gRPC and gruf in Ruby

Exploring Google's new gRPC protocol for interservice communication

Presenter

Shaun McCormick

Date

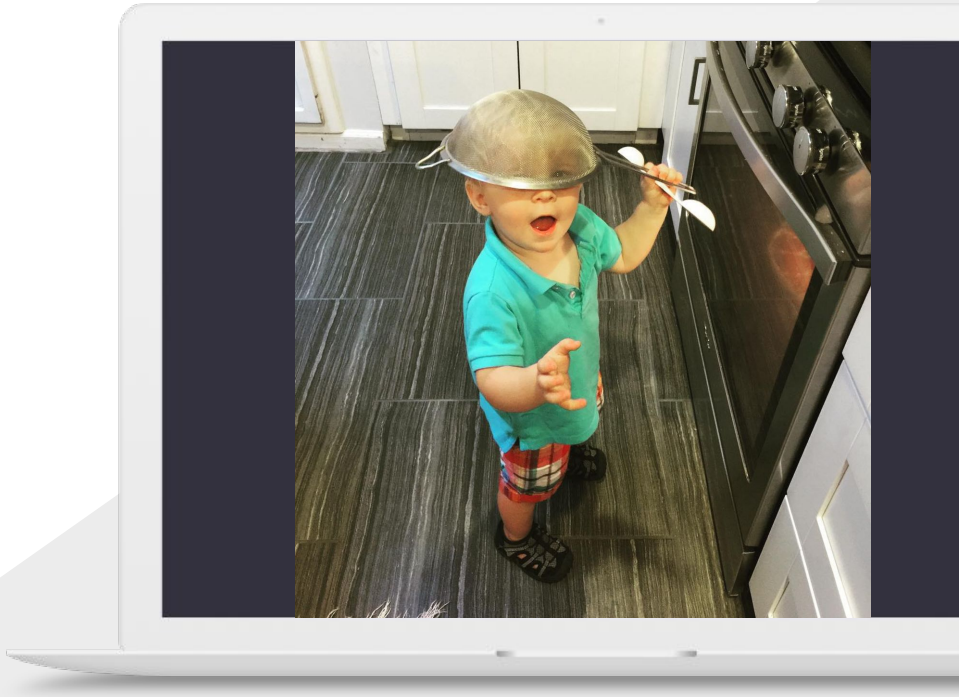
September 11, 2017

BIGCOMMERCE

Shaun McCormick

@splittingred
shaun.mccormick@bigcommerce.com

- Staff Software Engineer @ BigCommerce
- Ruby for 8+ years
- Services team at BC, driving SOA efforts
- Somewhat obsessed about board games
- Feedback always appreciated!



Overview

- What is gRPC?
- Why do we use gRPC?
- What is gruf? Why was it made?
- Using gRPC+Gruf in Ruby
- Thoughts/Caveats on gRPC in a SOA

BIGCOMMERCE

What is gRPC?

What is gRPC?

gRPC is a RPC-based service communication framework, written by Google, that operates over HTTP/2. It is an alternative to the common REST/HTTP1 protocol.

In our testing, it has shown to be 3x faster than a normal REST request.

Automatic client/server generation

Supports over 10 languages, with easily extendable stubs generated from automatable tooling.

Defined, type-safe service contracts

Utilizes Google Protobuf for communication contracts, allowing type-safety over the wire.

BIGCOMMERCE

HTTP/2 Support

- Stream multiplexing
  - Many requests/streams over a single connection
- Binary framing and transport
- Header compression
- Bidirectional streaming
  - Allows response data to reach the client before it has finished sending
- Stream dependency management
  - Client can indicate resource prioritization to the server

BIGCOMMERCE

Protobuf in gRPC

- The service defines its methods
- Each method has a request and response object
- Messages can be shared across requests/responses
- Fields have validated, specified types
- Binary serialization since key name is not needed

```
syntax = "proto3";

package rpc;

service Products {
    rpc GetProduct(GetProductReq) returns (GetProductResp) {}
    rpc GetProducts(GetProductsReq) returns (stream Product) {}
    rpc CreateProducts(stream Product) returns (CreateProductsResp) {}
    rpc CreateProductsInStream(stream Product) returns (stream Product) {}
}

message Product {
    uint32 id = 1;
    string name = 2;
    float price = 3;
}

message GetProductReq {
    uint32 id = 1;
}

message GetProductResp {
    Product product = 1;
}

message GetProductsReq {
    string search = 1;
    uint32 limit = 2;
}

message CreateProductsResp {
    repeated Product products = 1;
}
```

4 Request Types

COLUMN TITLE	COLUMN TITLE
Unary Request/Response	Most common, familiar to REST-style APIs
Server Streaming	Send back large payloads and iterate on them as they come
Client Streaming	Excellent for sending large payloads from clients, reduces strain on server
Bidirectional Streaming	Useful for chat-style apps, async request/response workflows

Why do we use gRPC?

Interoperability

- With multiple languages at BC, we needed a consistent service transport
- Enabled abstraction and automation of:
  - Tracing
  - Health checks
  - Monitoring/metrics
- Simple, canonical service naming/discovery (through Consul/LinkerD):
  - All requests go to `linkerd:4142`
  - Consul tag: `grpc:bigcommerce.rpc.content.Placements`
  - L5d dtab: `/svc/bigcommerce.rpc.content.Placements => /#/io.l5d.consul/.local/content-grpc;`

BIGCOMMERCE

Type Safety

- We use a mix of type-safe and duck-typed languages at BigCommerce
- Having type-safety at the contract level allowed us to avoid common pitfalls experienced in type-unaware protocols such as REST
  - For example, client libraries in Scala/Java will often throw exceptions at runtime when given invalid types
  - Having this enforced before sending prevented those issues
- Ensures the *right* data is sent and data integrity issues wouldn't occur as a consequence of poor API definition/enforcement

BIGCOMMERCE

Automatic Client/Server Code Generation

- 6+ languages at BigCommerce
  - Writing clients/servers for this was unmanageable, slowed service adoption
- Automated this through an Interface Definition Library (IDL)
  - Gave visibility to the API contracts for all services
  - Increased knowledge-sharing across the company
  - Eliminated cost of development for clients for API, gave implicit access

BIGCOMMERCE

Performance

We found a 3x increase in throughput using gRPC vs REST with all other factors equal. Most of this was found in eliminating query parsing on GET requests, and drastically reduced serialization/deserialization times for payloads.

Persistent, multiplexed connections also gave us reduced TLS overhead for SSL termination.

BIGCOMMERCE

So why’d we make gruf then?

Ruby gRPC library is only a client/server stub

While the core functionality of gRPC is excellent, there were a few things missing that we needed to use it at scale.

No interceptor support

At present, there is no Ruby interceptor support for clients/servers. We needed this for better stats management, entity marshalling, authentication, distributed tracing, and more.

Lacking error handling

Because message definition was strict, we didn't have a consistent way of passing back error payloads. Gruf pushes those into trailing metadata, making serialization and standardization of errors possible.

Improved request logging

Engineers here were familiar with Rails logs, yet grpc core logging left much to be desired. Adding in verbose request logs made developing and debugging gRPC services much easier.

BIGCOMMERCE

Gruf (Grpc RUby Framework)

We built Gruf as a sort of a framework layer over the Ruby gRPC libraries, and open-sourced it on the MIT license.

https://github.com/bigcommerce/gruf

- Gruf enabled us to provide interceptor support out of the box
  - Working with Google to have this native to gRPC Ruby core
- Provided an anti-corruption layer against gRPC libraries
  - As an early adopter, we wanted changes in gRPC to be able to be centrally protected from our services

BIGCOMMERCE

Gruf Middleware

With Gruf's middleware approach, we were able to tightly integrate with our service architecture

- Custom protobuf serialized error payloads
- Automatic statsd monitoring and distributed tracing
- Standardized auth management
- LinkerD context propagation
- Standardized test helpers

BIGCOMMERCE

Demo Time!

gruf-demo

https://github.com/bigcommerce/gruf-demo

- Rails library we open-sourced to demonstrate utilizing gRPC+gruf in a Rails application
- MIT license, feel free to copy/use
- Examples for each request type, along with the corresponding .proto file

BIGCOMMERCE

Thoughts/Caveats

Service Discovery / Load Balancing

- Recommend: LinkerD+Consul or Istio for client-side LB
  - Ensure you can use a Least Loaded algorithm to balance load
  - Consider failure accruals from the start
- Traditional LB: HAProxy
  - Easier setup, but higher latency and throughput risks
- gRPC has excellent recommendations here
- LinkerD/Consul - can utilize consul and linkerd tags to propagate context and re-route traffic on-demand through headers
  - Allows dynamic blue/greens, using integration/staging envs for dev, hot failover
  - Allows on-demand distributed tracing of requests across services
- Consider doing TLS client-side to eliminate need to deal with SSL in services

BIGCOMMERCE

Standards Are Key

- Early on determine an error and authorization strategy
  - Doing this late is *hard* due to the contractual nature of RPC
- Reuse messages if they are platform-wide and domain-agnostic
  - Don’t share request/response messages across services
- If your company has multiple languages, consider building an IDL
  - Utilize a CI tool + GitHub for automatic code gen
- Store generated client/server stubs in a separate, versioned repository from services
  - Consider a “core” repository for platform-level messages
- Always be backwards-compatible
  - Deprecate but don’t remove fields
  - Consider just adding a new method

BIGCOMMERCE

Partial Updates Are Tricky

- RPC doesn't really support partial updates of resources
  - This is because RPC is method/behavior-based, *not* resource-based
- Can be difficult when migrating PATCH REST endpoints
  - JSON has 3 states for fields: populated, null, and absent
  - Protobuf cannot detect absence/presence of a field
- Options:
  - One: Use a fetch then merged update
  - Two: Use a protobuf "field mask" (requires passing in field names as strings)
- No really good options here

BIGCOMMERCE

gRPC Libraries Rapidly Maturing

- Six-week minor version release cycle
- .0 patch releases often unstable, especially in dynamic languages (Ruby/PHP/Python)
  - Consider a 2-4 week lag on adopting new versions
- Gruf adds shielding from this
  - Can also pin gRPC versions in services due to gruf loose version coupling

BIGCOMMERCE

The End - Questions?