gRPC Design and Implementation

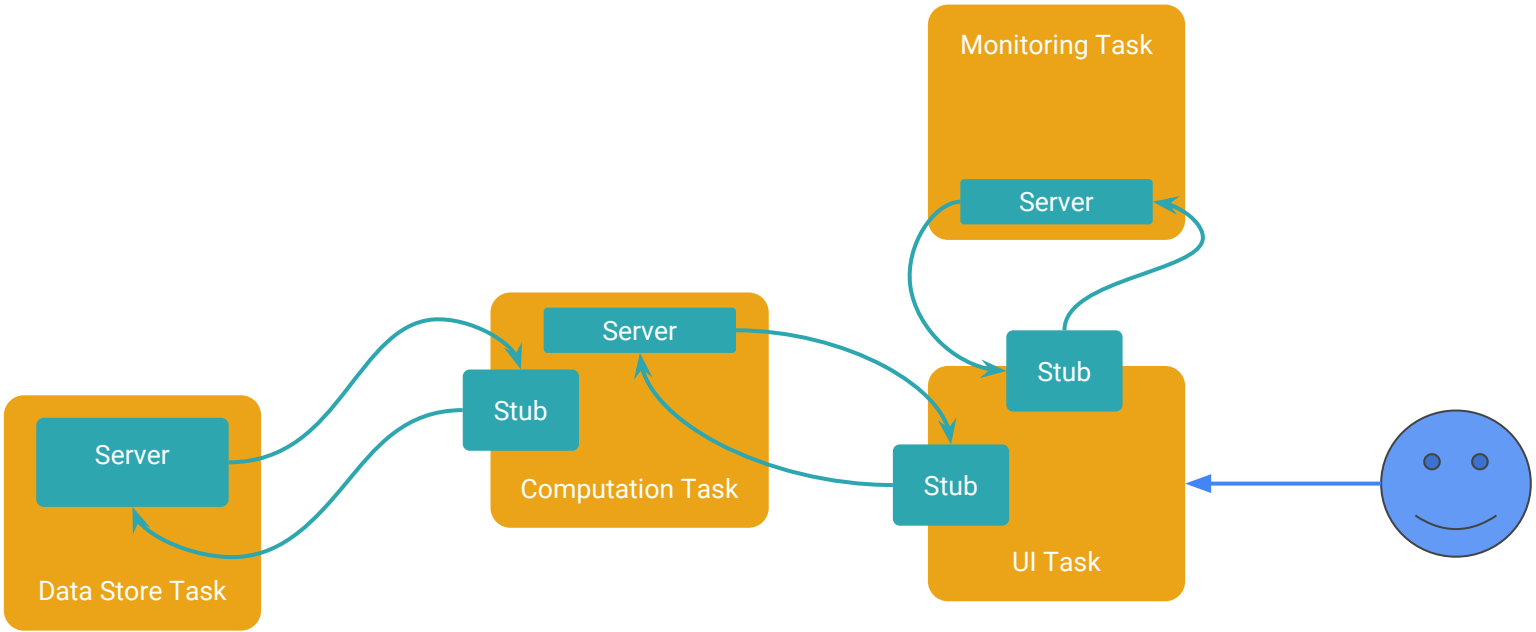
Stanford Platform Lab Seminar - April 2016

Vijay Pai - gRPC Software Engineer

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"}
```

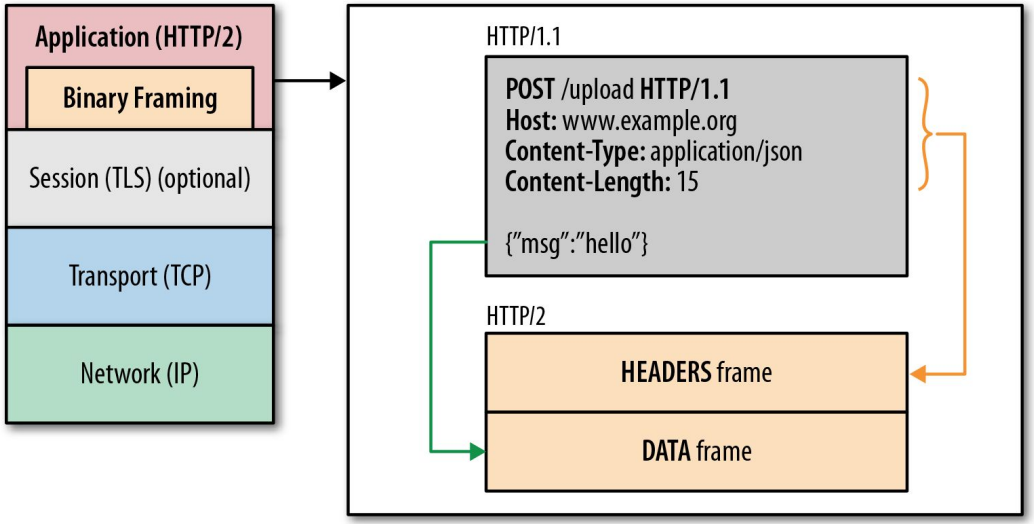
Using an HTTP/1.1 transport and its limitations

- **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**
- **Neat demo at http2demo.io**



gRPC

gRPC in a nutshell

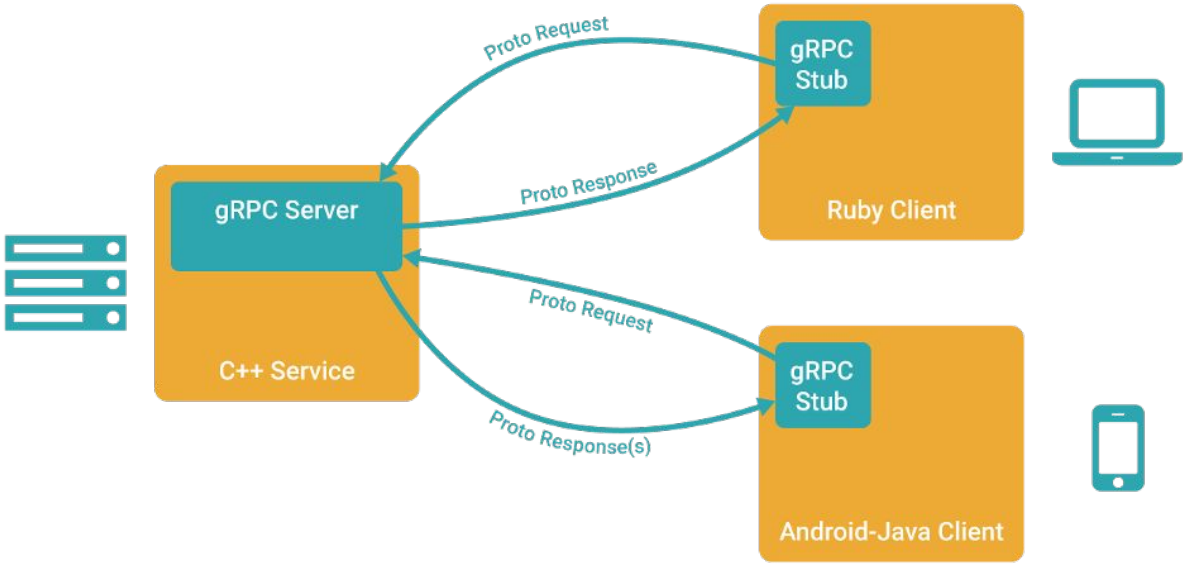
- IDL to describe service API
- Automatically generates client stubs and abstract server classes in 10+ languages
- Takes advantage of feature set of HTTP/2

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;
  }

  repeated PhoneNumber phone = 4;
}
```

Example gRPC client/server architecture



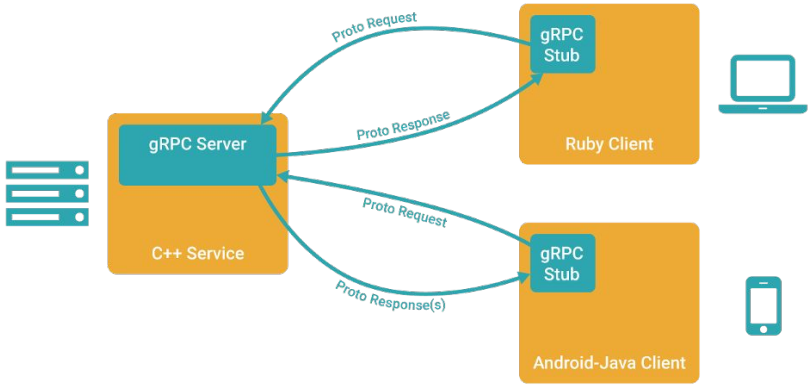
Getting Started

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;
}
```

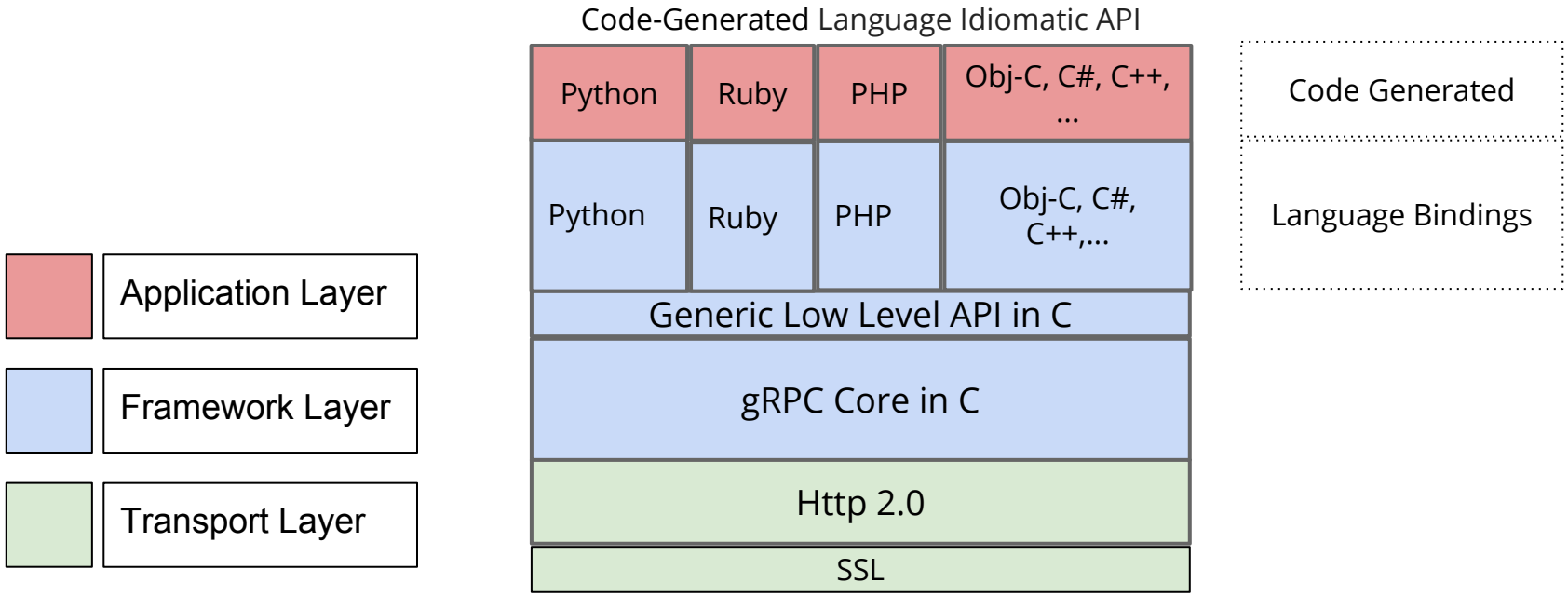
An (anonymized) case study

- **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

Overview of C++ implementation and performance

Google Cloud Platform

gRPC C Core and Wrapped Language Stack



Role of C Core

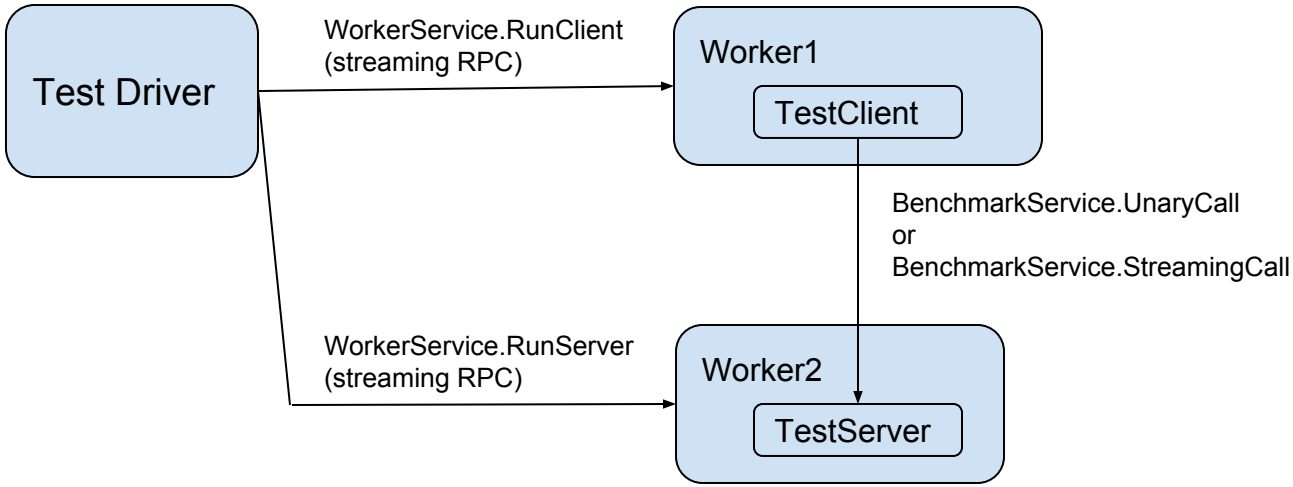
- **Includes a full HTTP/2 transport implementation with authentication, etc**
  - Philosophy of not using external dependences
- **Manages interactions with system**
  - Receives asynchronous event notifications (socket events, timers)
  - Can use generic poll, or platform-specific mechanisms like epoll and IOCP
- **Provides interface to wrapped-language stack**
  - Completion queue as an abstraction of system-level event notifications
    - Tied to RPC semantics, not just bytes on sockets
- **Not intended as an application-level API!**
  - No support for protobufs
- **Core uses application threads to do its work***
  - Let them call into the library and then do the library work there

* Except for DNS resolver, but that doesn't count

C++ API basics and threading model

- **Synchronous API**
  - Client - block waiting for response (unary) or read (streaming)
  - Server - send each incoming RPC to a gRPC-managed dynamically-sized thread-pool for processing. This is the only API where gRPC C/C++ uses its own worker threads to support RPCs
- **Asynchronous API**
  - Client or server threads can use a `CompletionQueue` to wait for events (read/write/response notifications, alarms).
- **Can be mixed between client/server, or even among different methods in same server**

gRPC performance testing

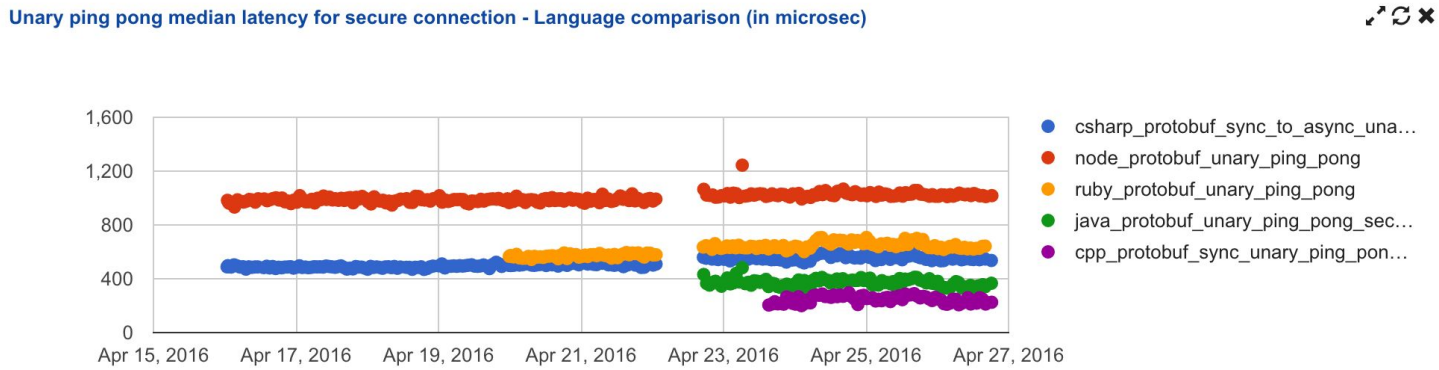


Structure of gRPC multi-language, multi-platform performance testing tools

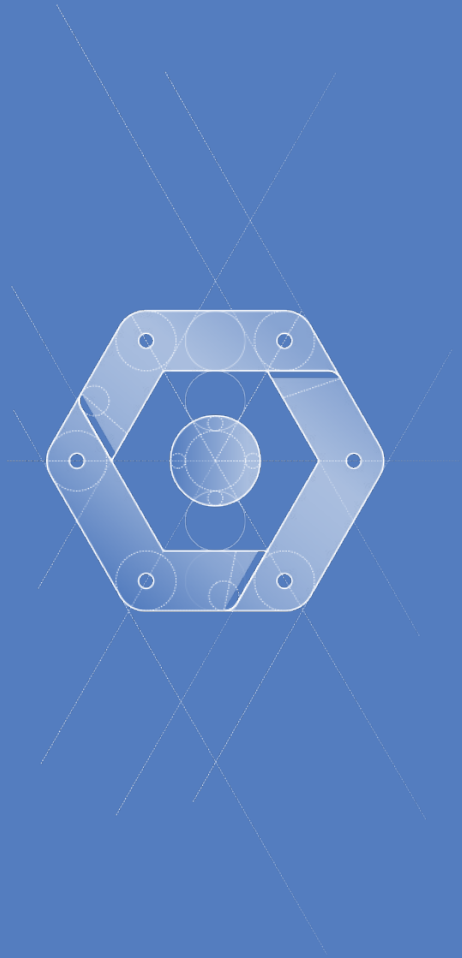
- The workers speak the same proto API regardless of what language (or API) they are implemented in, so useful for interop testing

Current performance metrics of interest

- **Contentionless latency today: 54 µs localhost, 135 µs in cloud**
- **32-core server throughput: 350,000 messages per second**
- **Performance dashboard will soon be linked and visible at grpc.io**
  - Continuous testing to avoid regressions and breakages
  - Easy comparison across languages



Wrap-up



grpc is **Open Source**

We welcome your help!

http://grpc.io/contribute

https://github.com/grpc

irc.freenode.net *#grpc*

@grpcio

grpc-io@googlegroups.com

Occasional public meetups with free pizza