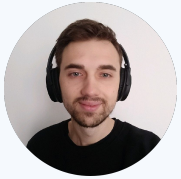
Using Protocol Buffers and gRPC at scale

Introduction



Evgeny
Engineering Team Lead

MessageBird powers communication between businesses and their customers — across any channel, always with the right context, and on every corner of the planet. If you've ever ordered takeaway, returned a package, contacted customer service or requested a login code, it's almost guaranteed your interactions have been powered by MessageBird's technology.

7B+ Devices
MessageBird's global business reaches over 7 billion devices.

25,000+ Customers
Customers in over 60+ countries, across a great variety of industries.

700+ Employees
More than 700 employees who represent more than 55 nationalities.

Agenda

1. **Introduction to Protocol Buffers and gRPC**
   Short introduction to get everybody up to speed

2. **Why you should consider it?**
   Reasons to choose Protocol Buffers and gRPC

3. **How is it used in MessageBird**
   Tips and tricks for using Protocol Buffers and gRPC in a large organization

Protocol Buffers overview

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way.

It’s like JSON, except it's smaller and faster, and it generates native language bindings.

C++ | C# | Dart | Go | Java | Kotlin | Python | etc.

Source: Protocol Buffer docs

Protocol buffers are ideal for any situation in which you need to serialize typed data in extensible manner. They are most often used for defining communications protocols (together with gRPC) and for data storage.

Some of the advantages of using protocol buffers include:

- **Compact data storage**
- **Fast parsing**
- **Availability in many programming languages**
- **Optimized functionality through automatically-generated classes**

Protocol Buffers overview

- **Protocol Buffers specification covers two protocol versions: proto2 and proto3.**
- **Supported types:**
  - Scalar Types
  - Enumerations
  - Repeated Fields (arrays)
  - Maps
  - Oneof (unions)
- **Protobuf ships with pre-defined types:**
  - Wrappers, such as StringValue, Int32Value, BytesValue.
  - Built-in types for duration, timestamp, etc.
  - See all built-in types here.
- **Difference between default and empty values.**

Simple example of a search request message:

- First line specifies that you are using proto3 syntax.
- The SearchRequest message definition has three fields (name/value pairs).
- Each field has a name and a type.

message.proto

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

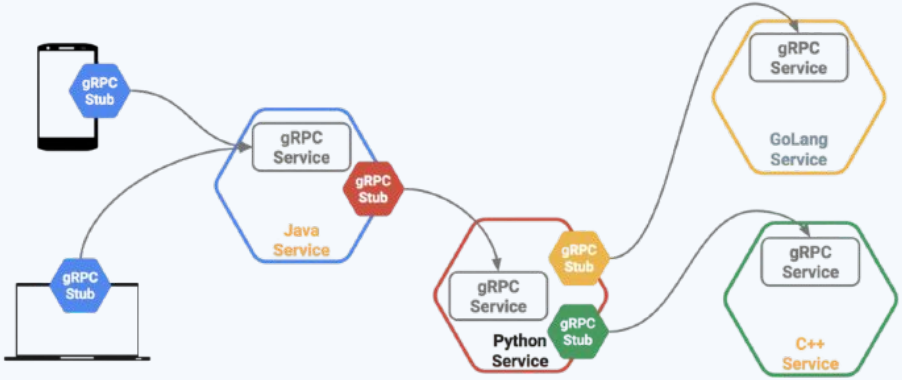
How do Protocol Buffers work?

Schema definition	Code generation	Compilation	Runtime usage
Create .proto file to define data structures and services	Generate code using protoc compiler	Compile generated protobuf code	Use protobuf classes to serialize, share and deserialize data
Output .proto files	**Output** .java, .go, .py, etc.	**Output** compiled code classes	

What is gRPC?

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment.

- Simple service definitions.
- It automatically generates client and server stubs for your service in a variety of languages.
- Unary calls and bi-directional streaming support with HTTP/2-based transport.



Source: gRPC [docs]

Greeter Service example

service.proto

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

server.go

```
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloReply, error) {
	return &pb.HelloReply{
		Message: "Hello, " + in.GetName()
	}, nil
}
```

client.go

```
resp, err = client.SayHello(ctx, &pb.HelloRequest{Name: name})
if err != nil {
	log.Fatalf("could not greet: %v", err)
}
log.Printf("Greeting: %s", resp.GetMessage())
```

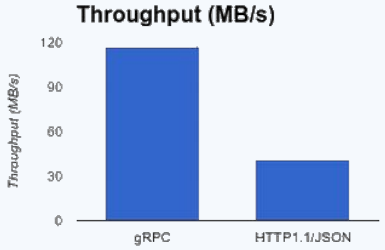
Why Protocol Buffers and gRPC?

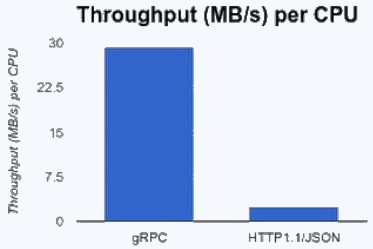
It’s crucial for high-load services to utilize efficient encoding and communicate via a fast and reliable channel

Efficient Data Encoding and Transport Layer

- Significant difference in throughput per CPU means you need less resources.
- Two major factors behind this difference:
  - Efficient binary data encoding
    - gRPC keeps data in binary both in the client memory and on the wire
    - Binary encoding is more efficient in processing time and space requirements compared to Base64 or JSON
  - Efficient transport build on top of HTTP/2
    - Header compression
    - Multiplexed requests over a single connection

Source: [Announcing gRPC Alpha for Google Cloud Pub/Sub]





Dealing with Binary Data and gRPC Services

Binary Encoding

To deserialize binary data, you need to know the message type and use a compatible version of generated classes in the code.

Making a gRPC call

A developed and rich ecosystem offers a variety of tools to make interaction with gRPC services seamless.

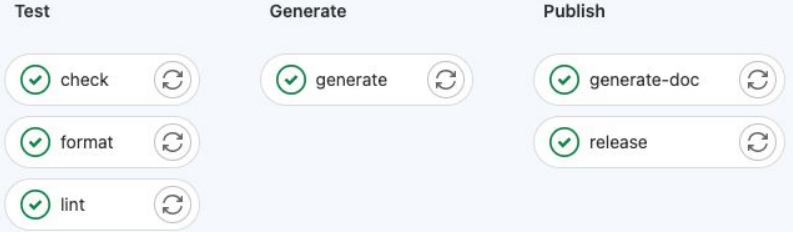
- grpcurl - Like cURL, but for gRPC. Command-line tool for interacting with gRPC servers.
- grpcui - An interactive web UI for gRPC, similar to Postman.

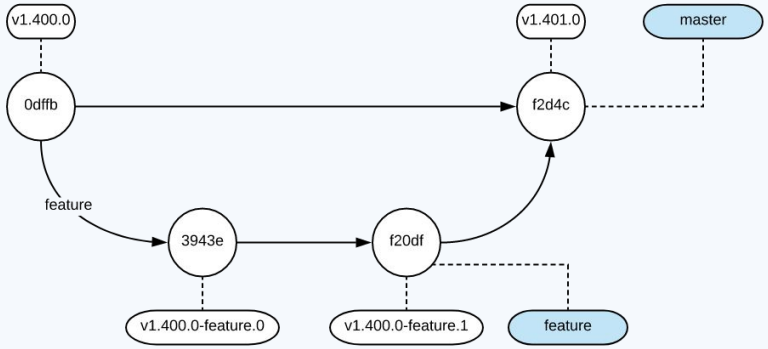
```
$ grpcurl grpc.server.com:443 greet.Greeter/SayHello \
       -d '{"name": "World"}'

{
  "message": "Hello, World"
}
```

Automation and CI/CD

- **protobuf-source** - git repository with data structure and service definitions.
- CI/CD pipeline consists of the following stages:
  - **Test stage.**
    - prototool is used for checking, linting and formatting proto files.
  - **Generate stage** produces code files using protoc compiler in multiple languages.
  - **Publish stage.**
    - protoc-gen-doc plugin is used for generating internal documentation.
    - Release script pushes changes to the corresponding **protobuf-gen-*** git repositories following semantic versioning.





Monitoring and Observability

- Both server and client expose metrics via Prometheus:
  - Total number of started and completed RPCs, regardless of success or failure.
  - Total number of RPC stream messages sent and received.
  - Histogram of response latency of started and completed RPCs.
- OpenTelemetry tracing to monitor and troubleshoot calls in complex distributed systems.
  - Always propagate tracing context.
  - Use a reasonable sampling rate.
- Each service has a dedicated monitoring dashboard.
- Well-defined defined SLIs (service level indicators) and SLO (service level objectives) for business critical applications.

gRPC protocol supports interceptors, i.e. middleware that is executed either on the server-side before the request is passed onto the user's application logic, or on the client-side around the user call. It's a perfect place to implement logging or monitoring.

Tips and Tricks

- Centralized git repository with data structures and service definitions acts as “source of truth”. It also defines a “contract” between various teams in the company.
- Load balancing is tricky. See this guide.
  - Proxy or client-side load balancing.
  - Transport (L3/L4) or Application (L7).
  - Service mesh setup.
- Debugging or solving performance issues is quite challenging.
  - Channelz - a tool that provides comprehensive runtime info about connections at different levels in gRPC.
- Additional tooling is required to inspect binary data or making an RPC call.
  - Invest time in building and owning your own internal tools.
  - Reverse-proxies make it easy to translate HTTP/JSON requests to gRPC calls.
    - grpc-gateway - gRPC to JSON proxy generator.
    - Read more on gRPC support in Envoy.
- Check awesome-grpc - curated list of useful resources.

Thanks for your attention!

Questions?