2 - [Theory] gRPC Internals Deep Dive

Protocol Buffers & Language Interoperability

Protocol Buffers role in gRPC

- Protocol Buffers is used to define the:
  - Messages (data, Reauest and Response)
  - Service (Service name and RPC endpoints)
- We then generate code from it!

Efficiency of Protocol Buffers over JSON

- gRPC uses Protocol Buffers for communications.
- Let's measure the payload size vs JSON:

JSON: 55 bytes

```
{
    "age": 35,
    "first_name": "Stephane",
    "last_name": "Maarek"
}
```

Same in Protocol Buffers: 20 bytes

```
message Person {
    int32 age = 1;
    string first_name = 2;
    string last_name = 3;
}
```

- **We save in Network Bandwidth**

- Parsing JSON is actually CPU intensive (because the format is human readable)
- Parsing Protocol Buffers (binary format) is less CPU intensive because it's closer to how a machine represents data
- By using gRPC, the use of Protocol Buffers means faster and more efficient communication, friendly with mobile devices that have a slower CPU

Quick tour on grpc.io

https://grpc.io/

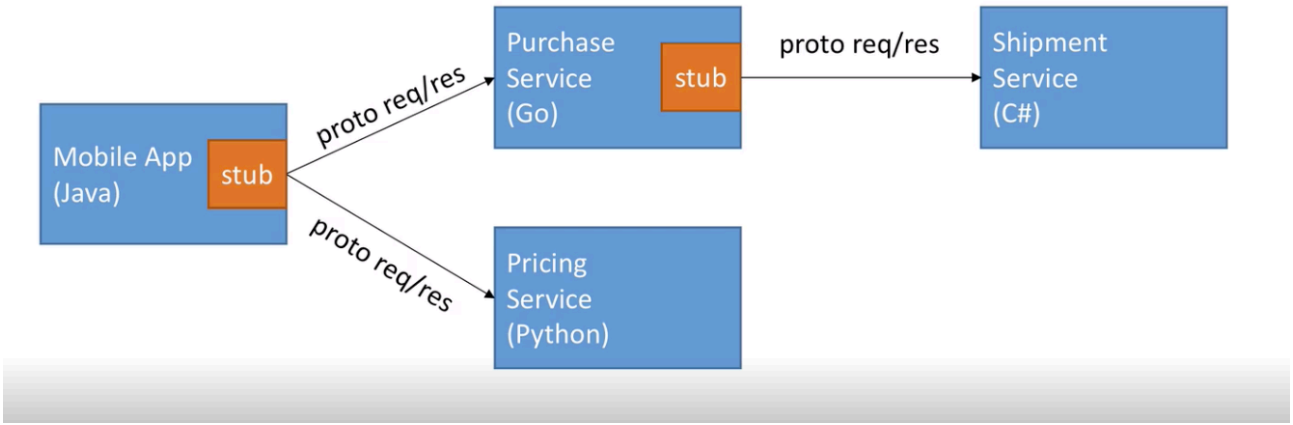
- And many other languages are unofficially supported! (Swift, etc...)

gRPC Languages

- gRPC will have these main implementations:
    - GRPC-JAVA: Pure implementation of gRPC in Java
    - GRPC-Go: Pure implementation of gRPC in Go
    - GRPC-C: Pure implmenetation of gRPC in C
        - gRPC C++
        - gRPC Python
        - gRPC Ruby
        - gRPC Objective C
        - gRPC PHP
        - gRPC C#
- Other languages implement gRPC natively or rely on C implementation

gRPC can be used by any language

- Because the code can be generated by any language, it makes it super simple to create micro-service in any language that interact with each other



Summary: Why Protocol Buffers?

- 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, therefore very efficient to send / receive on a network and serialize / de-serialize on a CPU
- Protocol Buffers defines rules to make an API evolve without breaking existing clients, which is helpful for microservices

HTTP/2

What's HTTP/2?

- gRPC leverages HTTP/2 as a backbone for communications

- Demo: https://imagekit.io/demo/http2-vs-http1
- HTTP/2 is the newer standard for inernet communications that address common pitfall of HTTP/1.1 on modern web pages
- Before we go into HTTP/2, let's look at some HTTP/1.1 request

How HTTP /1.1 works

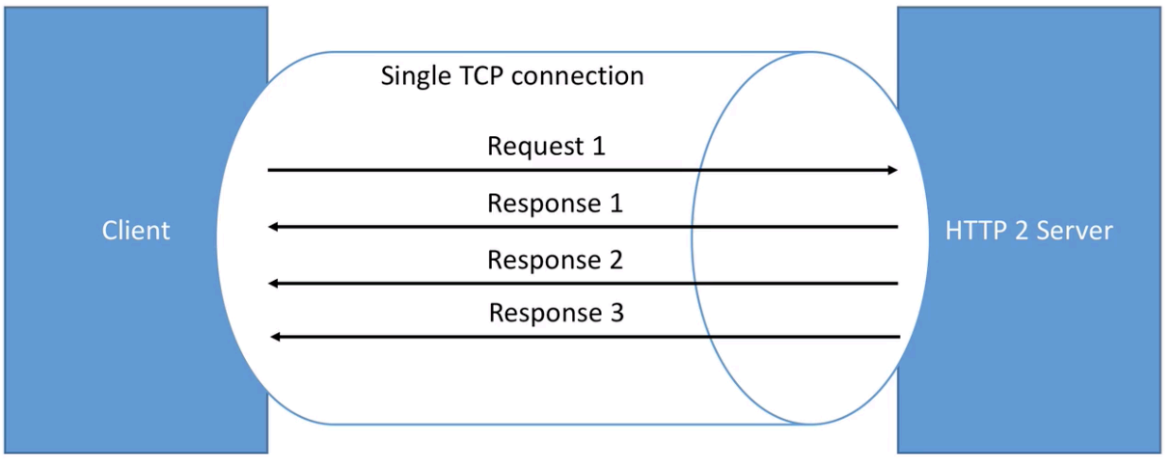
- HTTP 1.1 was released in 1997. It has worked great for many years!
- HTTP 1.1 opens a new TCP connection to a server at each request
- It does not compress headers (which are plaintext)
- It only works with Request/Reponse mechanism (no server push)
- HTTP was originally composed of two commands:
    - GET: to ask for content
    - POST: to send content
- Nowadays, a web page loads 80 assets on average
- Headers are sent at every request and are PLAINTEXT (heavy size)
- Each request opens a TCP connection
- These inefficiencies add latency and increase network packet size

How HTTP/2 works

- HTTP 2 was released in 2015. It has been battled tested for many years! (and was before that tested by Google under the name SPDY)
- HTTP 2 supports multiplexing
    - The client & server can push messages in parallel over the same TCP connection
    - This greatly reduces latency
- HTTP 2 supports server push
    - Servers can push streams (multiple messages) for one request from the client
    - This saves round trips (latency)
- HTTP2 supports header compression
    - Headers (text based) can now be compressed
    - These have much less impact on the packet size
    - (remember the average http request may have over 20 headers, due to cookies, content cache, and application headers)
- HTTP/2 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)

- HTTP/2 is secure (SSL is not required but recommended by default)

How HTTP/2 works

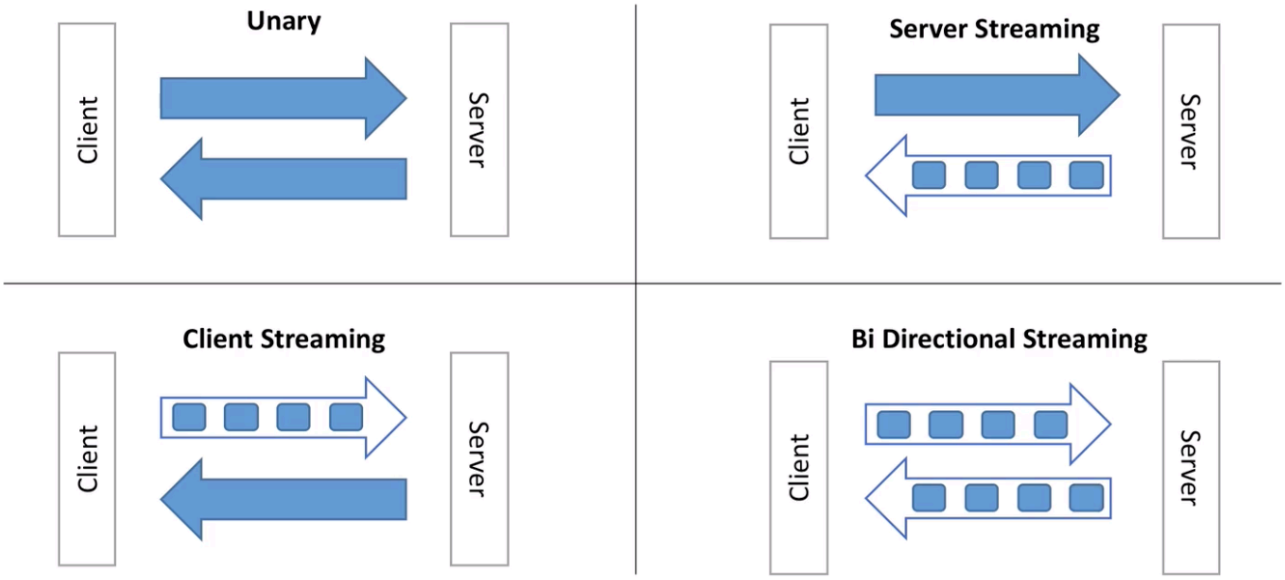


HTTP/2: Bottom Line

- Less chatter
- More efficient protocol (less bandwith)
- Reduced Latency
- Increased Security

- **And you get all these improvements out of the box by using the gRPC framework!**

4 Types of gRPC APIs



- Unary is what a traditional API looks like (HTTP REST)
- HTTP/2 as we've seen, enables APIs to now have streaming capabilities

- The server and the client can push multiple messages as part of one request!
- In gRPC it's very easy to define these APIs as we'll see

```
service GreetService {
  // Unary
  rpc Greet(GreetRequest) returns (GreetResponse) {};

  // Streaming Server
  rpc GreetManyTimes(GreetManyTimesRequest) returns (stream GreetManyTimesResponse) {};

  // Streaming Client
  rpc LongGreet(stream LongGreetRequest) returns (LongGreetResponse) {};

  // Bi Directional Streaming
  rpc GreatEveryone(stream GreetEveryoneRequest) returns (stream GreetEveryoneResponse) {};
}
```

Scalability in gRPC

- gRPC Servers are asynchronous by default
- This means they do not block threads on request
- Therefore each gRPC server can serve millions of requests in parallel

- gRPC Clients can be asynchronous or synchronous (blocking)
- The client decides which model works best for the performance needs
- gRPC Clients can perform client side load balancing

- As a proof of scalability: Google has 10 BILIION gRPC requests being made per second internally

Security in gRPC

- By default gRPC strongly advocates for you to use SSL (encryption over the wire) in your API
- This means that 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
- Additionally using Interceptors, we can also provide authentication (we'll learn about Interceptors in the advanced section)

gRPC vs REST

REST API example

```
POST /member/109087/cart HTTP/1.1
Host: api.example.com
Authorization: Basic username:password
Content-type: application/json
Accept: application/hal+json

{
    "inventory_id": 12345,
    "quantity": 1
}

HTTP/1.1 201 Created
Date: Mon, 20 Jun 2011 21:15:00 GMT
Content-Type: application/hal+json
Location: /member/109087/cart/14418796
```

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 requests only
Stream Support	Request/Response support only
API Oriented - "What" (no constraints - free design)	CRUD Oriented (Create - Retrieve - Update - Delete / POST GET PUT DELETE)
Code Generation through Protocl Buffers i 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

- https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html: Finds that gRPC is 25 times more performant than REST API (as defined as time to have the response for an API)

Section Summary - Why use gRPC

- Easy code definition in over 11 languages
- Uses a modern, low latency HTTP/2 trasnport mechanism
- SSL Security is built in
- Support for streaming APIs for maximum performance
- gRPC is API oriented, instead of Resource Oriented like REST