Our experience designing and building gRPC services

BEN IBINSON
FEBRUARY 27, 2018

This is the final post in a series on how we scaled Bugsnag's new Releases dashboard backend pipeline using gRPC. Read our first blog on why we selected gRPC for our microservices architecture, and our second blog on how we package generated code from protobufs into libraries to easily update our services.

The Bugsnag engineering team recently worked on massively scaling our backend data-processing pipeline to support the launch of the Releases dashboard. The Releases dashboard (for comparing releases to improve the health of applications) included support for sessions which would mean a significant increase in the amount of data processed in our backend. Because of this and the corresponding increase in call load, we implemented gRPC as our microservices communications framework. It allows our microservices to talk to each other in a more robust and performant way.

In this post, we'll walk through our experiences building out gRPC and some of the gotchas, gRPC examples, and development tips we've learned along the way.

Our experience with gRPC

For the Releases dashboard, we needed to implement gRPC in Ruby, Java, Node, and Go. This is because we've built the services in our data-processing Pipeline in the language best suited for the job. Our initial investigation uncovered that all client libraries have various degrees of maturity. Most of them are serviceable, but not all are feature complete. However, these libraries are progressing fast and were mature enough for our needs. Nevertheless, it's worth evaluating their current state before jumping in.

Designing a gRPC service

The gRPC service design process was much smoother than the RESTful interface in terms of the API specification. The endpoints were quickly defined, written, and understood, all self contained in a single protobuf file. However, there are very few rules on what these endpoints could be, which is generally the case with RPC. We needed to be strict on defining the role of the microservice, ensuring the endpoints reflected this role. We focused on making sure each endpoint was heavily commented, which helped our cross-continent teams avoid too many integration problems.

As an aside, it's important to write documentation and communicate common use cases involving these endpoints. This is typically outside the scope of the protobuf file, but has a large impact on what the endpoints should be.

Developing a gRPC service

Implementing gRPC was initially a rough road. Our team was unfamiliar with gRPC best practices so we had to spend more time than we wanted on building tools and testing our servers. At the time, there was a lack of good tutorials and examples for us to copy from, so the first servers created were based on trial and error. gRPC could benefit from some clear documentation and examples about the concepts it uses like stubs and channels.

This page is a good start at explaining the basics; however, we would have felt more confident if we knew more. For example, knowing how channels handle connection failure without having to check the client's source code. We also found most of the configurable options were only documented in source code and took a lot of time and effort to find. We were never really sure if the option had worked, which meant we ended up testing most options we changed. The barrier to entry for developing and testing gRPC was quite high. More intuitive documentation surrounding gRPC best practices and tools are essential if gRPC is here to stay, and there do seem to be more and more examples coming out.

Handling opinionated languages

There were a few gotchas along the way, including getting familiar with how protobufs handle default values. For example, in the protobuf format, strings are primitive and have a default value of `""`. Java developers identify `null` as the default value for strings. But beware, setting `null` for primitive protobuf fields like strings will cause runtime exceptions in Java.

The client libraries try to protect against invalid field values before transmitting and assume you are trying to set `null` to a primitive field. These safeguards are present to protect against conflicting opinions between different language applications e.g. `""`, `nil`, and `null` for strings. This led us to create wrappers for these messages to avoid confusion once you were in the application's native

language. On the whole, we've had very little need to dive into and debug the messages themselves. Client library implementations are very reliable at encoding and decoding messages.

How to debug a gRPC service

When we started using gRPC, the testing tools available were limited. Developers want to cURL their endpoints, but with gRPC equivalents to familiar tools like Postman either don't exist or are not very mature. These tools need to support both encoding and decoding messages using the appropriate protobuf file, and be able to support HTTP/2. You can actually cURL a gRPC endpoint directly, but this is far from a streamlined process. Some useful tools we came across were:

- protoc-gen-lint, linting for protobufs - This tool checks for any deviations from Google's Protocol Buffer style guide. We use this as part of our build process to enforce coding standards and catch basic errors. It's good for spotting invalid message structures and typos.
- grpcc, CLI for a gRPC Server - This uses Node REPL to interact with a gRPC service via its protobuf file, and is very useful for quickly testing an endpoint. It's a little rough around the edges, but looks promising for a standalone tool to hit endpoints.
- omgrpc, GUI client - Described as Postman for gRPC endpoints, this tool provides a visual way to interact with your gRPC services.
- awesome gRPC - A great collection of resources currently available for gRPC

Let me cURL my gRPC endpoint

In addition to these tools, we managed to re-enable our existing REST tools by using Envoy and JSON transcoding. This works by sending HTTP/1.1 requests with a JSON payload to an Envoy proxy configured as a gRPC-JSON transcoder. Envoy will translate the request into the corresponding gRPC call, with the response message translated back into JSON.

Step 1: Annotate the service protobuf file with google APIs. This is an example of a service with an endpoint that has been annotated so it can be invoked with a POST request to `/errorclass`.

```
import "google/api/annotations.proto";

package bugsnag.error_service;

service Errors {
    rpc GetErrorClass (GetErrorClassRequest) returns (GetErrorClassResponse) {
        option (google.api.http) = {
            post: "/errorclass"
```

```
            body: "*"
        };
    }
}

message GetErrorClassRequest {
    string error_id = 1;
}

message GetErrorClassResponse {
    string error_class = 1;
}
```

Step 2: Generate a proto descriptor set that describes the gRPC service. This requires the protocol compiler, or protoc installed (how to install it can be found here). Follow this guide on generating a proto descriptor set with protoc.

Step 3: Run Envoy with a JSON transcoder, configured to use the proto descriptor set. Here is an example of an Envoy configuration file with the gRPC server listening on port 4000.

```
{
  "listeners": [
    {
      "address": "tcp://0.0.0.0:3000",
      "filters": [
        {
          "type": "read",
          "name": "http_connection_manager",
          "config": {
            "codec_type": "auto",
            "stat_prefix": "grpc.error-service",
            "route_config": {
              "virtual_hosts": [
                {
                  "name": "grpc",
                  "domains": ["*"],
                  "routes": [
                    {
                      "timeout_ms": 1000,
                      "prefix": "/",
                      "cluster": "grpc-cluster"
                    }
                  ]
                }
              ]
            },
            "filters": [
              {
                "type": "both",
```

```
                "name": "grpc_json_transcoder",
                "config": {
                  "proto_descriptor": "/path/to/proto-descriptors.pb",
                  "services": ["bugsnag.error_service"],
                  "print_options": {
                    "add_whitespace": false,
                    "always_print_primitive_fields": true,
                    "always_print_enums_as_ints": false,
                    "preserve_proto_field_names": false
                  }
                }
              },
              {
                "type": "decoder",
                "name": "router",
                "config": {}
              }
            ]
          }
        }
      ]
    }
  ],
  "admin": {
    "access_log_path": "/var/log/envoy/admin_access.log",
    "address": "tcp://0.0.0.0:9901"
  },
  "cluster_manager": {
    "clusters": [
      {
        "name": "grpc-cluster",
        "connect_timeout_ms": 250,
        "type": "strict_dns",
        "lb_type": "round_robin",
        "features": "http2",
        "hosts": [
          {
            "url": "tcp://docker.for.mac.localhost:4000"
          }
        ]
      }
    ]
  }
}
```

Step 4: cURL the gRPC service via the proxy. In this example, we set up the proxy to listen to port 3000.

```
> curl -H "Accept: application/json" \
       -X POST -d '{"error_id":"587826d70000000000000001"}' \
```

```
        http://localhost:3000/errorclass

'{"error_class":"Custom Runtime Exception"}'
```

Although this technique can be very useful, it does require us to "muddy up" our protobuf files with additional dependencies, and manage the Envoy configurations to talk to these services. To streamline the process, we scripted the steps and ran an Envoy instance inside a Docker container, taking a protobuf file as a parameter. This allowed us to quickly set a JSON transcoding proxy for any gRPC services in seconds.

Running the gRPC Ruby client library on Alpine

We did encounter some trouble running the Ruby version of a gRPC client. When we came to build the applications container, we got the error:

```
LoadError: Error relocating /app/vendor/bundle/ruby/2.4.0/gems/grpc-1.4.1-x86_64-linux/src/ru
```

Most gRPC client libraries are written on top of a shared core library, written in C. The issue was due to using the an alpine version of Ruby with a precompiled version of the gRPC library requiring `glibc`. This was solved by setting `BUNDLE_FORCE_RUBY_PLATFORM=1` in the environment when running `bundle install` which will build the gems from source rather than using the precompiled version.

Final thoughts

Rolling gRPC services into production, we immediately observed latency improvements. Once the initial connection was made, transport costs were on the order of microseconds and effectively negligible compared to the call itself. This gave us confidence to ramp up to heavier loads which were handled with ease.

Now that we've streamlined our development process, and made our deployments resilient using Envoy, we can rollout new scalable gRPC communication links or upgrade existing ones quickly and efficiently. Load balancing did provide us with an interesting problem, and you can read about it here.