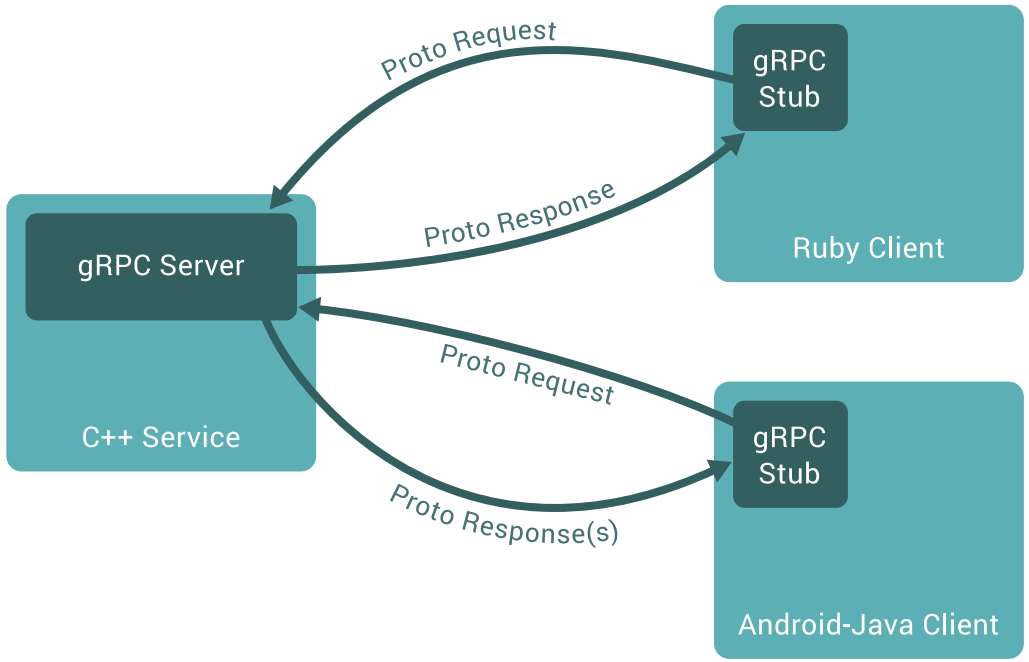
Intro to gRPC

Phaser Deep Dive
17 January 2019

Gian Biondi
Senior Software Engineer, Jet.com

What is gRPC?

gRPC is an open source, high performance RPC framework from Google. It is cross platform and language agnostic. It is primarily used for connecting polyglot services.



2

What is an RPC?

```
In distributed computing, a remote procedure call (RPC) is when a computer
program causes a procedure (subroutine) to execute in a different address space
(commonly on another computer on a shared network), which is coded as if it
were a normal (local) procedure call, without the programmer explicitly coding
the details for the remote interaction
- Wikipedia
```

3

What is Protobuf?

- gRPC uses Protocol Buffers as the message format.
- Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data.
- Protobufs are language neutral, platform neutral, and extensible.
- comes from Google
- open source

4

Why use Protobuf?

1. Backwards compatibility

- New fields will be discarded by services with older version of `.proto` file

2. Fixed Schema

- Typed schema provides validation on serialization/deserialization

3. Space efficient (3-10x smaller)

- Binary-Encoding, varint encoding, and no field names or overhead characters

4. Faster (20-100x faster)

- Binary-Encoding is more efficiently processed than recursively parsing string-encoded message and reflecting on type (if applicable)

5

How to use Protobuf?

- specify the structure of the data in a `.proto` file
- Each protobuf message is a very small (less than 1MB) logical record, consisting of a series of key-value pairs
- You distribute the `.proto` file to all the services communicating with that message
- compile it for the language of each service

6

A Protobuf Message

```
syntax = "proto3";

message Contact {
    string name = 1;
    int32 id = 2;
    string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
    }

    message PhoneNumber {
        string number = 1;
        PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}
```

7

How does Protobuf work?

1. Protobuf compiler generate *SerDes* code for a target language
2. When it serializes a protobuf message in code, it converts each field to its *wire type* and concatenates them together into a single byte stream, ready to send
3. The deserializer reads the byte stream sequentially, looking for a set of start bytes starting each field identifying the field, the *wire type*, and followed by the data
4. The stream is deserialized into a protobuf-compiler-generated object (for the target language)

8

Protobuf Encoding

9

Length Delimited Encoding

Strings are encoded as a series of bytes prefixed by a length. This goes for repeated fields, and embedded messages as well.

For Example:

```
12 07 | 74 65 73 74 69 6e 67
0x12 → Field Number = 2, type = 2
0x07 → Seven Bytes following
Message spells out "testing"
```

10

Base 128 VarInt encoding

VarInt incoding is a way to serialize integers using one or more bytes.

For Example: 300

```
Encoded in VarInt
1010 1100 0000 0010
# (44034)

# Uses only lower 7 bits (base 128);
# MSB is reserved to indicate continuation.
1010 1100 0000 0010
→ 010 1100  000 0010

# All Lower 7 Bits Concatenated (Little Endian)
000 0010  010 1100
# Concat the bytes
→  000 0010 ++ 010 1100
→  100101100
→  256 + 32 + 8 + 4 = 300
```

11

Fixed Encoding

- Certain types (float, fixedInt32, etc) always consume a fixed number of bytes
- Using negative `int*` will cause numbers to drop out of VarInt encoding and always consume ten bytes (Use `sint32` instead)

12

Protobuf Types

Protobuf supports a bunch of scalar types:

```
Type    Meaning             Used For
0        Varint              int32, int64, uint32, uint64, sint32, sint64, bool, enum
1        64-bit              fixed64, sfixed64, double
2        Length-delimited    string, bytes, embedded messages, packed repeated fields
3        Start group         groups (deprecated)
4        End group           groups (deprecated)
5        32-bit              fixed32, sfixed32, float

Decoding: (field_number << 3) | wire_type
```

13

Field Ordering

```
    string host = 1;
    string path = 2;
    string x_forwarded_for = 3;
```

- Specify order by using unique numeric tags for each field starting with 1, up to 536,870,911 (excluding 19000-19999 reserved for protobuf)
- Should keep the number of tags at 15 to use only one byte
- You can use field numbers in any order
- When serialized, the message will be in the correct order
- The sequential nature of the byte stream allows the decoder to work fast
- The decoder can also act on fields in any order.

From Protobuf to gRPC Service

15

Sample gRPC Proto

```
import 'google/protobuf/empty.proto';

service Core {
    rpc Assign (AssignmentRequest) returns (PhaserResponse){}
    rpc FlushCache(google.protobuf.Empty) returns (google.protobuf.Empty){}
}

message AssignmentRequest {
    string host = 1;
    string path = 2;
    string x_forwarded_for = 3;
    string user_agent = 4;
    string referer = 5;
    string dr_orpheus_header = 6;
    string jet_api_client = 7;
    map<string,string> supplemental_headers = 8;
}

message PhaserResponse {
    string build_header = 1;
    string dr_orpheus_header = 2;
}
```

16

Sample gRPC Client Implementation

```
import (
    "context"
    "log"
    "time"

    pb "go.jet.network/phaser/core-service/proto"
    "google.golang.org/grpc"
)

func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial("localhost:8080", grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    c := pb.NewCoreClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.Assign(ctx, &pb.AssignmentRequest{DrOrpheusHeader: "Dr. O: Phaser!"})
    if err != nil {
        log.Fatalf("could not assign: %v", err)
    }
}
```

17

Sample gRPC Server Implementation

```
type PhaserServer struct{}

// Assign Stub
func (ps *PhaserServer) Assign(ctx context.Context,
    req *pb.AssignmentRequest) (*pb.PhaserResponse, error) {
    pd := "Some Phaser Assignment"
    return &pb.PhaserResponse{DrOHeader: pd}, nil
}

func main() {
    lis, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatalf("Failed to listen: %v", err)
    }
    s := grpc.NewServer()

    // this will satisfy the interface laid out in pb.go file
    var ps *PhaserServer
    // Defined in Autogenerated pb.go file
    pb.RegisterCoreServer(s, ps)
    if err := s.Serve(lis); err != nil {
        log.Fatalf("Failed to serve: %v", err)
    }
}
```

gRPC Methods

gRPC gives us four kinds of methods:

1. Unary RPC
2. Server-Streaming RPC
3. Client-Streaming RPC
4. BiDirectional-Streaming RPC

19

HTTP/2

gRPC transmits data over HTTP/2 Protocol:
- Update to HTTP/1.1 Protocol
- Spec Released May 2015

Features

- Single TCP connection for multiple streams
- Server Push
- Supports **Binary Protocols**
- Multiplexed Streams
- Stream prioritization
- HPACK Stateful Header Compression

gRPC Feature Summary

- Idiomatic client libraries in 10 languages
- Highly efficient on wire and with a simple service definition framework
- Bi-directional streaming with http/2 based transport
- Pluggable auth, tracing, load balancing and health checking

Thank you

Gian Biondi
Senior Software Engineer, Jet.com
gianfranco.biondi@jet.com (mailto:gianfranco.biondi@jet.com)