KubeCon | CloudNativeCon

North America 2017

Modifying gRPC Services over Time

Eric Anderson, Staff Sw Eng, *Google*

Intended Audience

Have toyed with gRPC

- Making a service, maybe streaming
- Code generation

Have toyed with Protocol Buffers

- Know some data types, primitives vs messages

Searching for best practices/idioms

Assumptions

Cross-language/platform

- No language-specific tricks

gRPC-native

- Aware of REST+JSON, but not a focus

Servers are updated before Clients

- Invalid for intra-service communication, P2P
- Invalid for services provided by multiple implementations

What Type of Compatibility?

- Binary Compatibility (generated code ABI)
- Source Compatibility (generated code API)
- Wire Compatibility
- Behavior Compatibility

What Type of Compatibility?

- Binary Compatibility (generated code ABI)
- Source Compatibility (generated code API)
- Wire Compatibility
- Behavior Compatibility

Binary/Source Compatibility

Most bets off, across:
C++, Java, Python, Go, Ruby, C#, Node.js, ObjC, PHP

Removals and renames break API

Message type changes trivially break API/ABI

Primitive type changes break many ABIs, some APIs

Can add services, methods, messages, fields

Library Service (conceptual)

```
package google.example.library;

service LibraryService {
  rpc CreateBook(Book) returns (Book);
  rpc GetBook(string) returns (Book);
  rpc ListBooks(google.protobuf.Empty)
      returns (ListBooksResponse);
  rpc DeleteBook(string)
      returns (google.protobuf.Empty);
  rpc UpdateBook(Book) returns (Book);
}
```

Best Practices

Plan for replacement of the service:

- Include version number in package

Use messages as argument/return extensibility mechanism:

- When in doubt, give each RPC its own request/response messages

Add new services and methods as necessary:

- Avoid too many, but also avoid kitchen-sink `Do()` methods

Library Service (improved)

```
package google.example.library.v1;

service LibraryService {
  rpc CreateBook(CreateBookRequest) returns (Book);
  rpc GetBook(GetBookRequest) returns (Book);
  rpc ListBooks(ListBooksRequest)
      returns (ListBooksResponse);
  rpc DeleteBook(DeleteBookRequest)
      returns (google.protobuf.Empty);
  rpc UpdateBook(UpdateBookRequest) returns (Book);
}
```

Clock Service

```
package acme.infra;

service Clock {
  rpc GetTime (google.protobuf.Empty)
      returns (google.protobuf.Timestamp);
}
```

Clock Service (improved)

```
package acme.infra.v1;

service Clock {
  rpc GetTime (GetTimeRequest)
      returns (GetTimeResponse);
}

message GetTimeRequest {}
message GetTimeResponse {
  google.protobuf.Timestamp timestamp = 1;
}
```

What Type of Compatibility?

- Binary Compatibility (generated code ABI)
- Source Compatibility (generated code API)
- Wire Compatibility
- Behavior Compatibility

Wire Compatibility

RPC only distinguished by its name. “package.Service/Method”

Request/response types are implicit

Cardinality (unary vs streaming) is implicit

- Can add new services and methods
- Can add “stream” keyword (risky)
- Can rename/copy message (risky)

Wire Compatibility - Protobuf

Message names implicit, except when they’re not (i.e., Any)
Field names implicit, except when they’re not
(i.e., JSON, FieldMask)
Field tags explicit
Some types are same (int32 is equivalent to int64 on-wire)

- Can add new messages and fields
- Field can be made repeated (risky)
- Can “upgrade” some integer primitives (risky)

Best Practices

Limited by binary/source compatibility. Same practices

(But more tricks/hacks available if in control of
both client and server and a small enough project)

What Type of Compatibility?

- Binary Compatibility (generated code ABI)
- Source Compatibility (generated code API)
- Wire Compatibility
- Behavior Compatibility

Behavior Compatibility

Wide scope. Most effort likely spent here

Application-specific, but many common patterns

Best Practices

New primitive fields default to `0`, “”

Avoid updates clearing new fields

Use `google.rpc.Status` for error details

Break long-lived RPC occasionally

Best Practices

New primitive fields default to `0`, “”

Avoid updates clearing new fields

Use `google.rpc.Status` for error details

Break long-lived RPC occasionally

New primitive fields default to `0`, “”

Work with it instead of against it

If necessary, use `wrappers.proto` to box

```
// Wrapper message for `string`.
message StringValue {
  // The string value.
  string value = 1;
}
```

If multiple fields related, can make custom message to box once

Best Practices

New primitive fields default to `0`, “”

Avoid updates clearing new fields

Use `google.rpc.Status` for error details

Break long-lived RPC occasionally

Avoid updates clearing new fields

```
service LibraryService {
  rpc UpdateBook(UpdateBookRequest) returns (Book);
}
message UpdateBookRequest { Book book = 1; }
message Book {
  // The name is ignored when creating a book.
  string name = 1;
  string author = 2;
  string title = 3;
}
```

Avoid updates clearing new fields

```
service LibraryService {
  rpc UpdateBook(UpdateBookRequest) returns (Book);
}

message UpdateBookRequest { Book book = 1; }
message Book {
  // The name is ignored when creating a book.
  string name = 1;
  string author = 2;
  string title = 3;
  bool read = 4; // Users report they get bored
}
```

Avoid updates clearing new fields

FieldMask is partial solution

- Clients specify only the pieces they are updating:
  ```
  paths: "author"
  paths: "submessage.submessage.field"
  ```

Empty FieldMask defaults to “all fields”
Annoying to use all the time

Avoid updates clearing new fields

```
service LibraryService {
  rpc UpdateBook(UpdateBookRequest) returns (Book);
}

message UpdateBookRequest {
  Book book = 1;
  google.protobuf.FieldMask mask = 2;
}
```

Avoid updates clearing new fields

```
String name = request.getBook().getName();
Book.Builder book = db.getBook(name).toBuilder();
// Do consult MergeOptions
FieldMaskUtil.merge(request.getMask(),
                    request.getBook(),
                    book);
db.updateBook(name, book.build());
```

Avoid updates clearing new fields

REST-oriented style guide recommends against new fields

Protobuf 3.5 re-adds unknown field support

- Allows clients to do normal Get(), modify, Update()
- Unknown fields then not cleared

Best Practices

New primitive fields default to `0`, “”

Avoid updates clearing new fields

Use `google.rpc.Status` for error details

Break long-lived RPC occasionally

Use google.rpc.Status for error details

Encode error details in messages

error_details.proto provides some common error details. e.g.,

- DebugInfo (stack trace)
- QuotaFailure
- Help (link to documentation)
- LocalizedMessage

Use `google.rpc.Status` for error details

Used to recommend placing messages Metadata

Hard to pass around in some languages

Unclear which Metadata is error-related

Use google.rpc.Status for error details

```
message Status {
  int32 code = 1; // same as gRPC
  string message = 2; // same as gRPC
  repeated google.protobuf.Any details = 3;
}
```

Trivia: originally used by gRPC

At the time, `details` dropped since there was Metadata

Use google.rpc.Status for error details

Languages receiving utilities to work with `google.rpc.Status`

```
io.grpc.Status s =
     io.grpc.Status.fromThrowable(t);
// vs
com.google.rpc.Status s =
     io.grpc.protobuf.StatusProto.fromThrowable(t);
```

Best Practices

New primitive fields default to `0`, “”

Avoid updates clearing new fields

Use `google.rpc.Status` for error details

Break long-lived RPC occasionally

Break long-lived RPC occasionally

Clients can inadvertently avoid polling behavior

“Complete” RPC prematurely, after an age or random

Can free up connections, but mainly for client code health

Q&A

GitHub/Gitter: @ejona86

Email: ejona@google.com

IRC: #grpc   :-D