gRPC & gRPC-Web: A hands-on oriented introduction

Repo is on GitHub: https://github.com/RFS-0/gRPC.git

Content

- What is gRPC?
- Recap: What is a RPC?
- A quick overview of how gRPC works
- Why gRPC-Web?
- Limitations
- Use case: A simple chat application

What is gRPC (I / II)?

- gRPC is a modern open source high performance RPC framework that can run in any environment
- Stands for "gRPC Remote Procedure Calls"
- Implementations in
  - C++, Go, Node.js, PHP, Java, Ruby, Android Java, Dart, Python, C#, Objective-C, Web

What is gRPC (II / II)?

In short:

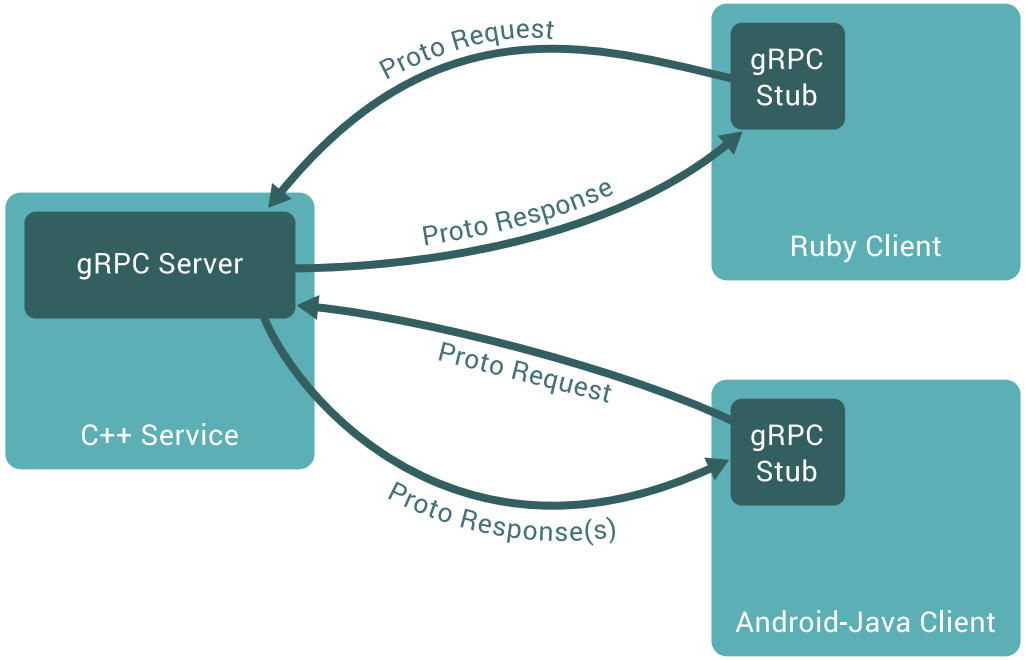
- RPC frame work with focus on performance
  - At Google O(10^10) RPCs per second
- Runs everywhere (Cloud native, Mobile, IoT etc.)
- Next generation of Stubby (used by Google for over a decade)
- Production ready
- Allows you to use Protocol buffers

Recap: What is a RPC?

- RPC = Remote Procedure Calls
- In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.

A quick overview of how gRPC works

- The dark boxes are what gRPC provides:



Why gRPC?

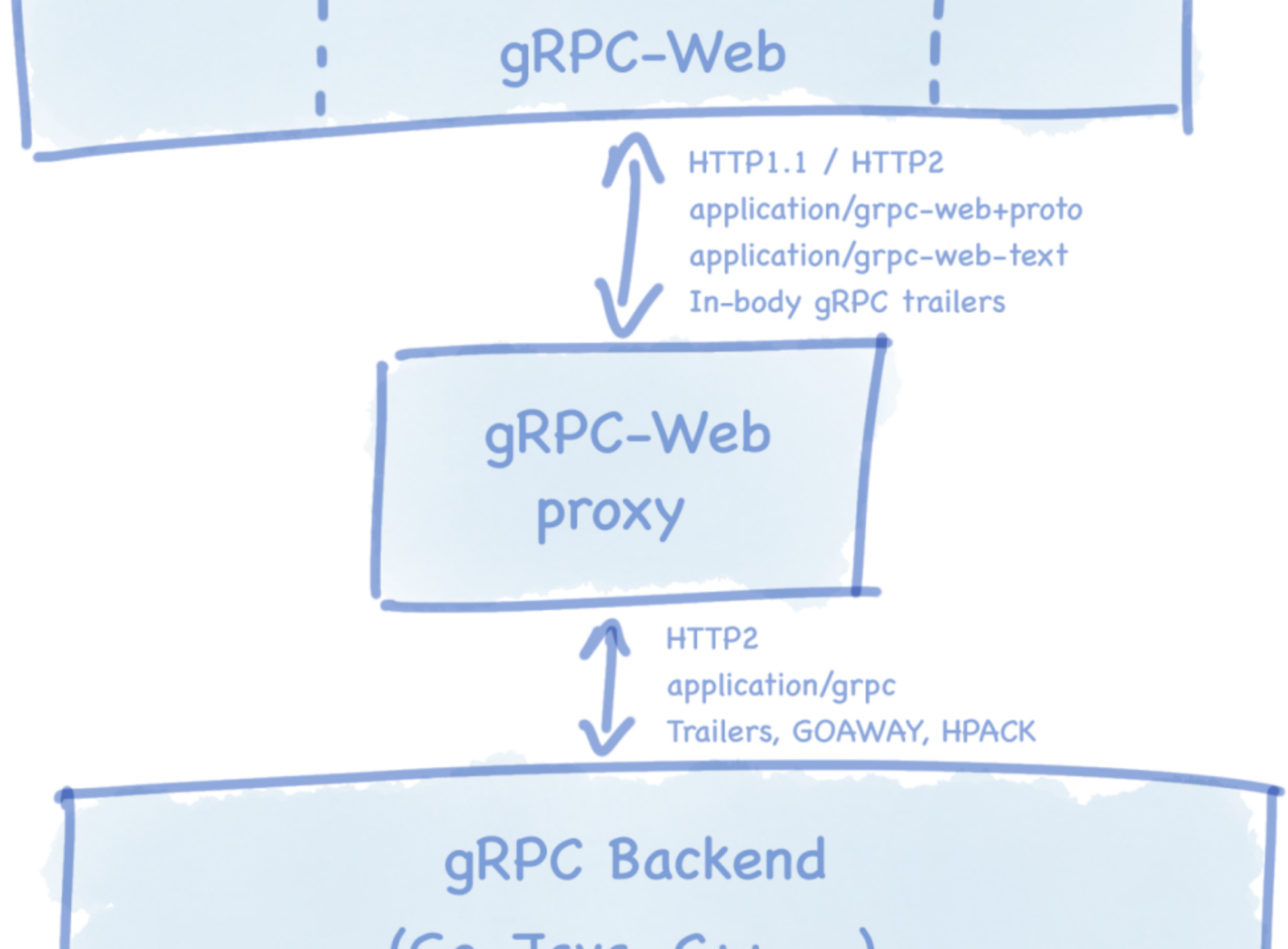
- Simple service definition
- Works across languages and platforms
- Start quickly and scale
- Bi-directional streaming and integrated auth
- Performance

Why gRPC-Web?

- gRPC-Web lets you access gRPC services built in this manner from browsers using an idiomatic API

Limitations of gRPC-Web

- It is currently impossible to implement the HTTP/2 gRPC spec3 in the browser, as there is simply no browser API with enough fine-grained control over the requests.
- None of the current implementations support client-side & bi-directional streaming


gRPC-Web
HTTP1.1 / HTTP2
application/grpc-web+proto
application/grpc-web-text
In-body gRPC trailers
gRPC-Web
proxy
HTTP2
application/grpc
Trailers, GOAWAY, HPACK
gRPC Backend

Use case: A simple chat application

Let's assume we want to implement a simple chat application for the browser using gRPC Web.
This task can be broken down into the following steps:

1. Define types
2. Define a service
3. Use protoc compiler to generate server-side and client-side code
4. Use generated code to implement desired logic
5. Setup proxy
6. Run proxy, server and client

1: Define types

For the chat application we create the following message types:

- EmptyRequest -> used to get all users or messages
- UserRequest -> used to create a user
- UserResponse -> represents a created user
- UserResponseList -> used to hold all created users
- MessageRequest -> used to create a message
- MessageResponse -> represent a created message
- MessageResponseList -> used to hold all created messages

2: Define service (I/III)

Like many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types.

2: Define service (II/III)

Inside the service definition we define the rpc methods we want to use. There are four kinds of service methods:

1. simple -> client sends request to server and waits for single response
   `rpc SayHello(HelloRequest) returns (HelloResponse) {}`
2. server-side streaming -> client sends a request to the server and gets a stream to read a sequence of messages back
   `rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse) {}`
3. client-side streaming -> client writes a sequence of messages and sends them to the server, again using a provided stream
   `rpc BidiHello(stream HelloRequest) returns (stream HelloResponse) {}`
4. bidirectional streaming -> both sides send a sequence of messages using a read-write stream
   `rpc BidiHello(stream HelloRequest) returns (stream HelloResponse) {}`

2: Define service (III/III)

For the chat application we use the following service definition:

- ChatService
  - createUser
  - getAllUsers
  - users
  - createMessage
  - getAllMessages
  - messages

3: Use protoc compiler to generate server-side and client-side code

- See `build.gradle` in `chat-server`
- See `README.md` in `protobuffers`

4. Use generated code to implement desired logic

- See `ChatService`, `ChatServer` and `ChatClient` in `chat-server`
- See `chat-client-service-impl.service.ts` in `chat-client`

5. Setup proxy

- See `README.md` in `proxy`

6. Run proxy, server and client

- See `README.md` in `proxy`
- Run `main` in `ChatServer`
- Run `ng serve` in `chat-client`