -bench=BenchmarkUnary$ .
goos: linux
goarch: amd64
pkg: github.com/aureliar8/high-perf-grpc
cpu: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
BenchmarkUnary      32554                 36327 ns/op
PASS
ok      github.com/aureliar8/high-perf-grpc     1.709s
```

```
❯ VTCODEC=1 go test -bench=BenchmarkUnary$ .
goos: linux
goarch: amd64
pkg: github.com/aureliar8/high-perf-grpc
cpu: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
BenchmarkUnary      34558                 33826 ns/op
PASS
ok      github.com/aureliar8/high-perf-grpc     1.714s
```

```
❯ go test -bench=BenchmarkUnary$ . -count=10 | tee default.txt
```

```
❯ VTCODEC=1 go test -bench=BenchmarkUnary$ . -count=10 | tee vt.txt
```

```
❯ benchstat default.txt vt.txt
goos: linux
goarch: amd64
pkg: github.com/aureliar8/high-perf-grpc
cpu: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
      │ default.txt │                vt.txt                │
      │   sec/op    │   sec/op     vs base                 │
Unary   36.21µ ± 1%   31.68µ ± 2%  -12.51% (p=0.000 n=10)
```

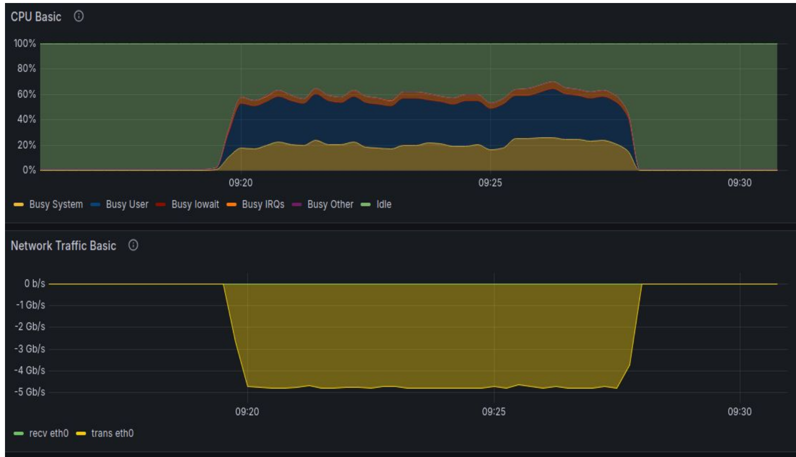
Another example: Grpc stream with large amount of data

```
message Chunk {
    bytes data = 2 ;
}

message PutResult {}

message GetRequest {}

service MyService {
    rpc Put (stream Chunk) returns (PutResult)    {}
    rpc Get (GetRequest)   returns (stream Chunk) {}
}
```

```
func (BasicImpl) Put(stream grpc.ClientStreamingServer[pb.Chunk, pb.PutResult]) error {
    for {
        chunk, err := stream.Recv()
        if err != nil {
            if err == io.EOF {
                return stream.SendAndClose(&pb.PutResult{})
            }
            return err
        }
        // do something with received data
        _ = chunk
    }
}

func (BasicImpl) Get(req *pb.GetRequest, stream grpc.ServerStreamingServer[pb.Chunk]) error {
    data := make([]byte, 16*1024)
    _, _ = rand.Read(data)
    for i := 0; i < 1024; i++ { // 16MB of data in total
        chunk := &pb.Chunk{Data: data}
        stream.Send(chunk)
    }
    return nil
}
```

How fast is it ?

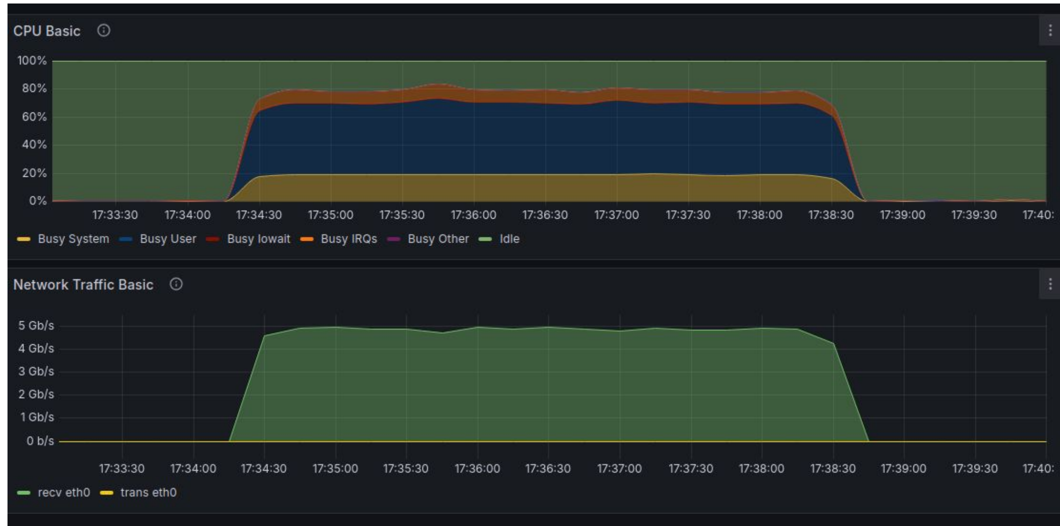
Tested with 2 vms (1 client & 1 server) of 2vCPU, with a 5Gbps link



My gRPC server: 1.3 CPU core to saturate the link when doing Get

Caddy http2 file server: 0.2 CPU core

It’s even worse when doing Put



My gRPC server: 1.6 CPU core to saturate the link
When doing Put

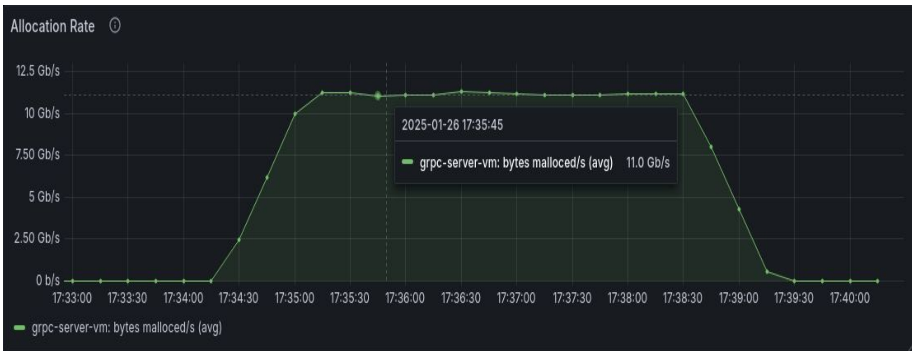
48.8s (100%)

root

grpc.(*Server).serveStreams.func2.1
grpc.(*Server).handleStream
grpc.(*Server).processStreamingRPC
pb._MyService_Put_Handler
main.BasicImpl.Put
com/aureliar8/high-perf-grpc/pb.sizeCache int32 }]).Recv
grpc.(*serverStream).RecvMsg
grpc.recv
grpc.recvAndDecompress
grpc.(*parser).recvMsg
grpc.nopBufferPool.Get
runtime.makeslice
runtime.mallocgc
memclrNoHeapPointers
(*Stream).Read
io.ReadFull
ReadAtLeast
Read
Read
Read
read
memmove
read
proto.codec.Unmarshal
proto.Unmarshal
UnmarshalOptions.unmarshal
(*MessageInfo).unmarshal
unmarshalPointer
unmarshalPointerEager
impl.consumeBytesNoZero
memmove
growslice
mallocgc
nextFree
refill
grow
alloc
func1
alloc

grpc.(*Server).handleRawConn.func1
grpc.(*Server).serveStreams
transport.(*http2Server).HandleStreams
http2.(*Framer).ReadFrame
io.ReadFull
io.ReadAtLeast
bufio.(*Reader).Read
net.(*conn).Read
net.(*netFD).Read
poll.(*FD).Read
poll.ignoringEINTRIO
syscall.Read
syscall.read
syscall.Syscall
syscall.RawSyscall6
syscall.Syscall6
handleData
(*Buffer).Write
write
grow
memmove
put
growSlice

gcBgMarkWorker
systemstack
func2
gcDrainMarkWorkerFract
runtime.gcDrain
scanobject
markroot
func1
run
func2
func1



Where does memory allocation comes from ?

53.6GB (100%)

root

grpc.(*Server).serveStreams.func2.1 | func1

grpc.(*Server).handleStream | serveStreams

grpc.(*Server).processStreamingRPC | HandleStreams

pb._MyService_Put_Handler | handleData

main.BasicImpl.Put | Write

MessageState "protogen:\"open.v1\""; github.com/aureliar8/high-perf-grpc/pb.unknownFields []uint8; github.com/aureliar8/high-perf-grpc/pb.sizeCache int32 }]).Recv | grow

grpc.(*serverStream).RecvMsg | growSlice

grpc.recv

grpc.recvAndDecompress | proto.codec.Unmarshal

grpc.(*parser).recvMsg | proto.Unmarshal

grpc.nopBufferPool.Get | proto.UnmarshalOptions.unmarshal

impl.(*MessageInfo).unmarshal

impl.(*MessageInfo).unmarshalPointer

impl.(*MessageInfo).unmarshalPointerEager

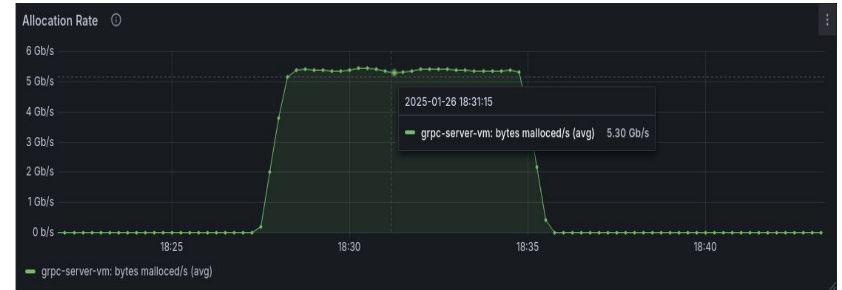
impl.consumeBytesNoZero

```go
// For the two compressor parameters, both should not be set, but if they are,
// dc takes precedence over compressor.
func recv(p *parser, c baseCodec, s *transport.Stream, dc Decompressor, m any, maxReceiveMessageSize int,
    payInfo *payloadInfo, compressor encoding.Compressor) error {
    buf, cancel, err := recvAndDecompress(p, s, dc, maxReceiveMessageSize, payInfo, compressor)
    if err != nil {
        return err
    }
    defer cancel()

    if err := c.Unmarshal(buf, m); err != nil {
        return status.Errorf(codes.Internal, "grpc: failed to unmarshal the received message: %v", err)
    }
    return nil
}
```

Reducing allocation from the grpc lib when receiving data

- Upgrade grpc to >= **v1.66** (Sept 2024)
- If using grpc < **v1.66** :

```
grpcServer := grpc.NewServer(
    experimental.RecvBufferPool(grpc.NewSharedBufferPool()),
)
```



Reducing memory allocation when unmarshalling

```go
func (BasicImpl) Put(stream grpc.ClientStreamingServer[pb.Chunk, pb.PutResult]) error {
    for {
        chunk, err := stream.Recv()
        if err != nil {
            if err == io.EOF {
                return stream.SendAndClose(&pb.PutResult{})
            }
            return err
        }
        // do something with received data
        _ = chunk
    }
}
```

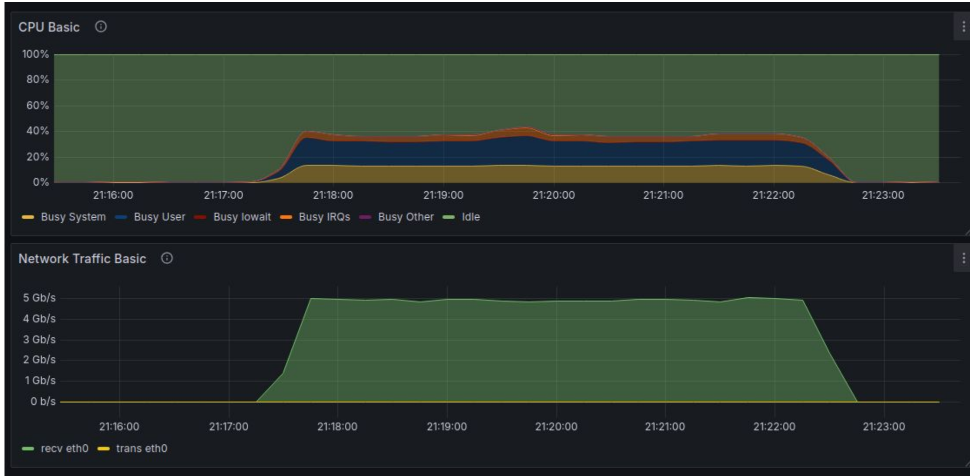
```go
type Chunk struct {
    state         protoimpl.MessageState
    Data          []byte
    unknownFields protoimpl.UnknownFields
    sizeCache     protoimpl.SizeCache
}
```

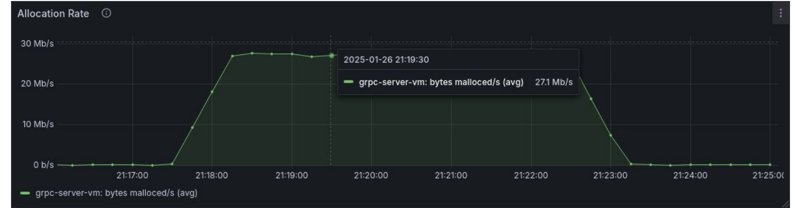
```go
// Recv reads one message from the stream of responses generated by the server.
// The type of the message returned is determined by the Res type parameter
// of the GenericClientStream receiver.
func (x *GenericClientStream[Req, Res]) Recv() (*Res, error) {
    m := new(Res)
    if err := x.ClientStream.RecvMsg(m); err != nil {
        return nil, err
    }
    return m, nil
}
```

```go
func (EfficientImpl) Put(stream grpc.ClientStreamingServer[pb.Chunk, pb.PutResult]) error {
    for {
        chunk := pb.ChunkFromVTPool()
        err := stream.RecvMsg(chunk)
        if err != nil {
            if err == io.EOF {
                return stream.SendAndClose(&pb.PutResult{})
            }
            return err
        }
        // do something with received data
        _ = chunk

        chunk.ReturnToVTPool()
    }
}
```

```go
// For the two compressor parameters, both should not be set, but if they are,
// dc takes precedence over compressor.
func recv(p *parser, c baseCodec, s *transport.Stream, dc Decompressor, m any, maxReceiveMessageSize int,
    payInfo *payloadInfo, compressor encoding.Compressor) error {
    buf, cancel, err := recvAndDecompress(p, s, dc, maxReceiveMessageSize, payInfo, compressor)
    if err != nil {
        return err
    }
    defer cancel()

    if err := c.Unmarshal(buf, m); err != nil {
        return status.Errorf(codes.Internal, "grpc: failed to unmarshal the received message: %v", err)
    }
    return nil
}
```

Reducing memory allocation when unmarshalling





~ 0.7 CPU core when saturating network on Put

Measuring the Get workload

18.83GB (100%) 18.61GB (98.8%)

root
grpc.(*Server).serveStreams.func2.1
grpc.(*Server).handleStream
grpc.(*Server).processStreamingRPC
pb._MyService_Get_Handler
main.BasicImpl.Get
com/aureliar8/high-perf-grpc/pb.unknownFields []uint8; github.com/aureliar8/high-perf-grpc/pb.sizeCache int32 }]).Send
grpc.(*serverStream).SendMsg
grpc.prepareMsg
grpc.encode
proto.codec.Marshal
proto.Marshal
proto.MarshalOptions.marshal

A new codec API to reduce allocation

```go
// Codec defines the interface gRPC uses to encode and decode messages.  Note
// that implementations of this interface must be thread safe; a Codec's
// methods can be called from concurrent goroutines.
type Codec interface {
    // Marshal returns the wire format of v.
    Marshal(v any) ([]byte, error)
    // Unmarshal parses the wire format into v.
    Unmarshal(data []byte, v any) error
```

```go
// CodecV2 defines the interface gRPC uses to encode and decode messages. Note
// that implementations of this interface must be thread safe; a CodecV2's
// methods can be called from concurrent goroutines.
type CodecV2 interface {
    // Marshal returns the wire format of v. The buffers in the returned
    // [mem.BufferSlice] must have at least one reference each, which will be freed
    // by gRPC when they are no longer needed.
    Marshal(v any) (out mem.BufferSlice, err error)
    // Unmarshal parses the wire format into v. Note that data will be freed as soon
    // as this function returns. If the codec wishes to guarantee access to the data
    // after this function, it must take its own reference that it frees when it is
    // no longer needed.
    Unmarshal(data mem.BufferSlice, v any) error
```

```go
type vtprotoMessage2 interface {
    MarshalToSizedBufferVT(data []byte) (int, error)
    UnmarshalVT([]byte) error
    SizeVT() int
}

// reuse the internal grpc buffer pool
var defaultBufferPool = mem.DefaultBufferPool()

type VTCodecV2 struct{}

func (VTCodecV2) Marshal(v any) (mem.BufferSlice, error) {
    vt, ok := v.(vtprotoMessage2)
    if !ok {
        return nil, fmt.Errorf("failed to marshal, message is %T", v)
    }
    size := vt.SizeVT()
    buf := defaultBufferPool.Get(size)
    if _, err := vt.MarshalToSizedBufferVT((*buf)[:size]); err != nil {
        defaultBufferPool.Put(buf)
        return nil, err
    }
    return mem.BufferSlice{mem.NewBuffer(buf, defaultBufferPool)}, nil
}
```

~ 0.7 CPU core when saturating network on Get

Summary

Workload	Benefit	What to do
Unary request	approx -10% CPU (will increase if proto msg are larger & more complex)	Use vtprotobuf codec
Egress stream	2x reduction in CPU usage	Use a recent grpc version & a CodecV2 implementation that uses memory pools
Ingress stream	2.5x reduction in CPU usage	- Use a recent grpc version or enable internal memory pooling - In the handler: pool received messages

Thanks you !

Aurélien Deroide