<https://grpc.io/>

**gRPC**

A high performance, open-source universal RPC framework

What is gRPC?

gRPC is a modern, open source, high-performance remote procedure call (RPC) framework that can run anywhere. gRPC enables client and server applications to communicate transparently, and simplifies the building of connected systems. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

Why gRPC?

* **Simple Service Definition**

Define your service using Protocol Buffers, a powerful binary serialization toolset and language.

* **Works across languages and platforms**

Automatically generate idiomatic client and server stubs for your service in a variety of languages and platforms.

* **Start quickly and scale**

Install runtime and dev environments with a single line and also scale to millions of RPCs per second with the framework

* **Bi-directional streaming and integrated auth**

Bi-directional streaming and fully integrated pluggable authentication with http/2 based transport

# **gRPC Concepts Overview**

Remote Procedure Calls (RPCs) provide a useful abstraction for building distributed applications and services. The libraries in this repository provide a concrete implementation of the gRPC protocol, layered over HTTP/2. These libraries enable communication between clients and servers using any combination of the supported languages.

## **Interface**

Developers using gRPC start with a language agnostic description of an RPC service (a collection of methods). From this description, gRPC will generate client and server side interfaces in any of the supported languages. The server implements the service interface, which can be remotely invoked by the client interface.

By default, gRPC uses [Protocol Buffers](https://github.com/google/protobuf) as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages.

### **Invoking & handling remote calls**

Starting from an interface definition in a .proto file, gRPC provides Protocol Compiler plugins that generate Client- and Server-side APIs. gRPC users call into these APIs on the Client side and implement the corresponding API on the server side.

#### **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 is desirable to have the ability to start RPCs without blocking the current thread.

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

## **Streaming**

gRPC supports streaming semantics, where either the client or the server (or both) send a stream of messages on a single RPC call. The most general case is Bidirectional Streaming where a single gRPC call establishes a stream in which both the client and the server can send a stream of messages to each other. The streamed messages are delivered in the order they were sent.

# **Protocol**

The [gRPC protocol](https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md) specifies the abstract requirements for communication between clients and servers. A concrete embedding over HTTP/2 completes the picture by fleshing out the details of each of the required operations.

## **Abstract gRPC protocol**

A gRPC call comprises a bidirectional stream of messages, initiated by the client. In the client-to-server direction, this stream begins with a mandatory Call Header, followed by optional Initial-Metadata, followed by zero or more Payload Messages. The server-to-client direction contains an optional Initial-Metadata, followed by zero or more Payload Messages terminated with a mandatory Status and optional Status-Metadata (a.k.a.,Trailing-Metadata).

## **Implementation over HTTP/2**

The abstract protocol defined above is implemented over [HTTP/2](https://http2.github.io/). gRPC bidirectional streams are mapped to HTTP/2 streams. The contents of Call Header and Initial Metadata are sent as HTTP/2 headers and subject to HPACK compression. Payload Messages are serialized into a byte stream of length prefixed gRPC frames which are then fragmented into HTTP/2 frames at the sender and reassembled at the receiver. Status and Trailing-Metadata are sent as HTTP/2 trailing headers (a.k.a., trailers).

## **Flow Control**

gRPC uses the flow control mechanism in HTTP/2. This enables fine-grained control of memory used for buffering in-flight messages.

**Comparison of gRPC with REST APIs**

* <https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711>

Let’s take a glance of gRPC and implement a simple service using Golang.

# **Prerequisites**

* Install Go:[*https://golang.org/doc/install*](https://golang.org/doc/install)
* Install protoc : gRPC uses protobuf to communicate, in order to generate relevant files, you will need to install protoc : *< Protoc installation link >*
* Install protoc-gen-go : as we use go in this post, you will need the go-support for protobuf

*go get -u github.com/golang/protobuf/protoc-gen-go*

* Install grpc : the grpc package for go

*go get -u google.golang.org/grpc*

(Note: make sure your GOPATH is correctly set up so that your packages and project will be under **GOPATH/src** )

Lets create a simple calculator application which can perform sum and subtraction of two input numbers.

1. Under $GOPATH/src/ create a folder calculator

$ *mkdir calculator*

1. Move into calculator folder and create a directory proto and move into it

$ *cd calculator*

*$ mkdir proto*

*$ cd proto*

1. Create the protobuf file called calculation.proto inside proto folder

$ touch calculation.proto

1. Open the calculation.proto file in any editor of choice and copy the following content.

syntax = "proto3";

package calculation\_service;

//Request structure. param1 and param2 are operands to calculate on

message ServiceRequest {

int64 param1 = 1;

int64 param2 = 2;

}

//Response structure, result will hold the output of computation. error\_code will be set in case of error.

message ServiceResponse {

int64 result = 1;

int32 error\_code = 2;

}

service Calculation {

rpc sum(ServiceRequest) returns (ServiceResponse);

//api to add two numbers

rpc sub(ServiceRequest) returns (ServiceResponse);

//api to subtract two numbers

}

The above code represents the interface definition using which we generate stubs and use it in our application. This file is language agnostic. Using corresponding compiler and plugin we can create stubs for respective languages.

For our example we will use golang as our programming language. We create a server application which implements the service we have defined in proto file. Client invokes the corresponding api to fetch results.

1. Move back to project root folder

*$ cd $GOPATH/src/calculator*

1. Compile the proto file using the following command. This creates calculation.pb.go stub file under proto folder

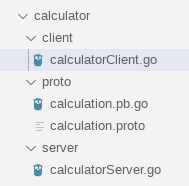
*$ protoc -I proto/ -I${GOPATH}/src --go\_out=plugins=grpc:proto ./proto/calculation.proto*

1. Create two calculatorClient.go and calculatorServer.go files under project root folder.

*$ touch calculatorClient.go*

*$ touch calculatorServer.go*

1. The project hierarchy looks like below.



1. Lets implement **calculatorServer.go** file.

package main

import (

pb "calculator/proto"

"context"

"fmt"

"log"

"net"

"google.golang.org/grpc"

)

type calculationServer struct {

}

func (s \*calculationServer) Sub(ctx context.Context, req \*pb.ServiceRequest) (\*pb.ServiceResponse, error) {

param1 := req.GetParam1()

param2 := req.GetParam2()

return &pb.ServiceResponse{Result: (param1 - param2)}, nil

}

func main() {

//Create a server that listens on port 9000 on all interfaces

port := 9000

lis, err := net.Listen("tcp", fmt.Sprintf(":%d", port))

if err != nil {

log.Fatalf("failed to listen: %v", err)

}

log.Println("Calculator server started at port ", port)

grpcServer := grpc.NewServer()

pb.RegisterCalculationServer(grpcServer, &calculationServer{})

grpcServer.Serve(lis)

}

1. To run server application run command

$ go run calculatorServer.go

1. Lets implement calculatorClient.go

package main

import (

pb "calculator/proto"

"context"

"fmt"

"log"

"os"

"time"

"google.golang.org/grpc"

)

var port int = 9000

func addition() (\*pb.ServiceResponse, error) {

serverAddress := fmt.Sprintf("127.0.0.1:%d", port)

conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(), grpc.WithBlock(), grpc.WithTimeout(time.Second))

if err != nil {

return nil, err

}

client := pb.NewCalculationClient(conn)

ctx, cancel := context.WithTimeout(context.Background(), 3\*time.Second)

defer cancel()

var num1 int64

var num2 int64

fmt.Println("Enter num1: ")

fmt.Scanln(&num1)

fmt.Println("Enter num2: ")

fmt.Scanln(&num2)

result, err := client.Sum(ctx, &pb.ServiceRequest{Param1: num1, Param2: num2})

return result, err

}

func subtraction() (\*pb.ServiceResponse, error) {

serverAddress := fmt.Sprintf("127.0.0.1:%d", port)

conn, err := grpc.Dial(serverAddress, grpc.WithInsecure(), grpc.WithBlock(), grpc.WithTimeout(time.Second))

if err != nil {

return nil, err

}

client := pb.NewCalculationClient(conn)

ctx, cancel := context.WithTimeout(context.Background(), 3\*time.Second)

defer cancel()

var num1 int64

var num2 int64

fmt.Println("Enter num1: ")

fmt.Scanln(&num1)

fmt.Println("Enter num2: ")

fmt.Scanln(&num2)

result, err := client.Sub(ctx, &pb.ServiceRequest{Param1: num1, Param2: num2})

return result, err

}

func main() {

for {

fmt.Println("---------------------------------------------")

fmt.Println("1. Addition")

fmt.Println("2. Subtraction")

fmt.Println("0: exit")

fmt.Println("Enter the required service")

var index int

fmt.Scanln(&index)

if index == 0 {

os.Exit(1)

}

if index == 1 {

result, err := addition()

if err != nil {

log.Println("Error: ", err)

} else {

log.Println("Result: ", result.GetResult())

}

continue

}

if index == 2 {

result, err := subtraction()

if err != nil {

log.Println("Error: ", err)

} else {

log.Println("Result: ", result.GetResult())

}

continue

}

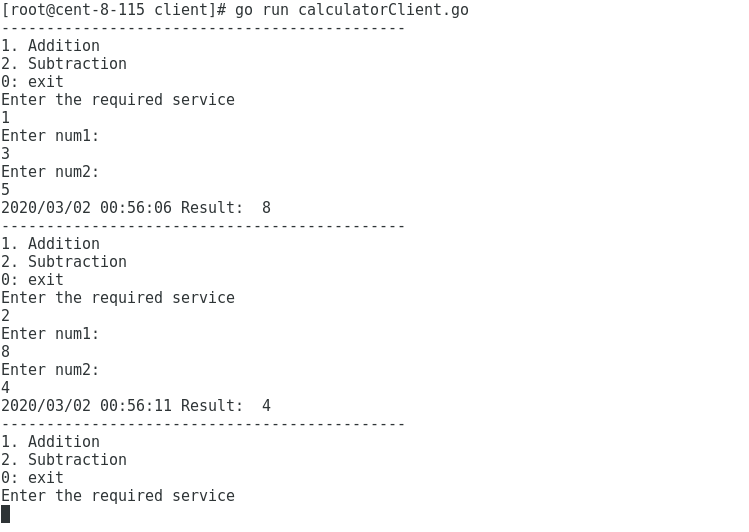
}

}

1. Run calculatorClient.go to run client

*$ go run calculatorClient.go*

Result:



This is the simplest usage of gRPC in golang, and certainly, there are many more other applications, especially that gRPC enables streaming call which allows both client and server to communicate with each other in streaming format. This is just a starting point and there are many more to explore.

**Reference**:

1. <https://grpc.io/docs/guides/>
2. <https://github.com/golang/protobuf>
3. <https://github.com/grpc/grpc-go>
4. <https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711>