GRPC

References

A number of these slides contain content copied directly from the following resources:

- http://www.grpc.io/

What Is RPC?

a `Remote Procedure Call` is when a computer program makes a call to another address space (commonly on another computer) across a network.

Remote Procedure Call Wikipedia

What Is GRPC?

gRPC is a modern open source high performance RPC framework that can run in any environment.

What Is GRPC?

gRPC is a modern open source high performance RPC framework that can run in any environment.

It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

What Is GRPC?

gRPC is a modern open source high performance RPC framework that can run in any environment.

It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

What Is GRPC?

gRPC is a modern open source high performance RPC framework that can run in any environment.

It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

- Allows for an application to directly call methods on a server like it was a local call.

What Is GRPC?

gRPC is a modern open source high performance RPC framework that can run in any environment.

It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

- Allows for an application to directly call methods on a server like it was a local call.
- Is based around the idea of defining a service.

Main Usage Scenarios

Main Usage Scenarios

- Efficiently connecting polyglot services in microservices style architecture

Main Usage Scenarios

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services

Main Usage Scenarios

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries

Core Features

Core Features

- Idiomatic client libraries in 10 languages.

Core Features

- Idiomatic client libraries in 10 languages.
- Highly efficient on wire and with a simple service definition framework.

Core Features

- Idiomatic client libraries in 10 languages.
- Highly efficient on wire and with a simple service definition framework.
- Bi-directional streaming with http/2 based transport.

Core Features

- Idiomatic client libraries in 10 languages.
- Highly efficient on wire and with a simple service definition framework.
- Bi-directional streaming with http/2 based transport.
- Pluggable auth, tracing, load balancing and health checking.

Enterprise Dialing Features

grpc.DialOption

Enterprise Dialing Features

- Built in Authorization

Enterprise Dialing Features

- Built in Authorization
- Back Off Logic

grpc.DialOption

Enterprise Dialing Features

- Built in Authorization
- Back Off Logic
- Balancing Logic available (round robin)

grpc.DialOption

Enterprise Dialing Features

- Built in Authorization
- Back Off Logic
- Balancing Logic available (round robin)
- Supports Compression

grpc.DialOption

Enterprise Dialing Features

- Built in Authorization
- Back Off Logic
- Balancing Logic available (round robin)
- Supports Compression
- Keep Alive

grpc.DialOption

Enterprise Dialing Features

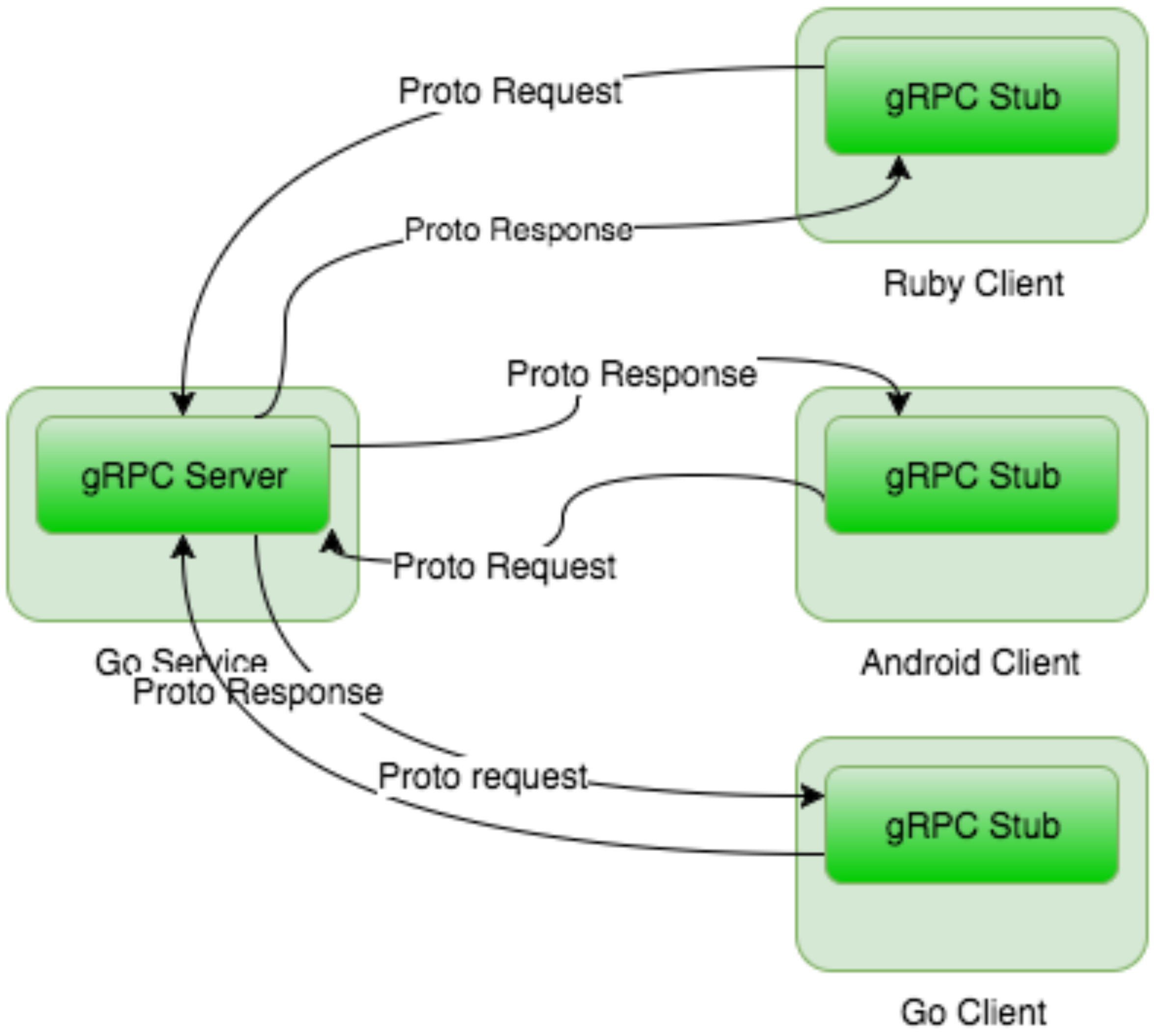
- Built in Authorization
- Back Off Logic
- Balancing Logic available (round robin)
- Supports Compression
- Keep Alive
- Stats Tracking

grpc.DialOption

Simple Service Definition

Uses protocol buffers to define your service.

Works Across Languages And Platforms



GRPC Current Languages

- C++
- C#
- Go
- Java
- Android Java
- Node
- Objective-C
- PHP
- Python
- Ruby

Proto3 Current Languages

- C++
- C#
- Go
- Java
- Android Java
- ~~Node~~
- Objective-C
- ~~PHP~~
- Python
- Ruby

Simple Service Definition

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

Unary RPC's

Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.

```
rpc SayHello(HelloRequest) returns (HelloResponse){}
```

Server Streaming RPC's

Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages.

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){}
```

Client Streaming RPC's

Client streaming RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response.

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {}
```

Bidirectional Streaming RPC's

Bidirectional streaming RPCs are where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse){}
```

Synchronous Vs. Asynchronous

Synchronous RPC calls that block until a response arrives from the server are the closest approximation to the abstraction of a procedure call that RPC aspires to. On the other hand, networks are inherently asynchronous and in many scenarios it’s useful to be able to start RPCs without blocking the current thread.

The gRPC programming surface in most languages comes in both synchronous and asynchronous flavors.

Authentication API

gRPC provides a simple authentication API based around the unified concept of Credentials objects, which can be used when creating an entire gRPC channel or an individual call.

Credential Types

Credentials can be of two types:

- `Channel credentials`, which are attached to a Channel, such as SSL credentials.
- `Call credentials`, which are attached to a call.

Walkthrough

Get The GRPC GO Library

Install the Go gRPC package as well as the protobuf plugin.

```
$ go get google.golang.org/grpc
$ go get github.com/golang/protobuf/protoc-gen-go
```

Simple RPC To Retrieve Website

```
syntax = "proto3";
package hoover;
import "google/protobuf/duration.proto";

service Service {
    rpc Get (GetRequest) returns (GetReply) {}
}

message GetRequest {
    string url = 1;
}

message GetReply {
    int32 response_code = 1;
    string body = 2;
    google.protobuf.Duration elapsed =3;
}
```

Generating The Code

There are two ways we can generate the gRPC code from the protocol buffer definition.

Using the `protoc` Command

```
$ protoc -I hoover/ hoover/hoover.proto --gogo_out=plugins=grpc:hoover
```

Using the `go generate` Command

```
//go:generate protoc --gogo_out=plugins=grpc:. hoover.proto
package hoover
```

Ø

Notice the use of the `plugins=grpc`. This is what tells protoc to generate the gRPC extensions

Server

We can just use an empty struct for this example

```go
type server struct{}
```

Server - Get Method

```go
func (s *server) Get(ctx context.Context, in *hoover.GetRequest) (*hoover.GetReply, error) {
    //TODO: check context to see if we are canceled
    url := in.GetUrl()
    log.Printf("retrieving %s\n", url)

    // start a timer for this request
    begin := time.Now()

    // Retrieve the site
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    elapsed := durationToProtoDuration(time.Since(begin))

    defer resp.Body.Close()
    var body []byte
    if body, err = ioutil.ReadAll(resp.Body); err != nil {
        return nil, err
    }
    log.Printf("finished retrieving %s\n", url)

    return &hoover.GetReply{
        ResponseCode: int32(resp.StatusCode),
        Body:         string(body),
        Elapsed:      elapsed,
    }, nil
}
```

Starting The Server

```go
const port = ":50051"
```

```go
func main() {
    // Get a port to communicate on
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    // create a new grpc server
    s := grpc.NewServer()

    // register our server
    hoover.RegisterServiceServer(s, &server{})

    // start serving
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

Creating The Client

```
address = "localhost:50051"
```
Ø

```
// Set up a connection to the server.
conn, err := grpc.Dial(address, grpc.WithInsecure())
if err != nil {
    log.Fatalf("did not connect: %v", err)
}
// defer the close
defer conn.Close()

// create our service with the connection
c := hoover.NewServiceClient(conn)
```
Ø

Calling Service Methods

```
    // create our service with the connection
    c := hoover.NewServiceClient(conn)

    r, err := c.Get(context.Background(), &hoover.GetRequest{Url: url})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
```

Ø

Display The Result

```
    // print the body of the page retrieved
    fmt.Println(r.Body)

    // print response code and elapsed time
    log.Printf("Response Code: %d, Elapsed: %s", r.ResponseCode, convertDuration(r.Elapsed))
```

Durations

Unfortunately the gPRC known type, `duration.Duration`, does not map correctly to `time.Duration`.

A duration in protobuff is made up of seconds and nanoseconds. To preserve our duration in nanosecond precision, we need to manually do the conversion ourselves.

Duration Conversion Functions

We are using the protobuff well-known type `duration`:

```
    "github.com/golang/protobuf/ptypes/duration"
```

Conversion from `time.Duraton` -> `*duration.Duration`

```
func durationToProtoDuration(d time.Duration) *duration.Duration {
    seconds := int64(d) / int64(time.Second)
    nanos := int64(d) - (seconds * int64(time.Second))
    return &duration.Duration{Seconds: seconds, Nanos: int32(nanos)}
}
```

Conversion from `*duration.Duration` -> `time.Duraton`

```
// convert a protubuff duration to a go duration
func convertDuration(d *duration.Duration) time.Duration {
    return time.Duration((d.Seconds * int64(time.Second)) + int64(d.Nanos))
}
```

Demo

Exercise

Create a gRPC caching service.

- The service should have `Get`, `Put`, and `Delete` functions to manage the service.
- The key should be of type `string`
- The value should be stored as a slice of `bytes`.
- The client should put a key/value into the cache store.
- The client should retrieve a previously set key.
- The client should delete a previously set key.

Solution - Proto Definition

```
syntax = "proto3";
package cachely;

service Cache {
  rpc Get(GetRequest) returns (GetResponse) {}
  rpc Put(PutRequest) returns (PutResponse) {}
  rpc Delete(DeleteRequest) returns (DeleteResponse) {}
}

message GetRequest {
  string key = 1;
}

message GetResponse {
  string key = 1;
  bytes value = 2;
}
```

cachely/app.proto

Solution - Proto Definition (cont)

```
message PutRequest {
  string key = 1;
  bytes value = 2;
}

message PutResponse {
  string key = 1;
}

message DeleteRequest {
  string key = 1;
}

message DeleteResponse {
  string key = 1;
}
```

```
$ protoc -I cachely/ cachely/app.proto --gogo_out=plugins=grpc:cachely
```

cachely/app.proto

Solution - Server

```go
type server struct {
    data syncmap.Map
}
```

Ø

server/main.go

Solution - Get Method

```go
func (s *server) Get(_ context.Context, req *cachely.GetRequest) (*cachely.GetResponse, error) {
    key := req.GetKey()
    log.Printf("looking up key %q\n", key)
    if v, ok := s.data.Load(key); ok {
        log.Printf("found key %q\n", key)
        return &cachely.GetResponse{
            Key:   key,
            Value: v.([]byte),
        }, nil
    }
    log.Printf("key not found %q\n", key)
    return nil, grpc.Errorf(codes.NotFound, "could not find key %s", key)
}
```

server/main.go

Solution - Put Method

```go
func (s *server) Put(_ context.Context, req *cachely.PutRequest) (*cachely.PutResponse, error) {
    log.Printf("storing key %q\n", req.GetKey())
    s.data.Store(req.GetKey(), req.GetValue())
    return &cachely.PutResponse{
        Key: req.GetKey(),
    }, nil
}
```

Ø

server/main.go

Solution - Delete Method

```go
func (s *server) Delete(_ context.Context, req *cachely.DeleteRequest) (*cachely.DeleteResponse, er
    log.Printf("deleting key %q\n", req.GetKey())
    s.data.Delete(req.GetKey())
    return &cachely.DeleteResponse{
        Key: req.GetKey(),
    }, nil
}
```

server/main.go

Solution - Starting The Server

```go
func main() {
	// open a port to communicate on
	lis, err := net.Listen("tcp", ":5051")
	if err != nil {
		log.Fatalf("failed to listen: %v", err)
	}

	// create a new grpc server
	s := grpc.NewServer()

	// register our service
	cachely.RegisterCacheServer(s, &server{
		data: syncmap.Map{},
	})

	// start listening and responding
	if err := s.Serve(lis); err != nil {
		log.Fatalf("failed to serve: %v", err)
	}
}
```

server/main.go

Solution - Client

```go
    // connect to the grpc server
    conn, err := grpc.Dial(":5051", grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }

    defer conn.Close()

    // create a new client
    c := cachely.NewCacheClient(conn)
```

client/main.go

Solution - Set A Value

```go
    // write a value
    _, err = c.Put(ctx, &cachely.PutRequest{
        Key:   "band",
        Value: []byte("The Beatles"),
    })
    if err != nil {
        log.Fatal(err)
    }
```

Ø

client/main.go

Solution - Get A Value

```
    // get a value
    gr, err := c.Get(ctx, &cachely.GetRequest{Key: "band"})
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("band:", string(gr.GetValue()))
```
Ø

client/main.go

Solution - Delete A Value

```go
    // delete a value
    _, err = c.Delete(ctx, &cachely.DeleteRequest{Key: "band"})
    if err != nil {
        log.Fatal(err)
    }

    // check it was deleted
    gr, err = c.Get(ctx, &cachely.GetRequest{Key: "band"})
    if err != nil {
        log.Fatal(err)
    }
```

client/main.go

Syncmap

We used an experimental package called `syncmap`. This removed the need for us to handle race conditions in our code.

sycnmap