Programming Models for
DISTRIBUTED COMPUTING

gRPC

BY PAUL GROSU (NORTHEASTERN U.), MUZAMMIL ABDUL REHMAN (NORTHEASTERN U.), ERIC ANDERSON (GOOGLE, INC.), VIJAY PAI (GOOGLE, INC.), AND HEATHER MILLER (NORTHEASTERN U.)

(14 min read)

—

gRPC

Paul Grosu (Northeastern U.), Muzammil Abdul Rehman (Northeastern U.), Eric Anderson (Google, Inc.), Vijay Pai (Google, Inc.), and Heather Miller (Northeastern U.)

Abstract

gRPC has been built from a collaboration between Google and Square as a public replacement of Stubby, ARCWire and Sake (missing reference). The gRPC framework is a form of an Actor Model based on an IDL (Interface Description Language), which is defined via the Protocol Buffer message format. With the introduction of HTTP/2 the internal Google Stubby and Square Sake frameworks are now been made available to the public. By working on top of the HTTP/2 protocol, gRPC enables messages to be multiplexed and compressed bi-directionally as premptive streams for maximizing capacity of any microservices ecosystem. Google has also a new approach to public projects, where instead of just releasing a paper describing the concepts will now also provide the implementation of how to properly interpret the standard.

Introduction

In order to understand gRPC and the flexibity of enabling a microservices ecosystem to become into a Reactive Actor Model, it is important to appreciate

the nuances of the HTTP/2 Protocol upon which it is based. Afterward we will describe the gRPC Framework - focusing specifically on the gRPC-Java implementation - with the scope to expand this chapter over time to all implementations of gRPC. At the end we will cover examples demonstrating these ideas, by taking a user from the initial steps of how to work with the gRPC-Java framework.

1 *HTTP/2*

The HTTP 1.1 protocol has been a success for some time, though there were some key features which began to be requested by the community with the increase of distributed computing, especially in the area of microservices. The phenomenon of creating more modularized functional units that are organically constructed based on a *share-nothing model* with a bidirectional, high-throughput request and response methodology demands a new protocol for communication and integration. Thus the HTTP/2 was born as a new standard, which is a binary wire protocol providing compressed streams that can be multiplexed for concurrency. As many microservices implementations currently scan header messages before actually processing any payload in order to scale up the processing and routing of messages, HTTP/2 now provides header compression for this purpose. One last important benefit is that the server endpoint can actually push cached resources to the client based on anticipated future communication, dramatically saving client communication time and processing.

1.1 *HTTP/2 Frames*

The HTTP/2 protocol is now a framed protocol, which expands the capability for bidirectional, asynchronous communication. Every message is thus part of a frame that will have a header, frame type and stream identifier aside from the standard frame length for processing. Each stream can have a priority, which allows for dependency between streams to be achieved forming a *priority tree*. The data can be either a request or response which allows for the bidirectional communication, with the capability of flagging the communication for stream termination, flow control with priority settings, continuation and push responses from the server for client confirmation. Below is the format of the HTTP/2 frame (missing reference):

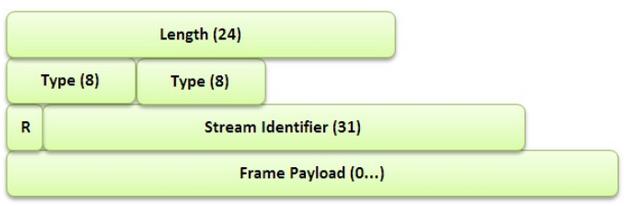

Figure 1: The encoding a HTTP/2 frame.

1.2 *Header Compression*

The HTTP header is one of the primary methods of passing information about the state of other endpoints, the request or response and the payload. This enables endpoints to save time when processing a large quantity to streams, with the ability to forward information along without wasting time to inspect the payload. Since the header information can be quite large, it is possible to now compress the them to allow for better throughput and capacity of stored stateful information.

1.3 *Multiplexed Streams*

As streams are core to the implementation of HTTP/2, it is important to discuss the details of their implemenation in the protocol. As many streams can be open simultanously from many endpoints, each stream will be in one of the following states. Each stream is multiplexed together forming a chain of streams that are transmitted over the wire, allowing for asynchronous bi-directional concurrency to be performed by the receiving endpoint. Below is the lifecycle of a stream (missing reference):

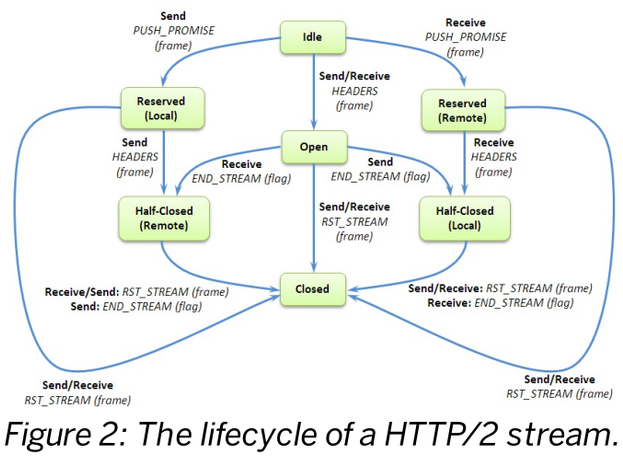

Figure 2: The lifecycle of a HTTP/2 stream.

To better understand this diagram, it is important to define some of the terms in it:

PUSH_PROMISE - This is being performed by one endpoint to alert another that it will be sending some data over the wire.

RST_STREAM - This makes termination of a stream possible.

PRIORITY - This is sent by an endpoint on the priority of a stream.

END_STREAM - This flag denotes the end of a *DATA* frame.

HEADERS - This frame will open a stream.

Idle - This is a state that a stream can be in when it is opened by receiving a *HEADERS* frame.

Reserved (Local) - To be in this state is means that one has sent a PUSH_PROMISE frame.

Reserved (Remote) - To be in this state is means that it has been reserved by a remote endpoint.

Open - To be in this state means that both endpoints can send frames.

Closed - This is a terminal state.

Half-Closed (Local) - This means that no frames can be sent except for *WINDOW_UPDATE*, *PRIORITY*, and *RST_STREAM*.

Half-Closed (Remote) - This means that a frame is not used by the remote endpoint to send frames of data.

1.4 *Flow Control of Streams*

Since many streams will compete for the bandwidth of a connection, in order to prevent bottlenecks and collisions in the transmission. This is done via the *WINDOW_UPDATE* payload for every stream - and the overall connection as well - to let the sender know how much room the receiving endpoint has for processing new data.

2 Protocol Buffers with RPC

Though gRPC was built on top of HTTP/2, an IDL had to be used to perform the communication between endpoints. The natural direction was to use Protocol Buffers is the method of stucturing key-value-based data for serialization between a server and client. At the time of the start of gRPC development only version 2.0 (proto2) was available, which only implemented data structures without any request/response mechanism. An example of a Protocol Buffer data structure would look something like this:

```
// A message containing the user's
message Hello {
  string name = 1;
}
```

Figure 3: Protocol Buffer version 2.0 representing a message data-structure.

This message will also be encoded for highest compression when sent over the wire. For example, let us say that the message is the string *"Hi"*. Every Protocol Buffer type has a value, and in this case a string has a value of `2`, as noted in the Table 1 (missing reference).

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

Table 1: Tag values for Protocol Buffer types.

One will notice that there is a number associated with each field element in the Protocol Buffer definition, which represents its *tag*. In Figure 3, the field `name` has a tag of `1`. When a message gets encoded each field (key) will start with a one byte value (8 bits), where the least-significant 3-bit value encode the *type* and the rest the *tag*. In this case tag which is `1`, with a type of 2. Thus the encoding will be `00001 010`, which has a hexdecimal value of `A`. The following

byte is the length of the string which is `2`, followed by the string as `48` and `69` representing `H` and `i`. Thus the whole tranmission will look as follows:

```
A 2 48 69
```

Thus the language had to be updated to support gRPC and the development of a service message with a request and a response definition was added for version version 3.0.0 of Protocol Buffers. The updated implementation would look as follows (missing reference):

```
// The request message containing t
message HelloRequest {
  string name = 1;
}

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

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

Figure 4: Protocol Buffer version 3.0.0 representing a message data-structure with the accompanied RPC definition.

Notice the addition of a service, where the RPC call would use one of the messages as the structure of a *Request* with the other being the *Response* message format.

Once of these Proto file gets generated, one would then use them to compile with gRPC to for generating the *Client* and *Server* files representing the classical two endpoints of a RPC implementation.

3 *gRPC*

gRPC was built on top of HTTP/2, and we will cover the specifics of gRPC-Java, but expand it to all the implementations with time. gRPC is a cross-platform framework that allows integration across many languages as denoted in Figure 5 (missing reference).

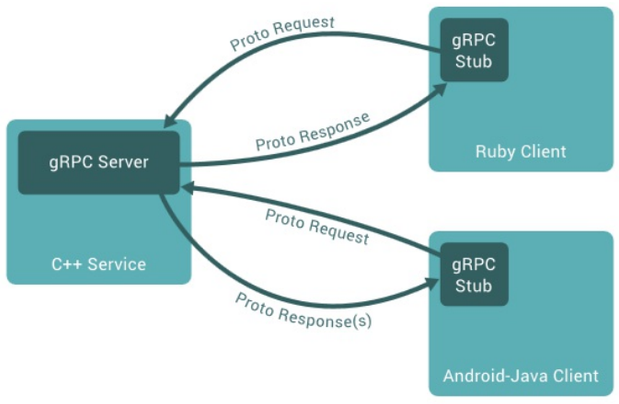

Figure 5: gRPC allows for asynchronous language-agnostic message passing via Protocol Buffers.

To ensure scalability, benchmarks are run on a daily basis to ensure that gRPC performs optimally under high-throughput conditions as illustrated in Figure 6 (missing reference).

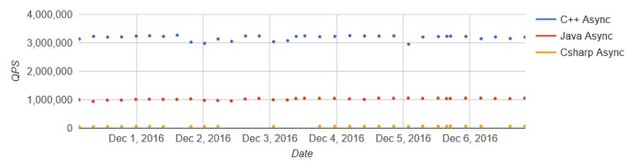

Figure 6: Benchmark showing the queries-per-second on two virtual machines with 32 cores each.

To standardize, most of the public Google APIs - including the Speech API, Vision API, Bigtable, Pub/Sub, etc. - have been ported to support gRPC, and their definitions can be found at the following location:

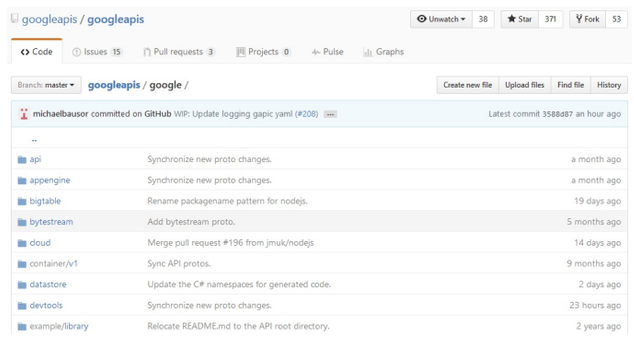

Figure 7: The public Google APIs have been updated for gRPC, and be found at https://github.com/googleapis/googleapis/tree/master/google

3.1 *Supported Languages*

The officially supported languages are listed in Table 2 (missing reference).

Language	Platform	Compiler
C/C++	Linux	GCC 4.4
		GCC 4.6
		GCC 5.3
		Clang 3.5
		Clang 3.6
		Clang 3.7
C/C++	Windows 7+	Visual Studio 2013+
C#	Windows 7+	.NET 4.5+
	Linux	Mono 4+
	Mac	Mono 4+
Node.js	Windows/Linux/Mac	Node v4+
PHP *	Linux/Mac	PHP 5.5+ and PHP 7.0+
Ruby	Windows/Linux/Mac	
Python	Windows/Linux/Mac	Python 2.7 and Python 3.4+
Go	Windows/Linux/Mac	Go 1.5+
Java	Windows/Linux/Mac	JDK 8 recommended. Gingerbread+ for Android

* still in beta

Table 2: Officially supported languages by gRPC.

3.2 *Authentication*

There are two methods of authentication that are available in gRPC:

- SSL/TLS
- Google Token (via OAuth2)

gRPC is flexible in that once can plug in their custom authentication system if that is preferred.

3.3 *Development Cycle*

In its simplest form gRPC has a structured set of steps one goes about using it, which has this general flow:

1. Download gRPC for the language of interest.

2. Implement the Request and Response definition in a ProtoBuf file.

3. Compile the ProtoBuf file and run the code-generators for the the specific language. This will generate the Client and Server endpoints.

4. Customize the Client and Server code for the desired implementation.

Most of these will require tweaking the Protobuf file and testing the throughput to ensure that the network and CPU capacities are optimally maximized.

3.4 *The gRPC Framework (Stub, Channel and Transport Layer)*

One starts by initializing a communication *Channel* between *Client* to a *Server* and storing that as a *Stub*. The *Credentials* are provided to the Channel when being initialized. These form a *Context* for the Client's connection to the Server. Then a *Request* can be built based on the definition in the Protobuf file. The Request and associated expected*Response* is executed by the *service* constructed in the Protobuf file. The Response is them parsed for any data coming from the Channel.

The connection can be asynchronous and bi-directionally streaming so that data is constantly flowing back and available to be read when ready. This allows one to treat the Client and Server as endpoints where one can even adjust not just the flow but also intercept and decoration to filter and thus request and retrieve the data of interest.

The *Transport Layer* performs the retrieval and placing of binary protocol on the wire. For *gRPC-Java* has three implementations, though a user can implement their own: *Netty, OkHttp, and inProcess.*

3.5 *gRPC Java*

The Java implementation of gRPC been built with Mobile platform in mind and to provide that capability it requires JDK 6.0 to be supported. Though the core of gRPC is built with data centers in mind - specifically to support C/C++ for the Linux platform - the Java and Go implementations are two very reliable platform to experiment the microservice ecosystem implementations.

There are several moving parts to understanding how gRPC-Java works. The first important step is to ensure that the Client and Server stub inferface code get generated by the Protobuf plugin compiler. This is usually placed in your *Gradle* build file called `build.gradle` as follows:

```
   compile 'io.grpc:grpc-netty:1.0.1
   compile 'io.grpc:grpc-protobuf:1.
   compile 'io.grpc:grpc-stub:1.0.1'
```

When you build using Gradle, then the appropriate base code gets generated for you, which you can override to build your preferred implementation of the Client and Server.

Since one has to implement the HTTP/2 protocol, the chosen method was to have a *Metadata* class that will convert

the key-value pairs into HTTP/2 Headers and vice-versa for the Netty implementation via *GrpcHttp2HeadersDecoder* and *GrpcHttp2OutboundHeaders*.

Another key insight is to understand that the code that handles the HTTP/2 conversion for the Client and the Server are being done via the *NettyClientHandler.java* and *NettyServerHandler.java* classes shown in Figures 8 and 9.

Figure 8: The Client Tranport Handler for gRPC-Java.

Figure 9: The Server Tranport Handler for gRPC-Java.

3.5.1 *Downloading gRPC Java*

The easiest way to download the gRPC-Java implementation is by performing the following command:

```
git clone -b v1.0.0 https://github.
```

Next compile on a Windows machine using Gradle (or Maven) using the following steps - and if you are using any Firewall software it might be necessary to temporarily disable it while compiling gRPC-Java as sockets are used for the tests:

```
cd grpc-java
set GRADLE_OPTS=-Xmx2048m
set JAVA_OPTS=-Xmx2048m
set DEFAULT_JVM_OPTS="-Dfile.encodi
echo skipCodegen=true > gradle.prop
gradlew.bat build -x test
cd examples
gradlew.bat installDist
```

If you are having issues with Unicode (UTF-8) translation when using Git on Windows, you can try the following commands after entering the `examples` folder:

```
wget https://raw.githubusercontent.
copy RouteGuideServer.java src\mair
```

3.5.2 *Running the Hello World Demonstration*

Make sure you open two Command (Terminal) windows, each within the `grpc-java\examples\build\install\examples\bin` folder. In the first of the two windows type the following command:

```
hello-world-server.bat
```

You should see the following:

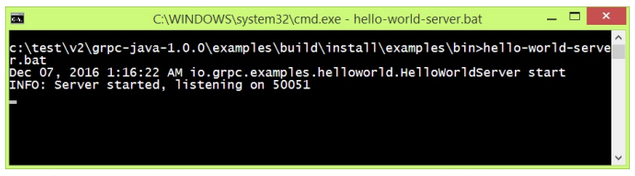

Figure 10: The Hello World gRPC Server.

In the second of the two windows type the following command:

```
hello-world-client.bat
```

You should see the following response:

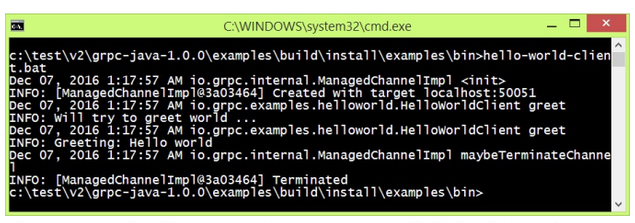

Figure 10: The Hello World gRPC Client and the response from the Server.

4 *Conclusion*

This chapter presented an overview of the concepts behing gRPC, HTTP/2 and will be expanded in both breadth and language implementations. The area of microservices one can see how a server endpoint can actually spawn more endpoints where the message content is the protobuf definition for new endpoints to be generated for load-balancing like for the classical Actor Model.

References

[Apigee]:
https://www.youtube.com/watch?
v=-2sWDr3Z0Wo

[Authentication]:
http://www.grpc.io/docs/guides/auth.html

[Benchmarks]:
http://www.grpc.io/docs/guides/benchmarking.html

[CoreSurfaceAPIs]:
https://github.com/grpc/grpc/tree/master/src/core

[ErrorModel]:
http://www.grpc.io/docs/guides/error.html

[gRPC]:
https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md

[gRPC-Companies]:
http://www.grpc.io/about/

[gRPC-Languages]:
http://www.grpc.io/docs/

[gRPC-Protos]:
https://github.com/googleapis/googleapis/

[Netty]: http://netty.io/

[RFC7540]:
http://httpwg.org/specs/rfc7540.html

[HelloWorldProto]:
https://github.com/grpc/grpc/blob/master/examples/protos/
helloworld.proto

[Protobuf-Types]:
https://developers.google.com/protocol-
buffers/docs/encoding

[gRPC-Overview]:
http://www.grpc.io/docs/guides/

[gRPC-Languages]:
http://www.grpc.io/about/#osp

[gRPC-Benchmark]:
http://www.grpc.io/docs/guides/benchmarking.html