gRPC:
A multi-platform RPC system

Microservices at Google

~O(10^{10}) RPCs per second.

Images by Connie Zhou

gRPC core ★ Star 4,235 Fork 714

gRPC java ★ Star 1,153 Fork 299

GRPC

http://grpc.io

Open source on Github for C, C++, Java, Node.js, Python, Ruby, Go, C#, PHP, Objective-C

OVERVIEW

gRPC is ...

Open Source RPC framework that makes it ***easy*** to build a heterogenous distributed system.

- Free as in beer! (and licensing)
- Based on HTTP/2 today (multiplexed, works with the Internet)
- Payload agnostic (we’ve implemented proto)
- Streaming & Flow-Controlled
- Designed for harsh environments (timeout, lameducking, load-balancing, cancellation, …)
- Support in 10 languages & first class mobile support
- Layered & Pluggable - Bring your own monitoring, auth, naming, load balancing ...

Project Status

- Core features and protocol are fully specified
- Rolled out for public Google APIs and widely used internally
  - Lots of mobile adoption
- Approaching 1.0 (GA) release in all languages
  - Stable APIs for key features
- Benefit of layering on top of HTTP/2 standard
  - Interoperability with 3rd party proxies, tools, libraries..
  - WHATWG Fetch

Multiple Languages

C/C++

python™

Java™

php

C#

node js™

http://www.http2demo.io/

HTTP/1.1

HTTP/2

Protocol Buffers

IDL (Interface definition language)

Describe once and generate interfaces for any language.

Data Model

Structure of the request and response.

Wire Format

Binary format for network transmission.

```
message SubscribeRequest {
  string topic = 1;
}

message Event {
  string details = 1;
}

service Topics {
  rpc Subscribe(SubscribeRequest)
returns (stream Event);
}
```

Implementation Details

- Three complete stacks: C/C++, Java and Go.
- Other language implementations wrap C-Runtime libraries.
  - Hand-written wrappers to maintain language idioms
- Why wrap C?
  - Development costs & Implementation Consistency
  - Performance
  - Feature evolution
- Easy one line installation via packages e.g *npm install grpc*

USE CASES

Use Cases

Build distributed applications

- In data-centers
- In public/private cloud

Client-server communication

- Clients and servers across:
  - Mobile
  - Web
  - Cloud
- Also
  - Embedded systems, IoT

Access Google Cloud Services

- From GCP
- From Android and iOS devices
- From everywhere else

Images by Connie Zhou

HOW TO GET STARTED

Typical development workflow

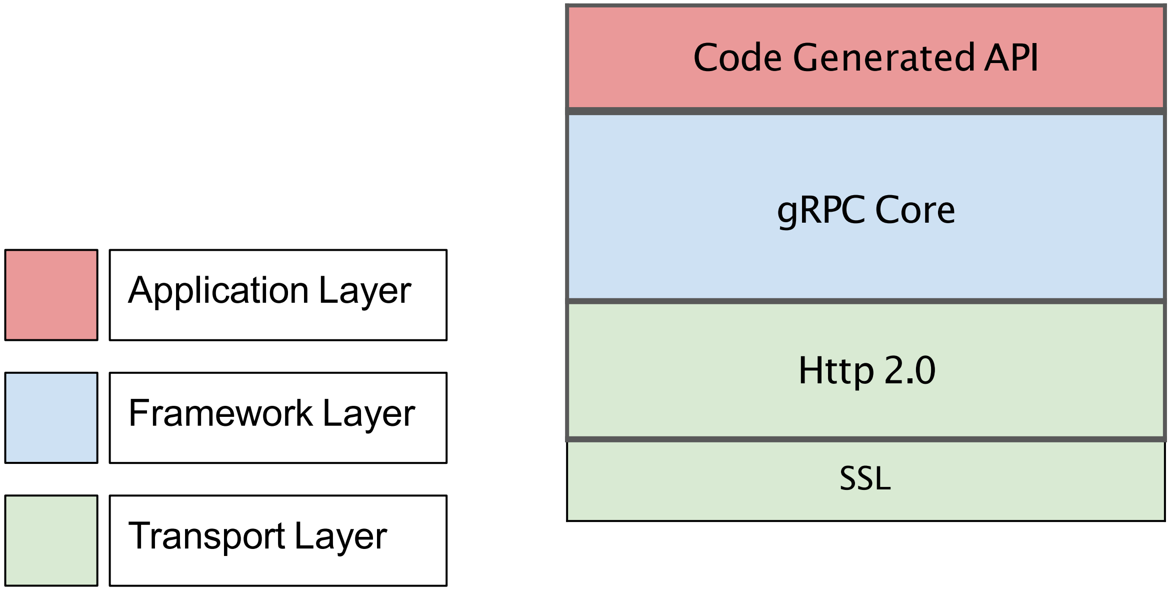
- Install
  - apt-get install protobuf-compiler
  - pip install grpcio
- Write the protos
- Use protoc to generate service interfaces, messages & stubs
- Implement services in server
- Client instantiates stub
- Test & Deploy

Advanced Deployment...

- Auth & Security - TLS [Mutual], Plugin auth mechanism (e.g. OAuth)
- Proxies - nghttp2, haproxy, Google LB, Nginx (in progress)
- Client-side load balancing - etcd, Zookeeper, Eureka, …
- Monitor & Trace - Zipkin, Google, DIY
- Mobile - Reconnect, QUIC
- Web - REST Adapter, WHATWG Fetch
- API Evolution - Protobuf, Versioning

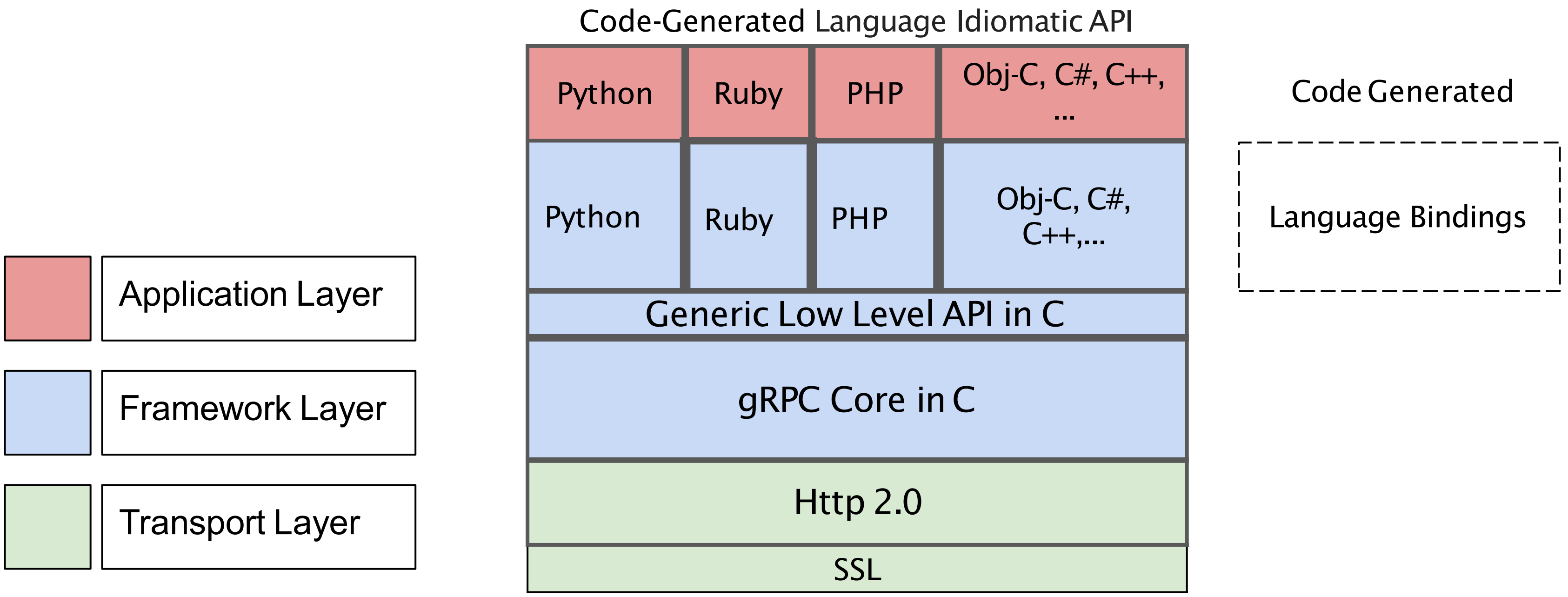
ARCHITECTURE

Architecture: Native Implementation in Language

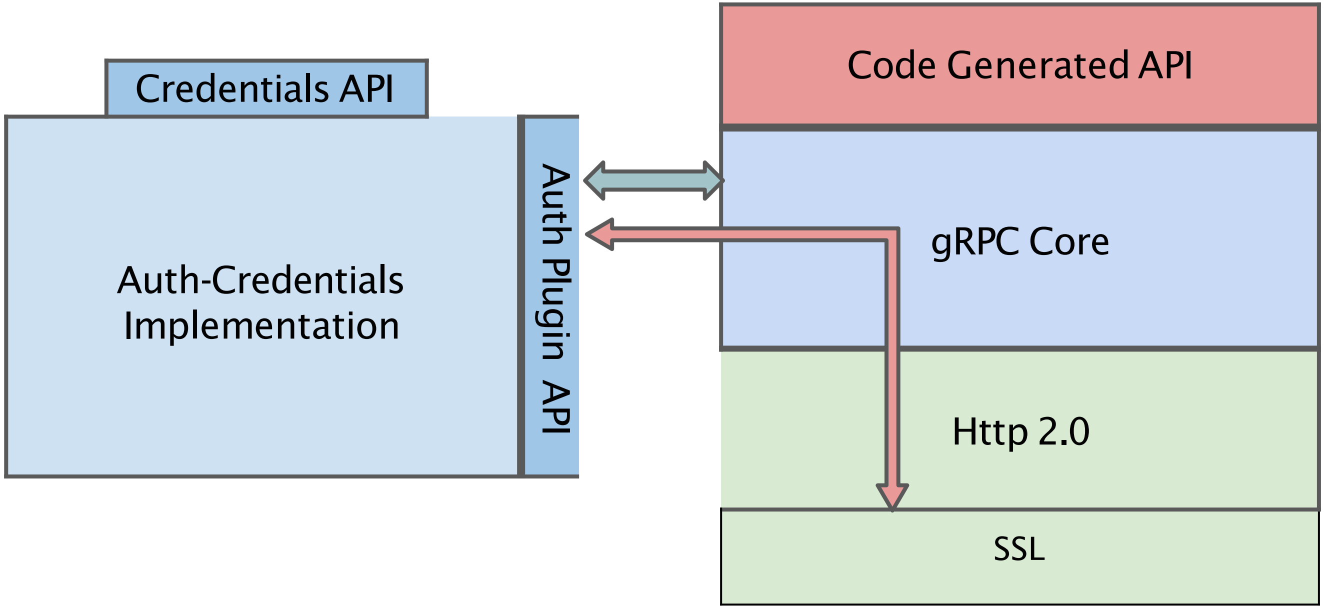


Planned in:
C/C++, Java, Go

Architecture: Derived Stack



Auth Architecture and API

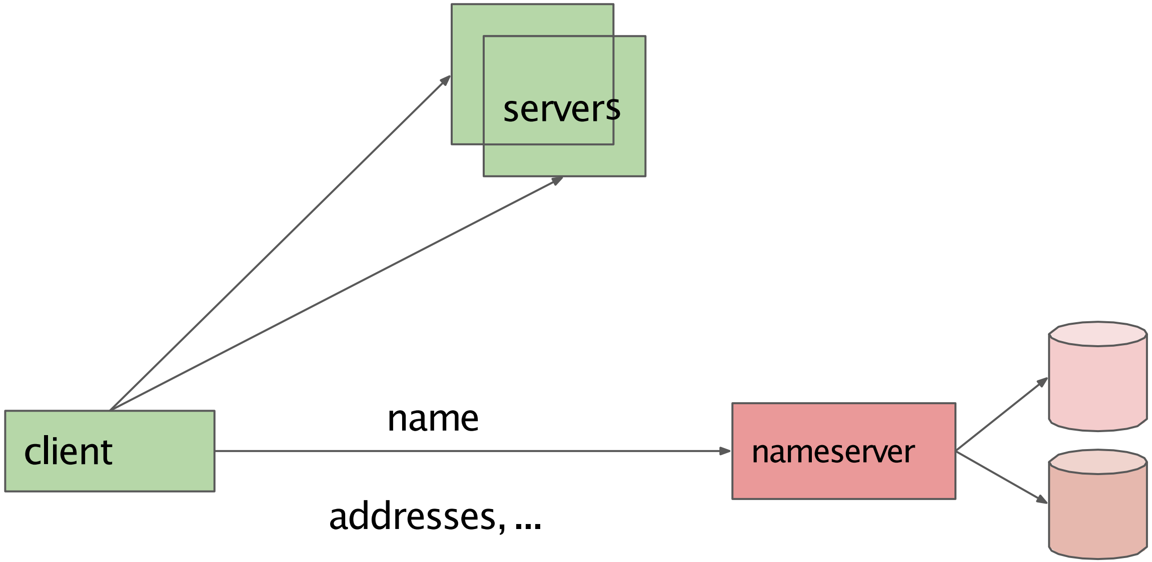


Metadata and Auth

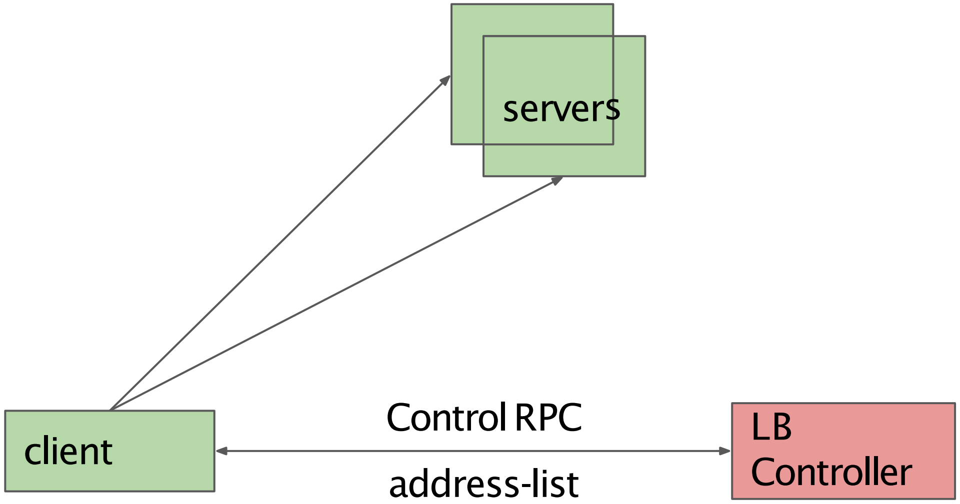
- Generic mechanism for attaching metadata to requests and responses
- Built into the gRPC protocol - always available
- Plugin API to attach “bearer tokens” to requests for Auth
  - OAuth2 access tokens
  - OIDC Id Tokens
- Session state for specific Auth mechanisms is encapsulated in an Auth-credentials object

ADVANCED FEATURES

gRPC: Naming



gRPC: LoadBalancing



Roadmap: Timeline

Initial Alpha Release

Additional Alpha Releases

Beta Release

Stability, Easy installation, 1.0 release

Load balancing, Naming, Performance

Debugging and Tracing, Browser support

Q1 ‘15

Q2 ‘15

Q3 ‘15, Q4’15

Q1 ‘16

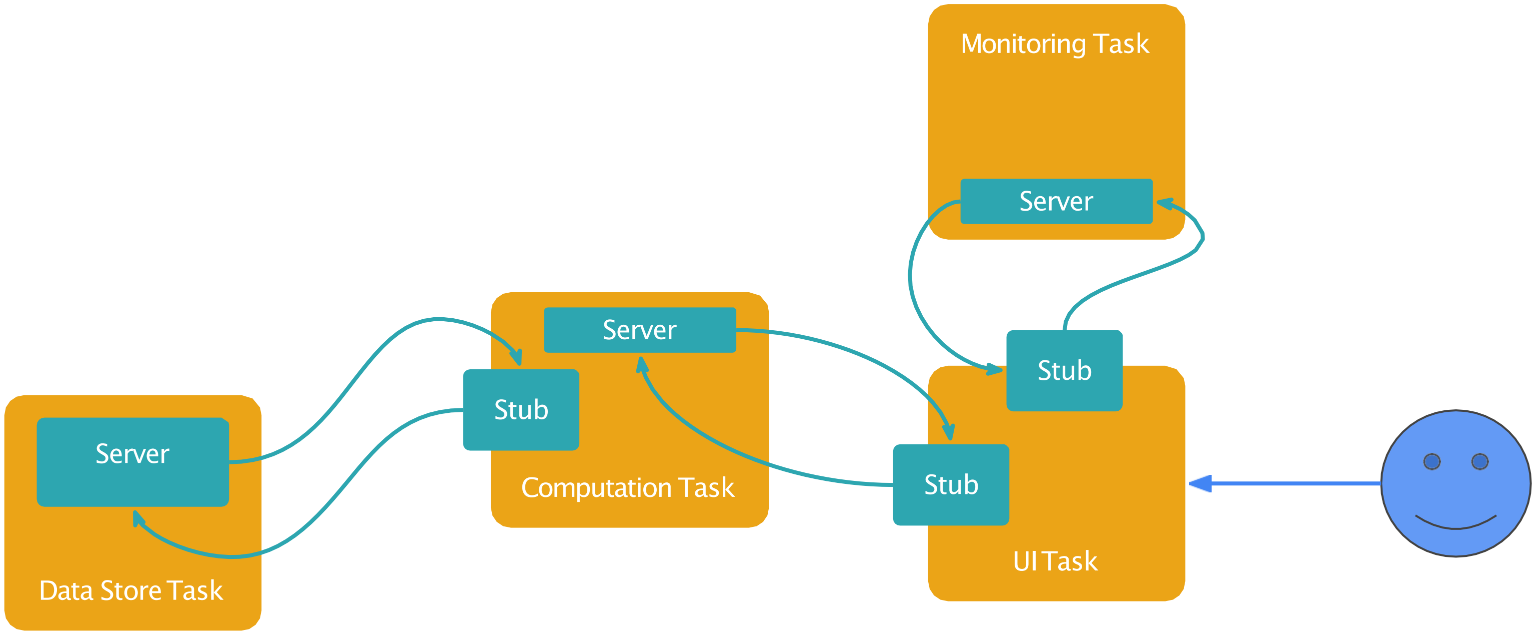
Q2 ‘16

Q3 16

Motivation for RPC systems

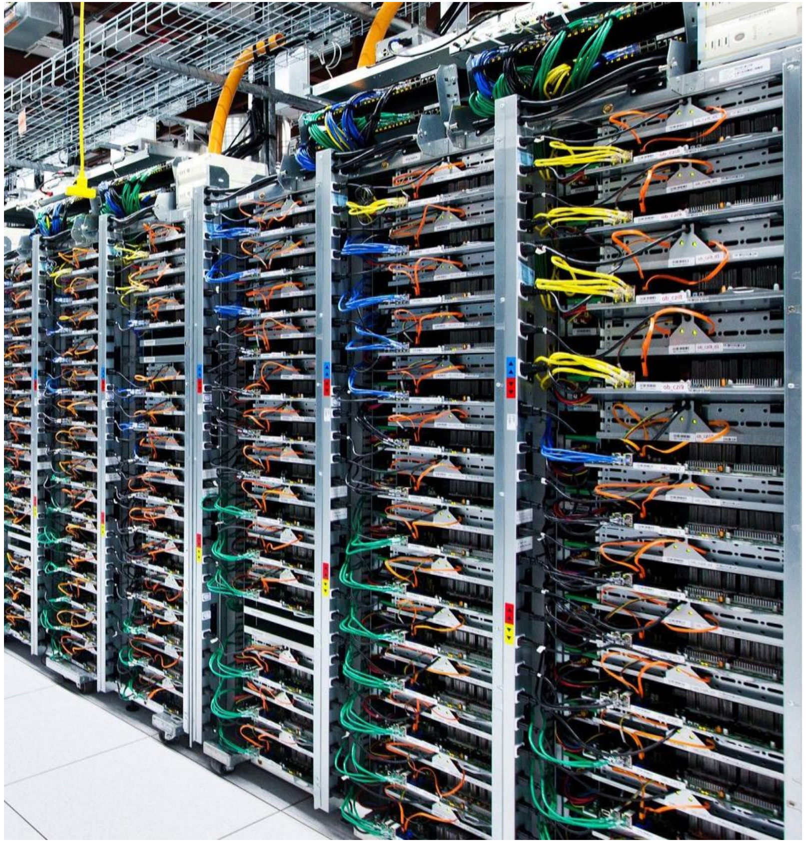
- **Large-scale distributed systems actually composed of microservices**
  - Allows loosely-coupled and even multilingual development
  - Scalability: things, cores, devices, nodes, clusters, and data centers (DCs)
- **Communication predominantly structured as RPCs**
  - Many models of RPC communication
  - Terminology: *Client* uses a *stub* to *call* a *method* running on a *service/server*
  - Easiest interfaces (synchronous, unary) resemble local procedure calls translated to network activity by code generator and RPC library
  - High-performance interfaces (async, streaming) look like Active Messaging
- **Long way from textbook description of RPCs!**

Application composed of microservices



gRPC: Motivation

- **Google has had 4 generations of internal RPC systems, called Stubby**
  - All production applications and systems built using RPCs
  - Over 10^{10} RPCs per second, fleetwide
  - APIs for C++, Java, Python, Go
  - Not suitable for open-source community! (Tight coupling with internal tools)
- **Apply scalability, performance, and API lessons to external open-source**



gRPC: Summary

- **Multi-language, multi-platform framework**
  - Native implementations in C, Java, and Go
  - C stack wrapped by C++, C#, Node, ObjC, Python, Ruby, PHP
  - Platforms supported: Linux, Android, iOS, MacOS, Windows
  - *This talk:* focus on C++ API and implementation (designed for performance)
- **Transport over HTTP/2 + TLS**
  - Leverage existing network protocols and infrastructure
  - Efficient use of TCP - 1 connection shared across concurrent framed streams
  - Native support for secure bidirectional streaming
- **C/C++ implementation goals**
  - High throughput and scalability, low latency
  - Minimal external dependencies

Using HTTP/2 as a transport

Using an HTTP transport: Why and How

- **Network infrastructure well-designed to support HTTP**
  - Firewalls, load balancers, encryption, authentication, compression, ...
- **Basic idea: treat RPCs as references to HTTP objects**
  - Encode request method name as URI
  - Encode request parameters as content
  - Encode return value in HTTP response

```
POST /upload HTTP/1.1
Host: www.example.org
Content-Type: application/json
Content-Length: 15

{"msg":"hello"}
```

Using an HTTP/1.1 transport and its limitations

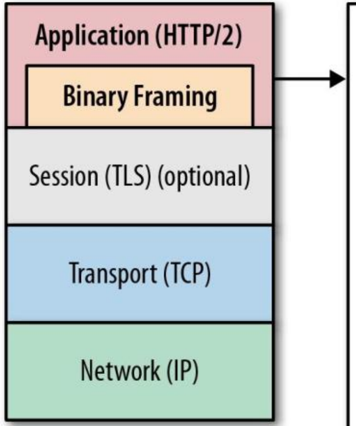
- **Request-Response protocol**
  - Each connection supports pipelining
  - …but not parallelism (in-order only)
  - Need multiple connections per client-server pair to avoid in-order stalls across multiple requests → multiple CPU-intensive TLS handshakes, higher memory footprint
- **Content may be compressed**
  - …but headers are text format
- **Naturally supports single-direction streaming**
  - …but not bidirectional

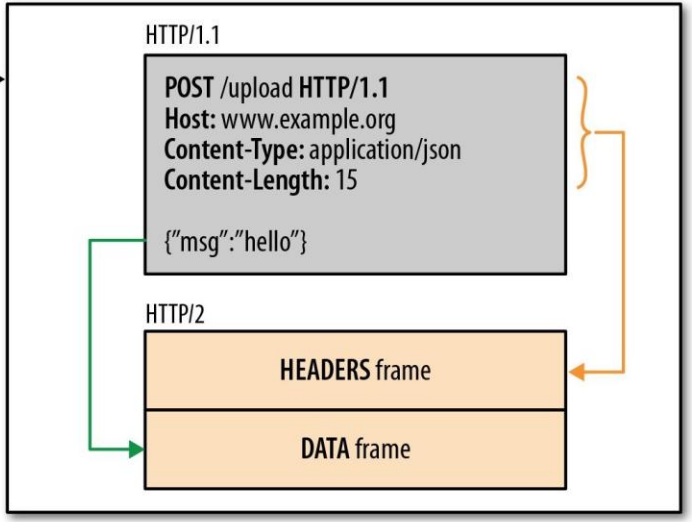
```
POST /upload HTTP/1.1
Host: www.example.org
Content-Type: application/json
Content-Length: 15

{"msg":"hello"}
```

HTTP/2 in a Nutshell

- **One TCP connection for each client-server pair**
- **Request → Stream**
  - Streams are multiplexed using framing
- **Compact binary framing layer**
  - Prioritization
  - Flow control
  - Server push
- **Header compression**
- **Directly supports bidirectional streaming**





http://www.http2demo.io/

HTTP/1.1

HTTP/2

gRPC

gRPC in a nutshell

- IDL to describe service API
- Automatically generates client stubs and abstract server classes in 10+ languages
- Takes advantage of feature set of HTTP/2

An Aside: Protocol Buffers

- Google’s Lingua Franca for serializing data: RPCs and storage
- Binary data representation
- Structures can be extended and maintain backward compatibility
- Code generators for many languages
- Strongly typed
- Not required for gRPC, but very handy

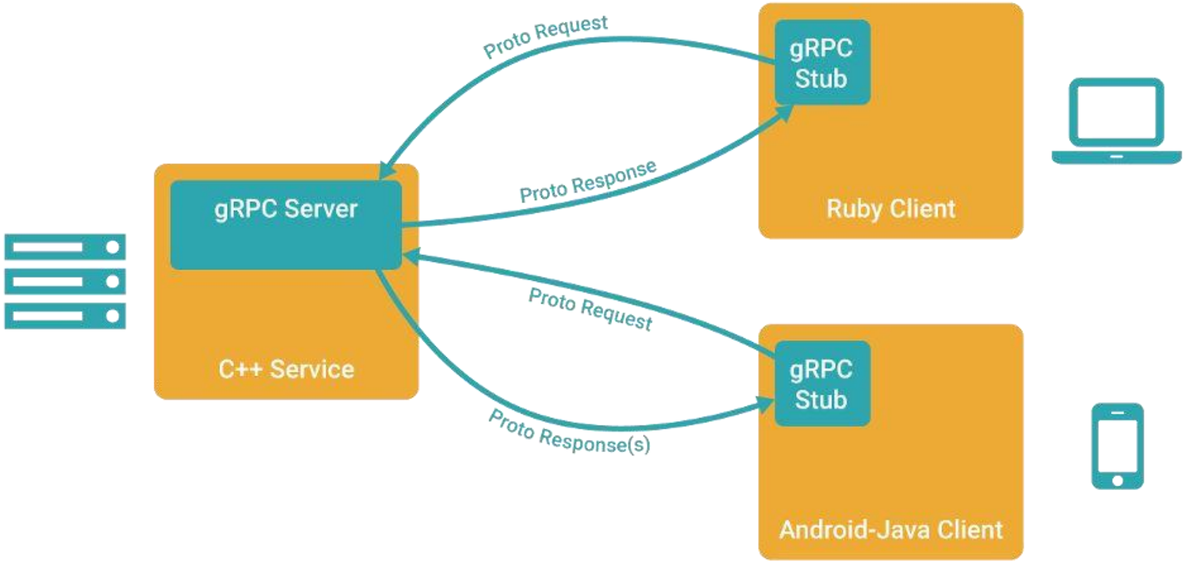
```
syntax = "proto3";

message Person {
  string name =
  1;  int32 id =
  2;  string
  email = 3;

  enum
    PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message
    PhoneNumber {
    string number =
    1;  PhoneType
    type = 2;
  }
```

Example gRPC client/server architecture



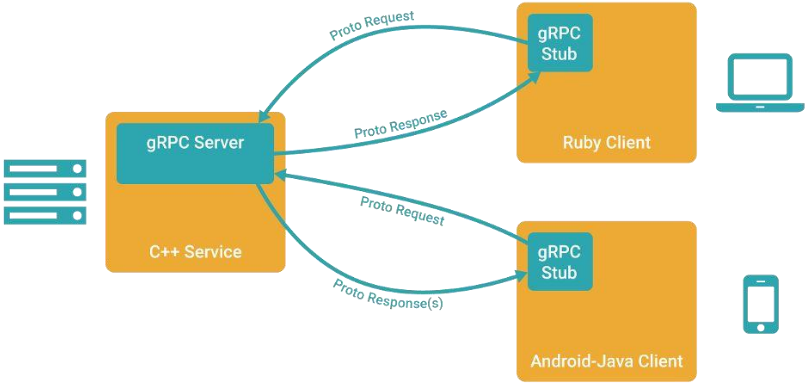
Getting Started

Define a service in a .proto file using Protocol Buffers IDL

Generate server and client stub code using the protocol buffer compiler

Extend the generated server class in your language to fill in the logic of your service

Invoke it using the generated client stubs



Example Service Definition

```
service RouteGuide {
  rpc GetFeature(Point) returns (Feature);
  rpc RouteChat(stream RouteNote) returns (stream RouteNote);
}
```

```
message Point {
  int32 Latitude = 1;
  int32 Longitude = 2;
}
```

```
message Feature {
  string name = 1;
  Point location = 2;
}
```

```
message RouteNote {
  Point location = 1;
  string message = 2;
}
```

An (anonymized) case study

- **Service needs to support bidirectional streaming with clients**
- **Attempt 1: Directly use TCP sockets**
  - Functional in production data center, but not on Internet (firewalls, etc)
  - Programmer responsible for all network management and data transfer
- **Attempt 2: JSON-based RPC over two HTTP/1.1 connections**
  - Start two: one for request streaming and one for response streaming
  - But they might end up load-balanced to different back-end servers, so the backing servers require shared state
  - Can only support 1 streaming RPC at a time for a client-server pair
- **Attempt 3: gRPC**
  - Natural fit

Authentication

SSL/TLS

gRPC has SSL/TLS integration and promotes the use of SSL/TLS to authenticate the server, and encrypt all the data exchanged between the client and the server. Optional mechanisms are available for clients to provide certificates to accomplish mutual authentication.

OAuth 2.0

gRPC provides a generic mechanism to attach metadata to requests and responses. Can be used to attach OAuth 2.0 Access Tokens to RPCs being made at a client.

Wrap-up