Can we do better than HTTP/JSON?

Open Source Leadership Summit, Feb 2017

Varun Talwar

@varungyan

Why is JSON so popular?

1. Simple
2. Self describing
3. Easy to debug
4. Easy to process in languages
5. Browser tooling
6. Less verbose than XML

Lacks

1. Extend as needed
2. Performance
3. Storage size

The Fallacies of Distributed Computing

The network is reliable

Latency is zero

Bandwidth is infinite

The network is secure

Topology doesn't change

There is one administrator

Transport cost is zero

The network is homogeneous

Yes, we can

An open, performant, resilient and universal way to connect and operate distributed systems and provide a framework to make much smarter clients and servers

Making Google frameworks available in Open

Borg → Kubernetes

Stubby → GRPC

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

Images by Connie Zhou

http://grpc.io

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

What is gRPC?

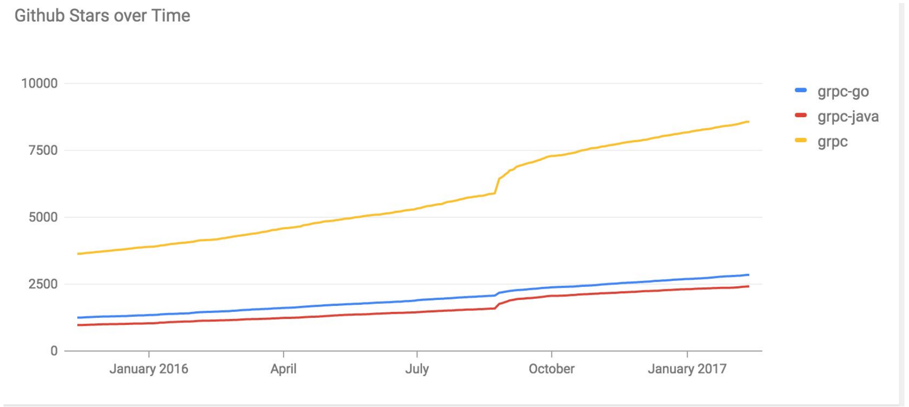
- **HTTP/2** and **Protocol Buffer** based RPC framework
- Evolution of Stubby; being adopted at Google as next gen framework
- Open, Secure, Performant, Multiplatform

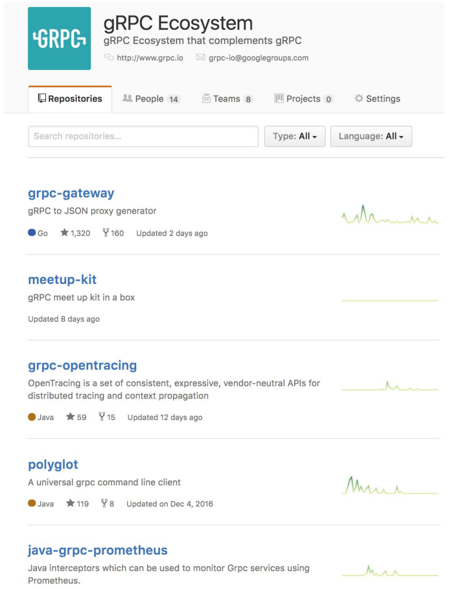
Multiplatform

- Idiomatic APIs in popular languages (C++, Go, Java, C#, Node.js, Ruby, PHP, Python)
- Supports mobile devices (Android Java, iOS Obj-C)
- Linux, Windows, Mac OS X

Where is the project today?

- 1.1 with stable APIs
- Joining CNCF soon :-)
- Well documented with an active community
- Reliable with continuous running tests
    - Deployable in your environment
- Measured with an open performance dashboard
    - Deployable in your environment
- Well adopted inside and outside Google




gRPC Ecosystem
gRPC Ecosystem that complements gRPC
http://www.grpc.io grpc-io@googlegroups.com

grpc-gateway
gRPC to JSON proxy generator
Go ★ 1,320 160 Updated 2 days ago

meetup-kit
gRPC meet up kit in a box
Updated 8 days ago

grpc-opentracing
OpenTracing is a set of consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation
Java ★ 59 15 Updated 12 days ago

polyglot
A universal grpc command line client
Java ★ 119 8 Updated on Dec 4, 2016

java-grpc-prometheus
Java interceptors which can be used to monitor Grpc services using Prometheus.

Some early adopters

Square NETFLIX lyft
The New York Times

Microservices: in data centres

CoreOS docker
carbon3D
Cockroach LABS

Client Server communication/Internal APIs

JUNIPER NETWORKS CISCO ARISTA

Streaming telemetry from network devices

Allo

Mobile Apps

PROTOBUF & HTTP/2

Protocol Buffers are

- Efficient
  - Binary protocol; much faster serialization
  - 3-10x smaller and 20-100x faster than XML[1]
  - 1.5-3x smaller and 1.5-3x faster than JSON[2]
- With Simple IDL
- Strong typing -> avoids errors earlier & can enforce strict contracts
- Allows to extend/grow -> API evolution
- Used for logging, storage, and talking to other servers

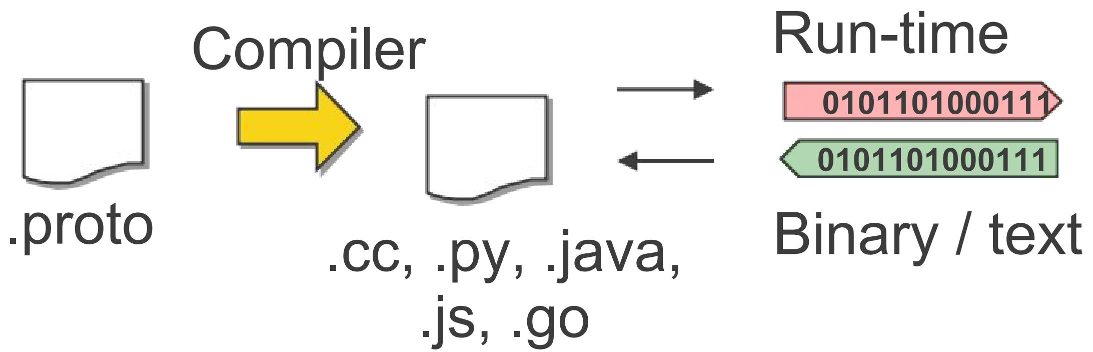
How does it look?

```
$ cat student.proto
// Definition of a Student object

syntax = "proto2";
option cc_api_version = 2;
option java_api_version = 2;

package social;

message Student {
  required int32 unique_id = 1;
  required string first_name = 2;
  ...
  optional double gpa = 8;
  optional string nickname = 9 [default = "bob"];
  repeated int32 friend_id = 10;
}
```



How we roll at Google

Save
There are two kinds of jobs here, Bob.
"Proto to proto" and "Proto to UI"
Two, Bob.

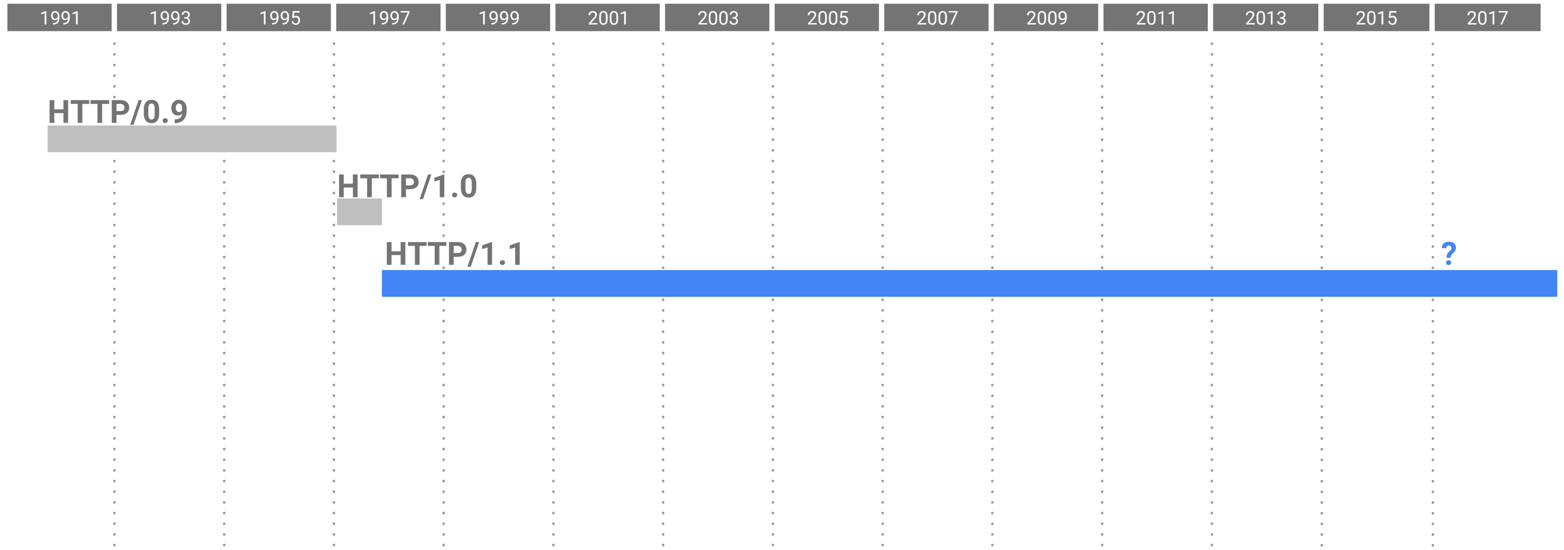
HTTP/1.x vs HTTP/2

http://http2.golang.org/gophertiles

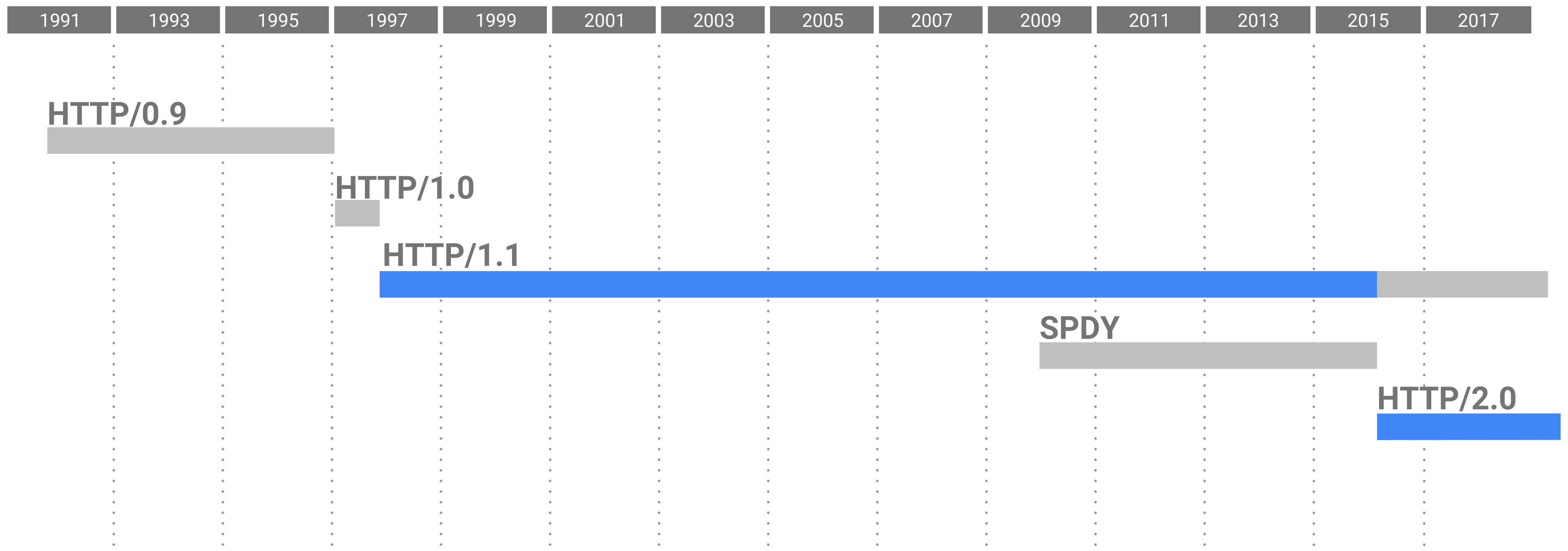
htt

Google

History of HTTP



History of HTTP



HTTP/2

HTTP/2 is extending, not replacing, the previous HTTP standards.

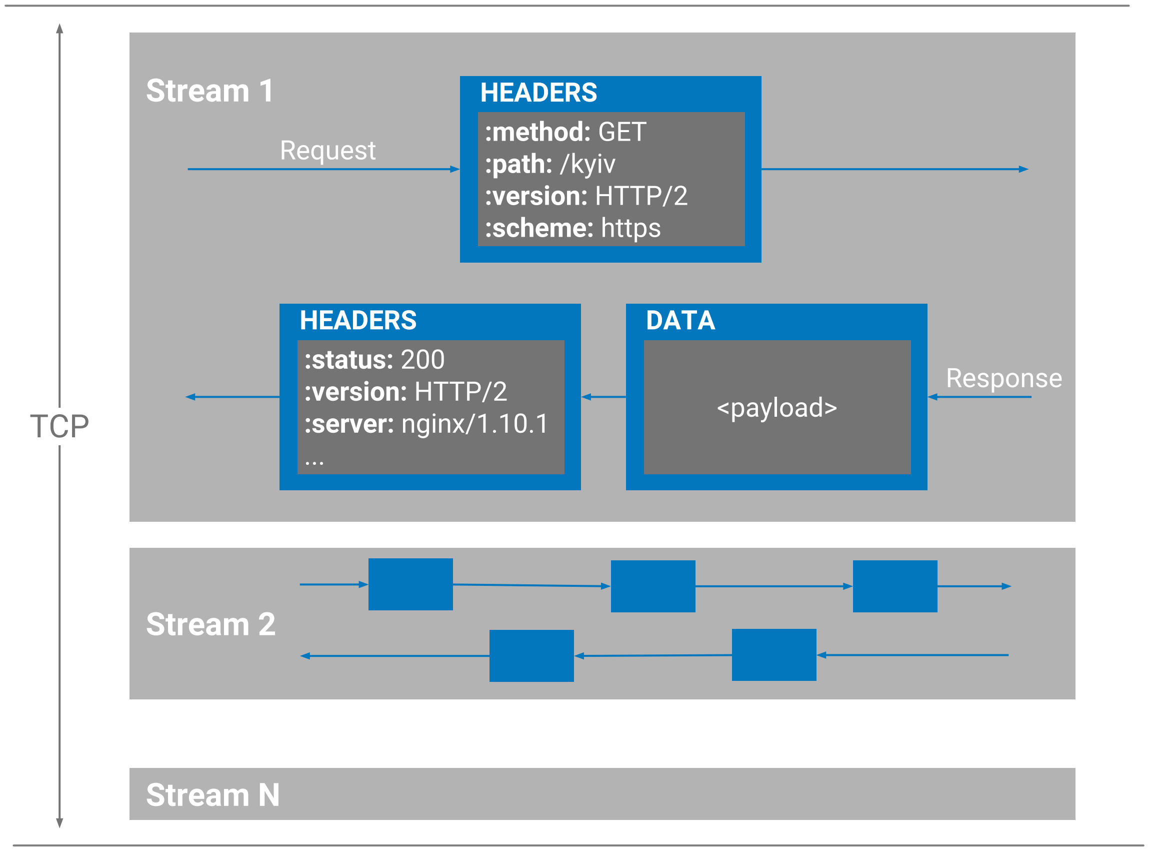
The application semantics of HTTP are the same::

- HTTP header fields
- HTTP Methods
- Request-response
- Status codes
- URIs

HTTP/2 modifies how the data is formatted (framed) and transported between the client and server.

Binary Framing

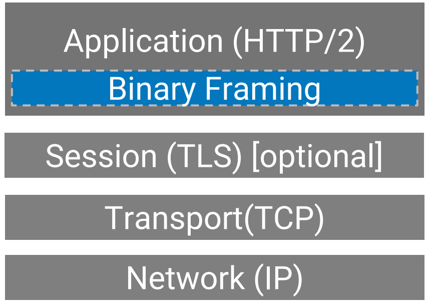
HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded **frames**, which are then mapped to **messages** that belong to a **stream**, and all of which are **multiplexed within a single TCP connection**.

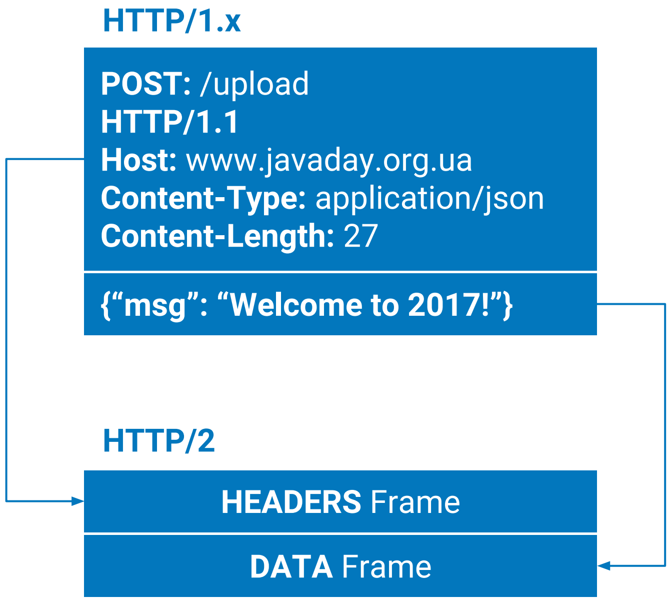


Google

HTTP/2 in One Slide

- Single TCP connection.
- No Head-of-line blocking.
- Binary framing layer.
- Request –> Stream.
- Header Compression.





KEY DESIGN PRINCIPLES

gRPC Principles & Requirements

Coverage & Simplicity

The stack should be available on every popular development platform and easy for someone to build for their platform of choice. It should be viable on CPU & memory limited devices.

http://www.grpc.io/blog/principles

gRPC Speaks Your Language

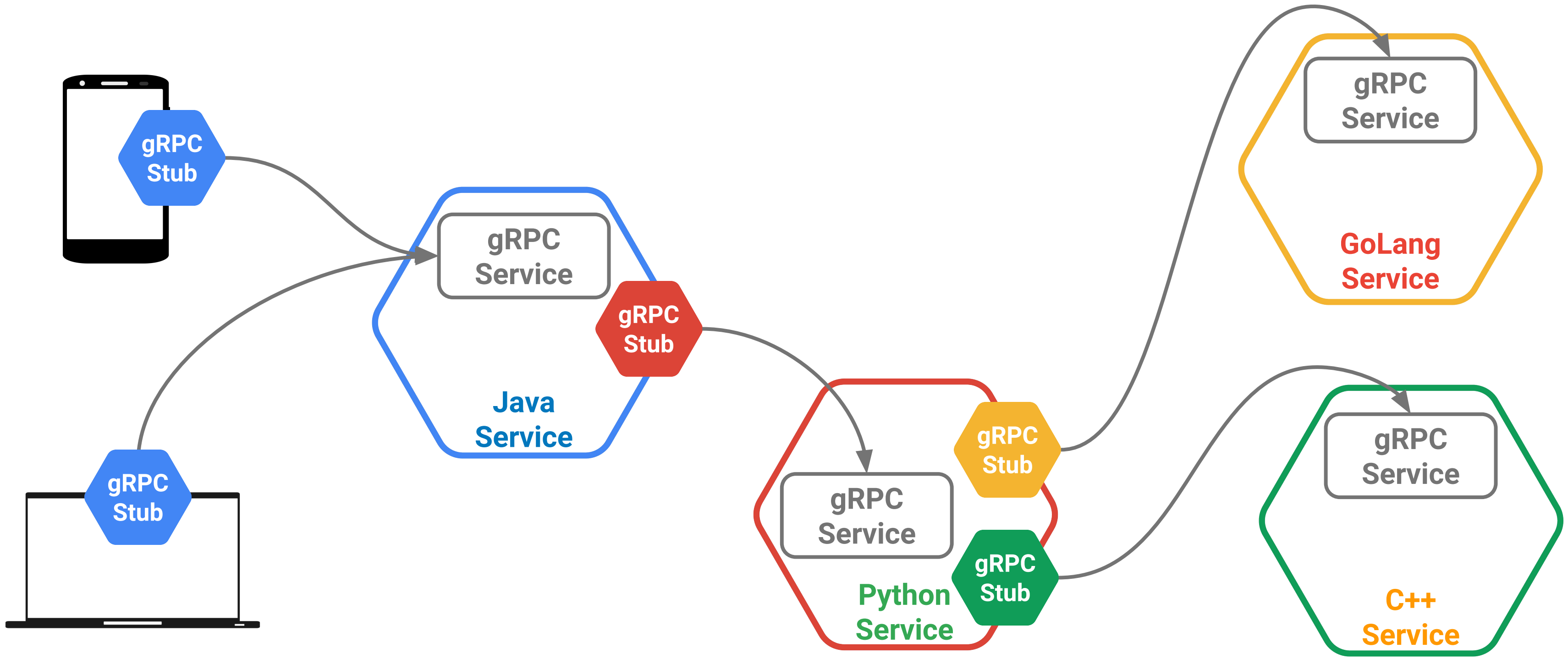
Service definitions and client libraries

- Java
- Go
- C/C++
- C#
- Node.js
- PHP
- Ruby
- Python
- Objective-C

Platforms supported

- MacOS
- Linux
- Windows
- Android
- iOS

Interoperability



Easy to get started

Language	Platform	Command
Node.js	Linux, Mac, Windows	`npm install grpc`
Python	Linux, Mac, Windows	`pip install grpcio`
Ruby	Linux, Mac, Windows	`gem install grpc`
PHP	Linux, Mac, Windows	`pecl install grpc-beta`
Go	Linux, Mac, Windows	`go get google.golang.org/grpc`
Objective-C	Mac	Runtime source fetched automatically from Github by Cocoapods
C#	Windows	Install gRPC NuGet package from your IDE (Visual Studio, Monodevelop, Xamarin Studio)
Java	Linux, Mac, Windows	Use our Maven and Gradle plugins that provide gRPC with statically linked `boringssl`
C++	Linux, Mac, Windows	Currently requires manual build and install

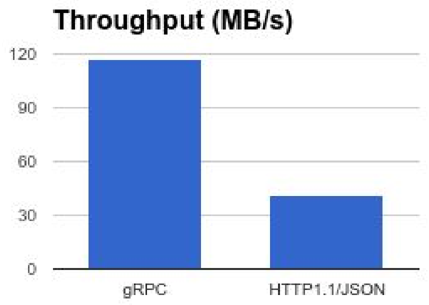
gRPC Principles & Requirements

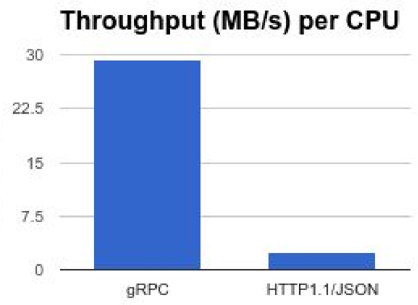
General Purpose and Performant

The stack should be applicable to a broad class of use-cases while sacrificing little when compared to a use-case specific stack

http://www.grpc.io/blog/principles

gRPC vs JSON/HTTP for Google Cloud Pub/Sub





3x increase in throughput

Publishing 50KB messages at maximum throughput from a single n1-highcpu-16 GPE VM instance, using 9 gRPC channels.

11x difference per CPU

More impressive than the almost 3x increase in throughput, is that it took only 1/4 of the CPU resources.

Source Blog of Cloud Pubsub

Some external perf comparisons

RPC	# of requests	# of clients	total time	per-request time
`jsonrpc`	300,000	1	8m7.270s	1.624ms
gRPC	300,000	1	36.715s	122.383μs
gRPC	300,000	100	7.167s	23.892μs

And if compared on memory usage:

RPC	`jsonrpc`	gRPC	delta
NsPerOp	487271046903	36716116701	-92.46%
AllocsPerOp	32747687	25221256	-22.98%
AllocedBytesPerOp	3182814152	1795122672	-43.60%

RPC	`jsonrpc`	gRPC with 100 clients	delta
NsPerOp	487271046903	7168591678	-98.53%
AllocsPerOp	32747687	25230286	-22.96%
AllocedBytesPerOp	3182814152	1795831944	-43.58%

RPC	gRPC	gRPC with 100 clients	delta
NsPerOp	36716116701	7168591678	-80.48%
AllocsPerOp	25221256	25230286	+0.04%
AllocedBytesPerOp	1795122672	1795831944	+0.04%

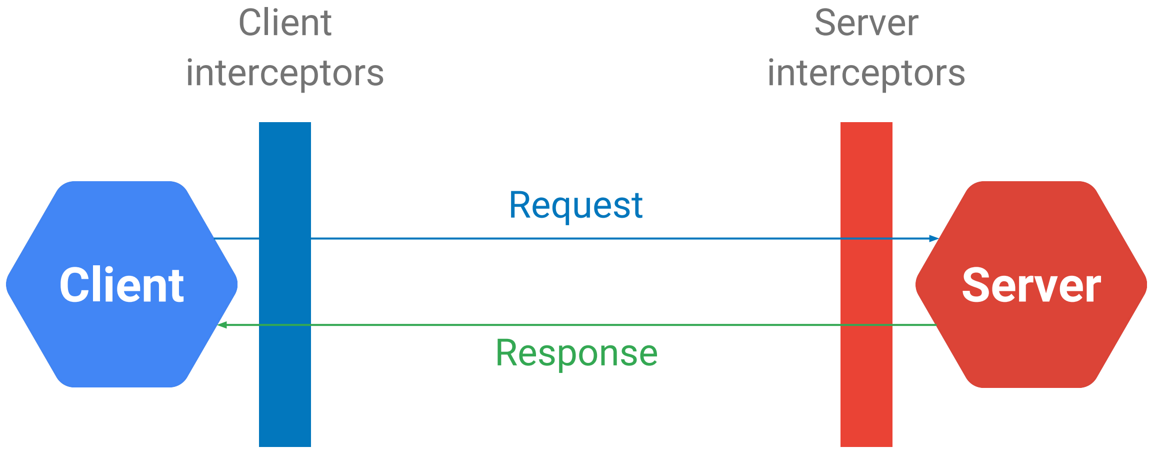
gRPC Principles & Requirements

Pluggable

Large distributed systems need security, health-checking, load-balancing and failover, monitoring, tracing, logging, and so on. Implementations should provide extensions points to allow for plugging in these features and, where useful, default implementations.

http://www.grpc.io/blog/principles

Interceptors



gRPC Principles & Requirements

Payload Agnostic

Different services need to use different message types and encodings such as protocol buffers, json, xml, thrift; the protocol and implementations must allow for this. Similarly the need for payload compression varies by use-case and payload type: the protocol should allow for pluggable compression mechanisms.

http://www.grpc.io/blog/principles

gRPC Principles & Requirements

Flow Control

Computing power and network capacity are often unbalanced between client and server. Flow control allows for better buffer management as well as providing protection from DOS by an overlay active peer.

http://www.grpc.io/blog/principles

gRPC Principles & Requirements

Network Control

Operations can be expensive and long lived.
Cancellation allows servers to reclaim resources when clients are well-behaved.
Deadlines let servers know what is the expected time of response and behave accordingly.

http://www.grpc.io/blog/principles

gRPC Principles & Requirements

Streaming

Storage systems rely on streaming and flow control to express large data sets. Other services like voice to text or stock tickers rely on streaming to represent temporally related message sequences.

http://www.grpc.io/blog/principles

gRPC Service Definitions

Unary

Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.

Server streaming

The client sends a request to the server and gets a stream to read a sequence of messages back.

The client reads from the returned stream until there are no more messages.

Client streaming

The client send a sequence of messages to the server using a provided stream.

Once the client has finished writing the messages, it waits for the server to read them and return its response.

BiDi streaming

Both sides send a sequence of messages using a read-write stream. The two streams operate independently. The order of messages in each stream is preserved.

gRPC Principles & Requirements

Metadata Exchange

Common cross cutting concerns like auth and tracing rely on the exchange of data that is not part of declared interface. Deployments rely on their ability to evolve these features at a different rate to the individual APIs exposed by services.

http://www.grpc.io/blog/principles

In summary, gRPC and Stubby have some lessons

- Common IDL with performant serializer gives better performance and improves developer productivity
- Framework handling hard concepts like streaming, deadlines, cancellations, flow control help makes devs life easier and services thinner
- Framework handling stats/tracing/logging, LB in a uniform way allows for easy change management and uniform observability

Single horizontal framework for all service-service communication with logging, monitoring, tracing, network controls, service discovery, load balancing built in; makes lives much easier within an organization

Let’s build on top and make it stronger

- Building on !
  - First class Web support
  - First class caching, compression support
  - First class debugging and testing support
  - First class metrics and tracing support
  - Tooling for testing, docgen, samplegen
  - Automagic retries
  - Smart load balancing support
  - First class service discovery and service-service auth

Together we can leverage this framework and let all developers build smarter clients and servers

Thank you and join us !

Twitter:	**@grpcio**
Site:	grpc.io
Group:	grpc-io@googlegroups.com
Repo:	github.com/grpc
	github.com/grpc/grpc-java
	github.com/grpc/grpc-go

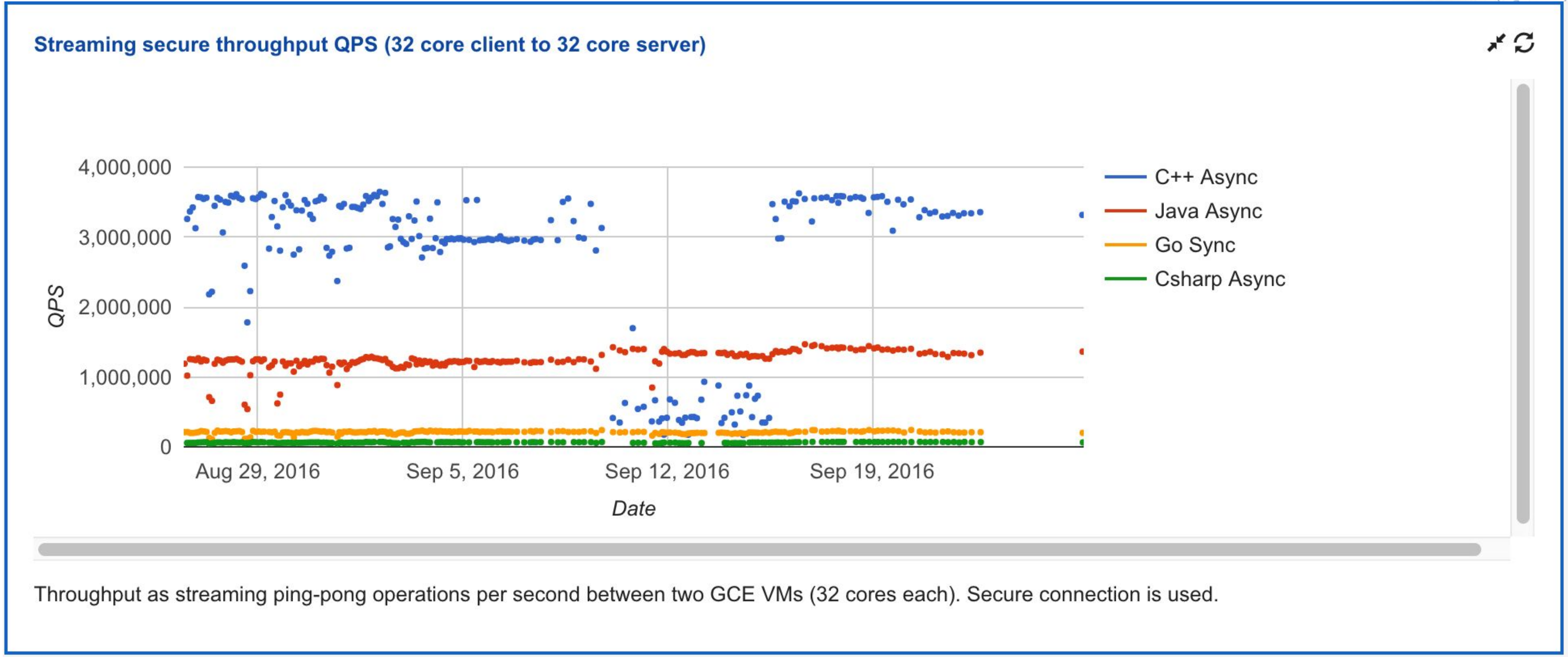
Q & A

H1 Protocol Overhead

HTTP metadata (headers).

Not compressed plain text headers for each and every HTTP request.

Workarounds: cookies, sessions, concatenation, sprinting, etc.



Streaming secure throughput QPS (32 core client to 32 core server)

Throughput as streaming ping-pong operations per second between two GCE VMs (32 cores each). Secure connection is used.

Trends

- Open Source
- IOT
- Mobile First
- AI First
- Public/Hybrid Clouds
- VR/AR
- Microservices
- SaaS/XaaS
- Software Defined Everything

Trends

- **Open Source** => Spreading everywhere !
- **IOT** => millions of devices, constrained resources, reliable network
- **Public/Hybrid Cloud** => distributed environments, cloud services
- **VR/AR** => resource intensive, battery/data consuming
- **Microservices** => network performance matters
- **SaaS/XaaS** => consume anything from anywhere
- **Mobile first** => first class mobile libs
- **Software Defined Everything** => Apps, Config, Networks

We can do better in this world where IOT, performance matters, developer productivity and agility matters, resilience matters, microservices, multi-cloud,

1

HTTP1.x/JSON doesn’t cut it !

1. WWW, browser growth - bled into services
2. Stateless
3. Text on the wire
4. Loose contracts
5. TCP connection per request
6. Nouns based
7. Harder API evolution
8. Think compute, network on cloud platforms

2

Establish a lingua franca

1. Protocol Buffers - Since 2003.
2. Start with IDL
3. Have a language agnostic way of agreeing on data semantics
4. Code Gen in various languages
5. Forward and Backward compatibility
6. API Evolution

3

Design for fault tolerance and control

- Sync and Async APIs
- Need fault tolerance: Deadlines, Cancellations
- Control Knobs: Flow control, Service Config, Metadata

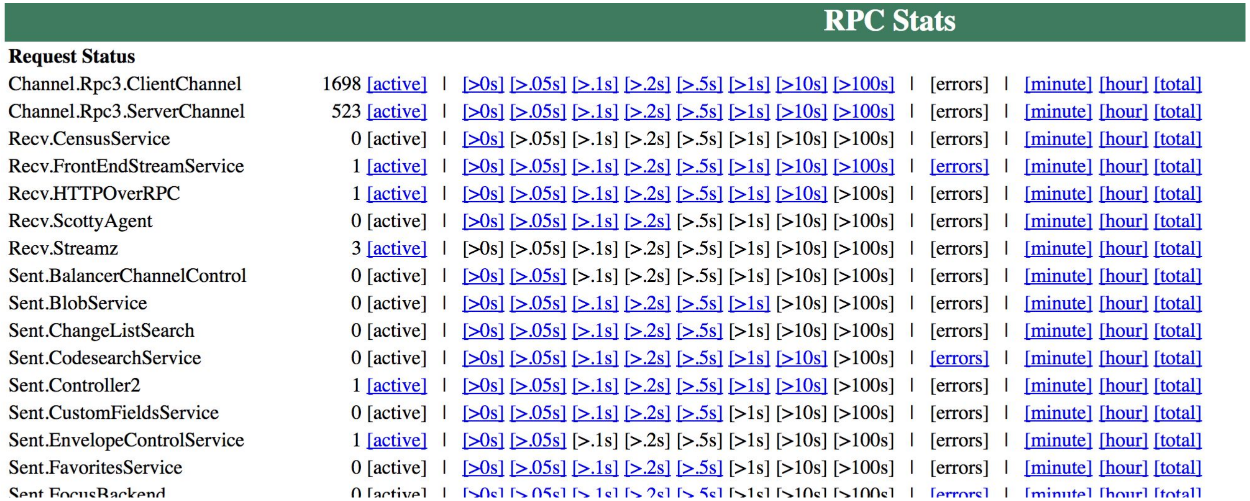
Service Config

- Policies where server tells client what they should do
- Can specify deadlines, lb policy, payload size per method of a service
- Loved by SREs, they have more control
- Discovery via DNS

4

Don’t fly blind: Stats

- What is the mean latency time per RPC?
- How many RPCs per hour for a service?
- Errors in last minute/hour?
- How many bytes sent? How many connections to my server?



RPC Stats

Request Status

Channel.Rpc3.ClientChannel	1698 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Channel.Rpc3.ServerChannel	523 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Recv.CensusService	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Recv.FrontEndStreamService	1 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Recv.HTTPOverRPC	1 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Recv.ScottyAgent	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Recv.Streamz	3 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.BalancerChannelControl	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.BlobService	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.ChangeListSearch	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.CodesearchService	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.Controller2	1 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.CustomFieldsService	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.EnvelopeControlService	1 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.FavoritesService	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]
Sent.FocusBackend	0 [active]	[>0s] [>.05s] [>.1s] [>.2s] [>.5s] [>1s] [>10s] [>100s]	[errors]	[minute] [hour] [total]

Data collection by arbitrary metadata is useful

- Any service’s resource usage and performance stats in real time by (almost) any arbitrary metadata
  1. Service X can monitor CPU usage in their jobs broken down by the name of the invoked RPC and the mdb user who sent it.
  2. Ads can monitor the RPC latency of shared bigtable jobs when responding to their requests, broken down by whether the request originated from a user on web/Android/iOS.
  3. Gmail can collect usage on servers, broken down by according POP/IMAP/web/Android/iOS. Layer propagates Gmail's metadata down to every service, even if the request was made by an intermediary job that Gmail doesn't own

- Stats layer export data to varz and streamz, and provides stats to many monitoring systems and dashboards

5

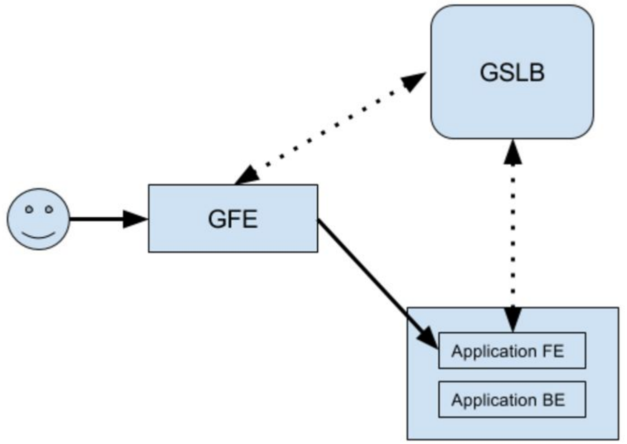
Diagnosing problems: Tracing

- 1/10K requests takes very long. Its an ad query :-) I need to find out.
- Take a sample and store in database; help identify request in sample which took similar amount of time

- I didnt get a response from the service. What happened? Which link in the service dependency graph got stuck? Stitch a trace and figure out.
- Where is it taking time for a trace? Hotspot analysis
- What all are the dependencies for a service?

5

Load Balancing is important !



Iteration 1: Stubby Balancer
Iteration 2: Client side load balancing
Iteration 3: Hybrid
Iteration 4: gRPC-lb

⟶	Access order of a particular incoming user request
···▶	Out of band requests reporting load/requesting advice

Google

HTTP/2 in the Wild

Implementations

- Apache Tomcat 8.5+
- Apache HTTP Server 2.4.17+ (C)
- NGINX (C)
- Jetty
- Netty
- Undertow
- Vert.x
- OkHttp (Android)

Tools

- curl 7.43.0+
- Wireshark
- jmeter
- HTTP/2 Test
- h2i
- h2load
- and more

Browsers

- Chrome
- Firefox
- Safari*
- Opera
- Edge
- IE 11*
- Android Browser
- Chrome for Android

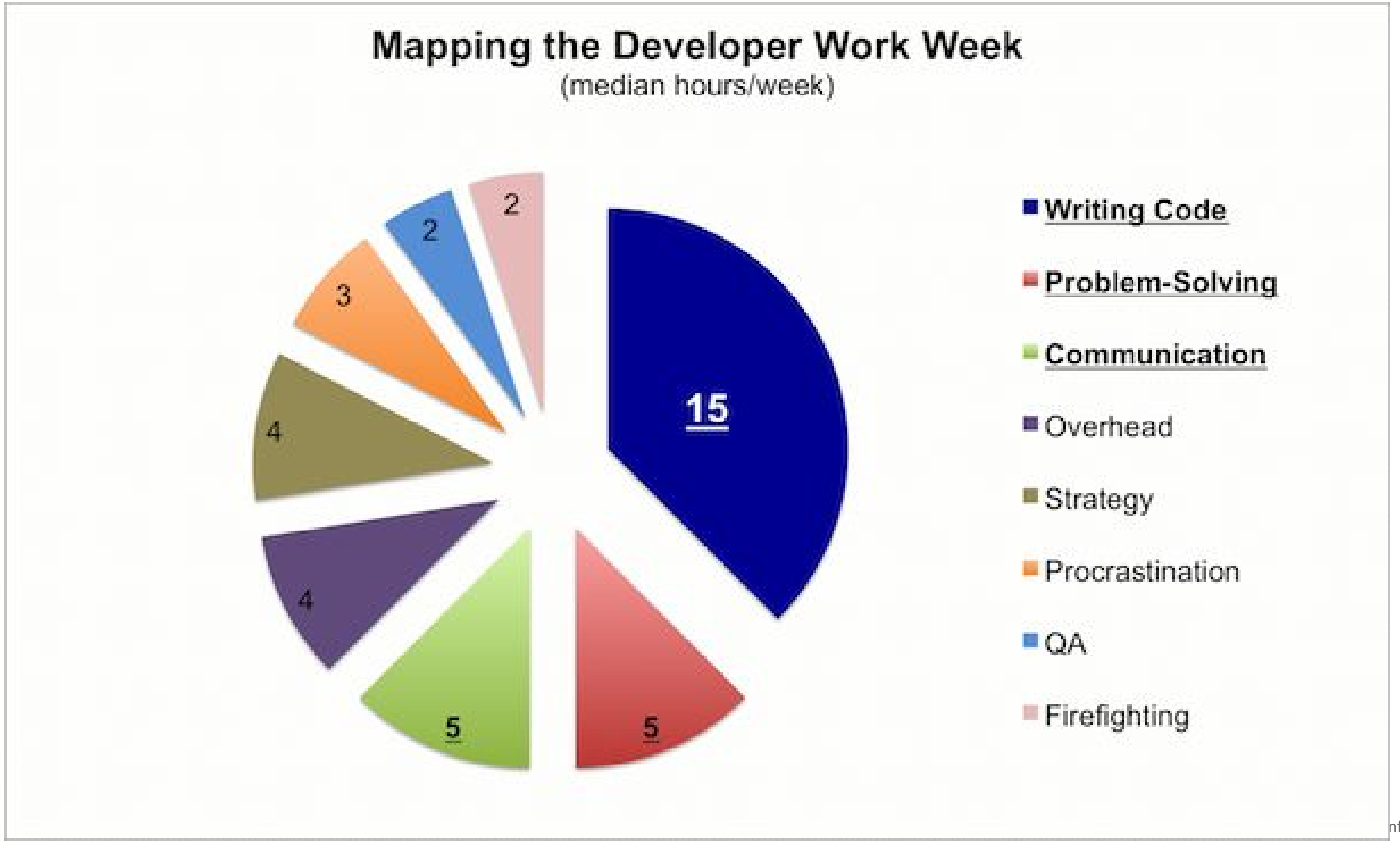
Java

- JEP 110: HTTP/2 Client.
- JEP 244: TLS ALPN Extension.

https://github.com/http2/http2-spec/wiki/Implementations
https://github.com/http2/http2-spec/wiki/Tools
http://caniuse.com/#feat=http2

Agility & Resilience

Developer Productivity

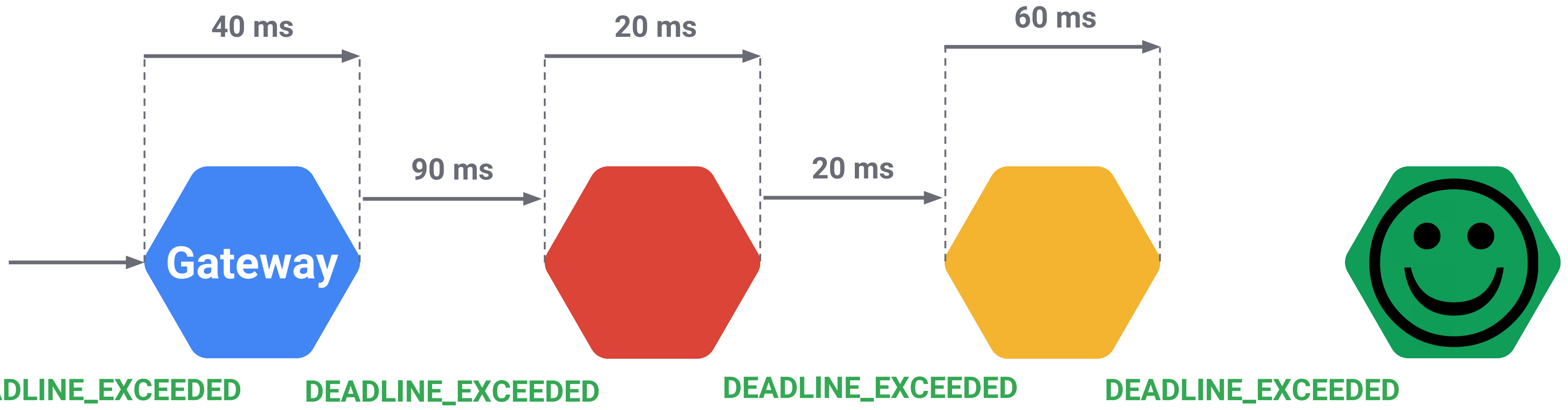


Performance

Google

Deadline Propagation

```
withDeadlineAfter(200, MILLISECONDS)
```



Now = 1476600000**000**

Deadline = 1476600000**200**

Now = 1476600000**040**

Deadline = 1476600000**200**

Now = 1476600000**150**

Deadline = 1476600000**200**

Now = 1476600000**230**

Deadline = 1476600000**200**

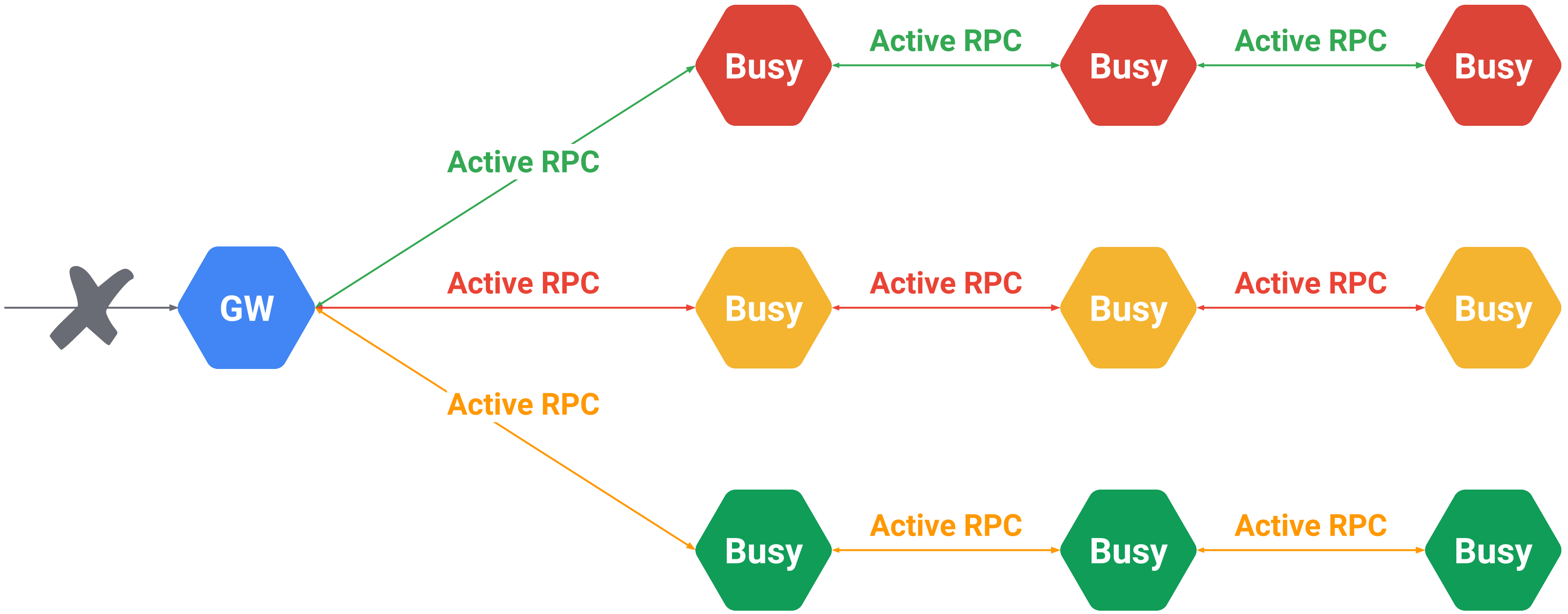
Cancellation?

Deadlines are expected.

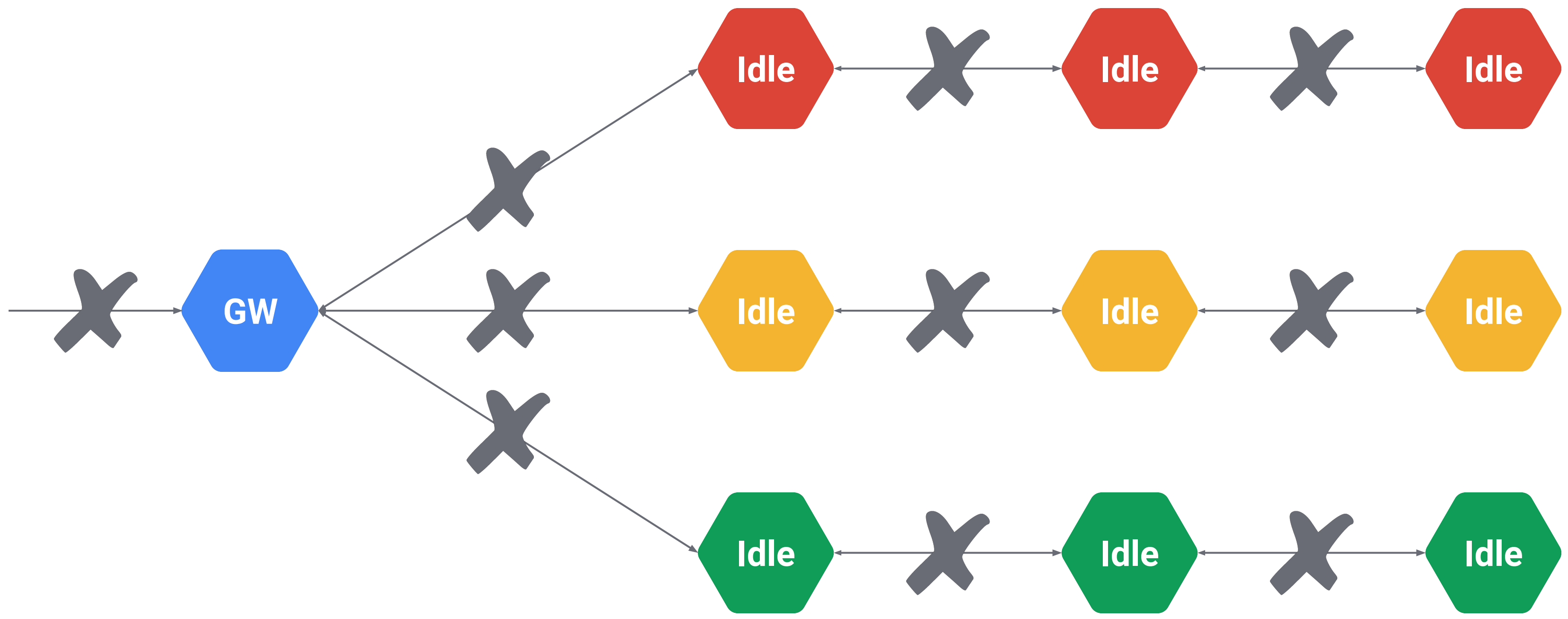
What about unpredictable cancellations?

- User cancelled request.
- Caller is not interested in the result any more.

Cancellation?



Cancellation Propagation


GW
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle