gRPC Introduction

Today's trend is to build microservices

- These microservices must exchange information and need to agree on:
  - The API to exchange data
  - The data format
  - The error patterns
  - Load Balancing
  - Many other

Things need to consider when building an API

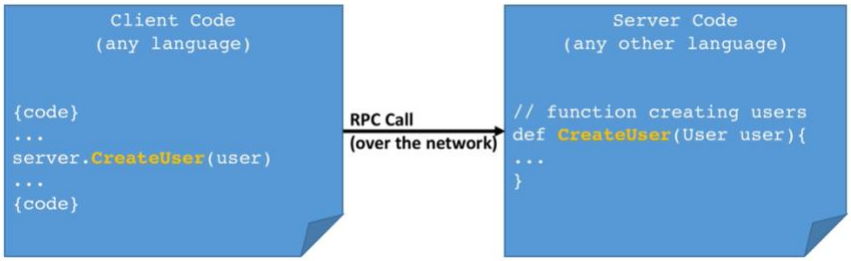
- Need to think about efficiency of the API
  - How much data do I get out of one call?
  - Too much data
  - Too little data → many API calls?
- How about latency?
- How about scalability to 1000s of clients?
- How about load balancing?
- How about inter operability with many languages?
- How about authentication, monitoring, logging?

What is gRPC

- Part of Cloud Native Computation Foundation
- Allows you to define REQUEST and RESPONSE for RPC and handles all the rest for you
- Modern, fast & efficient, build on top of HTTP/2, low latency, supports streaming, language independent, and makes it super easy to plug in authentication, load balancing, logging and monitoring

What's an RPC

- A remote procedure call.
- In the client code, it looks like you're just calling a function directly on the SERVER



How to get started

- At the core of gRPC, you need to define the messages and services using Protocol Buffer.
- The rest of gRPC code will be generated for you and you'll have to provide an implementation for it
- One .proto file works for over 12 programming languages (server and client), and allows you to use a framework that scales to millions of RPC per seconds

Why Protocol Buffers

- Are language agnostic

- Code can be generated for pretty much any language
- Data is binary and efficiently serialized (small payloads)
- Very convenient for transporting a lot of data
- Protocol Buffers allows for easy API evolution using rules

gRPC Deep Dive

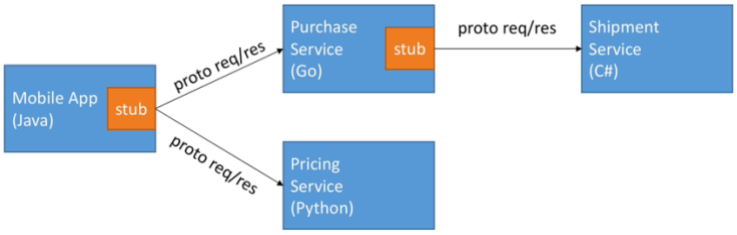
Protocol Buffers & Language Interoperability

Protocol Buffers

- Messages
  - Data Request and Response
- Server
  - Service name and RPC endpoints
  - Allows gRPC to generate code for us
- Efficiency of protocol buffers over JSON
  - Save in network bandwidth
  - Parsing JSON is CPU intensive
  - Parsing Protocol Buffers is less CPU intensive (binary code)
- Why
  - Easy to write message definition
  - The definition of the API is independent from the implementation
  - A huge amount of code can be generated, in any language, from a simple .proto file
  - The payload is binary
  - Protocol buffers defines rules to make an API evolve without breaking existing clients, which is helpful for microservices

gRPC languages

- 3 main implementations
  - JAVA, GO
  - C
  - C++, Python, Ruby, Objective C, PHP, C# reply on C
- The code can be generated for any language, it makes it super simple to create micro-services in any language that interact with each other
  - Stub



HTTP/2

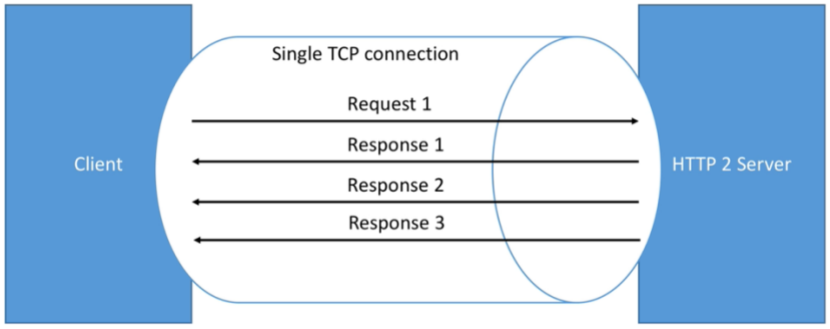
How HTTP/1.1 works

- Released in 1997.
- Opens a new TCP connection to a server at each request
- Doesn't compress headers

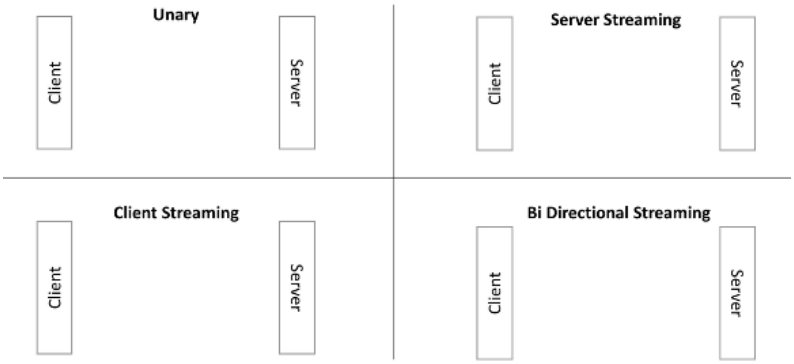
- Only works with Request/Response mechanism (no server push)
- Originally composed of two commands (GET, POST)
- Headers are sent at every request and are PLAAINTEXT
- Overall, add latency and increase network packet size

How HTTP/2 works

- Released in 2015
- Supports multiplexing
    - The client & server can push messages in parallel over the same TCP connection
    - Greatly reduces latency
- Supports server push
    - Servers can push streams (multiple messages) for one request from the client
        - Saves round trips (latency)
- Supports header compression
    - Headers (text based) can now be compressed
    - These have much less impact on the packet size
- Is binary
    - While HTTP/1 text makes it easy for debugging, it's not efficient over the network
    - Protocol buffers is a binary protocol and makes it a great match for HTTP2
- Is secure
    - SSL is not required but recommended by default
- Overall
    - Less chatter
    - More efficient protocol (less bandwidth)
    - Reduced latency
    - Increased security

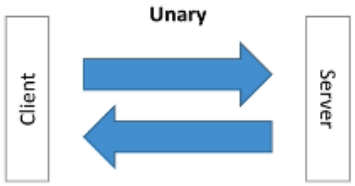


4 types of gRPC APIs



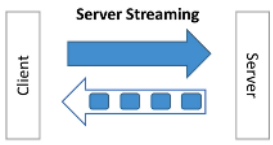
Unary

- the client sends a single request and gets back a single response



Server Streaming

- similar to a unary RPC, except that the server returns a stream of messages in response to a client's request.

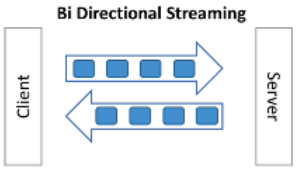


Client Streaming

- similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message.

Bidirectional streaming

- the call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline. The server can choose to send back its initial metadata or wait for the client to start streaming messages.
- Client- and server-side stream processing is application specific
    - the client and server can read and write messages in any order



Scalability in gRPC

- gRPC servers are asynchronous by default
    - don't block threads on request
- gRPC Clients can be asynchronous or synchronous
    - The client decides which model works best for the performance needs
- gRPC Clients can perform client side load balancing

Security in gRPC

- By default gRPC strongly advocates for you to use SSL (encryption over the wire) in your API.
    - gRPC has security as a first class citizen
- each language will provide an API to load gRPC with the required certificates and provide encryption capability out of the box
- can provide authentication with Interceptors

gRPC vs REST

gRPC	REST
Protocol Buffers – smaller, faster	JSON – text based, slower, bigger
HTTP/2 (lower latency) – from 2015	HTTP1.1 (higher latency) – from 1997
Bidirectional & Async	Client => Server request only
Stream support	Request/Response support only
API Oriented – "What" (no constraints – free design)	CRUD Oriented
Code Generation through Protocol Buffers in any language – 1st class citizen	Code generation through OpenAPI/Swagger(add-on) – 2nd class citizen
RPC based – gRPC does the plumbing for us	HTTP verbs based – we have to write the plumbing or use a 3rd party library