An Introduction to gRPC

Agenda

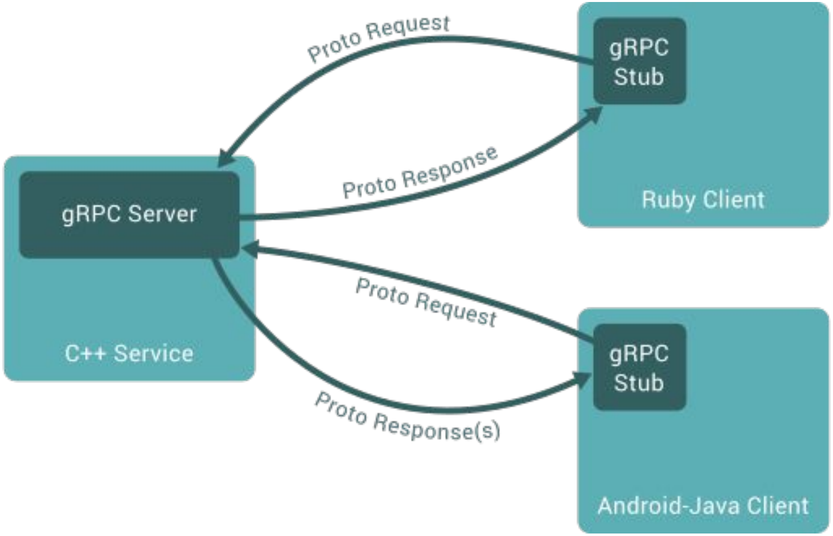
- Defining an service via Protocol Buffers
- Implementing the service
- Invoking the service
- Data Streaming
- Interoperability

References:

- gRPC Java Quickstart
- gRPC Basics: Java

What’s gRPC?

- gRPC is high-performance open-source universal RPC framework
- It leverages HTTP/2 streaming mechanisms
- It supports many languages



Protocol Buffers

- gRPC can use protocol buffers as both its Interface Definition Language (IDL) and as its underlying message interchange format
- Protocol buffers is open source mechanism for serializing structured data based on an efficient binary format to keep payload light and fast

Using gRPC with Maven (dependency)

```
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-netty-shaded</artifactId>
  <version>1.15.1</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-protobuf</artifactId>
  <version>1.15.1</version>
</dependency>
```

```
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-stub</artifactId>
  <version>1.15.1</version>
</dependency>
```

To add in the pom.xml

For the compiler plugin see

https://github.com/grpc/grpc-java

Principal Concepts & Terminology

There are 4 kinds of service (remote) methods

Unary RPCs: they implement the standard synchronous request-reply interaction - clients wait for the completion of the call

```
rpc SayHello(HelloRequest) returns (HelloResponse){}
```

Principal Concepts & Terminology

There are 4 kinds of service (remote) methods

Server streaming RPCs: a client sends a request and gets a stream of messages back: the client reads from the stream until there are no more messages

```
rpc LotsOfReplies(HelloRequest)
        returns (stream HelloResponse){}
```

Principal Concepts & Terminology

There are 4 kinds of service (remote) methods

Client streaming RPCs: the client writes a sequence of messages and sends them to the server using a stream

```
rpc LotsOfGreetings(stream HelloRequest)
               returns (HelloResponse) {}
```

Principal Concepts & Terminology

There are 4 kinds of service (remote) methods

Bidirectional streaming RPCs: where both sides send a sequence of messages using a read-write stream

```
rpc BidiHello(stream HelloRequest)
         returns (stream HelloResponse){}
```

gRPC Workflow

1. Define the interface by using Protocol Buffer language
2. Compile the interface through the protoc to generate the stub for client and server
3. Integrate your code with the stub

The interface

- Define the structure of the interface in a proto file: an ordinary text file with a .proto extension
- Protocol buffer data is structured as messages, a sort of record containing a list of fields (name-value pairs)
- A RPC service is described by listing the signature of methods, i.e., parameters and return types specified as protocol buffer messages

An Example of Interface

```
// 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;
}
```

Invoking the compiler

- We have to use the command tool protoc to generate the required Java code
- We need to use a plugin that saying protoc the code is about to generate will be used by GRPC

```
$ protoc --plugin=protoc-gen-grpc-java=build/exe/java_plugin/protoc-gen-grpc-java
  --grpc-java_out="$OUTPUT_FILE" --proto_path="$DIR_OF_PROTO_FILE"
"$PROTO_FILE"
```

- Maven invokes the compiler to generate the code

The result of the compilation (demo)

From the interface the protoc generates Java classes implementing

- The skeleton for the server and the stub for the client
- Classes to marshaling and unmarshaling messages

Implementing the service

You need to integrate the skeleton with the code

```
public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase{
      @Override
  public void hello(
   HelloRequest request, StreamObserver<HelloResponse> responseObserver) {
     String greeting = new StringBuilder() .append("Hello, ").append(request.getFirstName()).append(" ")
.append(request.getLastName()).toString();

    HelloResponse response = HelloResponse.newBuilder().setGreeting(greeting).build();

    responseObserver.onNext(response);
    responseObserver.onCompleted();
  }
}
```

Comments

- HelloRequest is a class generated by the protoc compiler and contains setter and getter methods to create a request
- HelloResponse is generated by the protoc compiler and contains setter and getter methods to create a response
- StreamObserver<HelloResponse> represents a stream of messages --- in our example we use it to send responses

StreamObserver Interface

- It is used by both the client stubs and service implementations for sending or receiving stream of messages
- For outgoing messages, a StreamObserver is provided by the GRPC to the service (as in the previous example)
- For incoming messages, the service implements the StreamObserver and passes it to the GRPC library for receiving (client streaming)

Implementing The Server

```
public class MyServer
{
   public static void main( String[] args ) throws Exception
   {
        Server server = ServerBuilder
                .forPort(8080)
                .addService((BindableService) new HelloServiceImpl()).build();
         server.start();
         server.awaitTermination();


   }
}
```

ServerBuilder

- ServerBuilder<T> forPort(int port): factory for creating a new ServerBuilder using port
- T addService(BindableService bindableService): set the service to handle incoming requests
- Server build(): creates a new server with the given configuration

Server

- Server start(): bind and start the server
- List<ServerServiceDefinition> getServices(): returns the services registered on the server
- Server shutdown(): shutdown the server (preexisting calls continue until termination but new calls are rejected)
- void awaitTermination(): waits for until the server terminates

Implementing the client

```
public class MyClient {

  ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost",
8080).usePlaintext(true).build();

    HelloServiceGrpc.HelloServiceBlockingStub stub  =
HelloServiceGrpc.newBlockingStub(channel);

  HelloResponse helloResponse = stub.hello(HelloRequest.newBuilder()
                          .setFirstName("Baeldung") .setLastName("gRPC") .build());

              System.out.println(helloResponse.getGreeting());     channel.shutdown();
          }
}
```

Channels

- A Channel is a virtual connection to perform RPC
- It may have zero or many actual connections to the endpoint based on its configuration
- ManagedChannel provides life-cycle management
- ManagedChannelBuilder allows construncting a ManagedChannel

How To Build a Channel

- ManagedChannelBuilder<?> forAddress(String name, int port): set the channel to interact with the endpoint denoted by address and port
- ManagedChannelBuilder<?> forTarget(String uri): set the channel to interact with the endpoint denoted by the uri

dns:///foo.googleapis.com:8080

foo.googleapis.com:8080

- ManagedChannel build(): creates a channel
- T usePlaintext(boolean skipNegotiation): use of a plaintext connection to the server

An Example Of Server Streaming

- We want to implement a service that works as a database to store clinical records
- The client uses the service to retrieve all the test results of a given patient

The Service Interface (.proto)

```
syntax = "proto3";
option java_multiple_files = true;
package clinicalrecords;

message PatientRecord {
        int32 recordId = 1;
        string patient_name = 2;
        string test_result = 3;
}
```

```
message PatientName {
        string patient_name = 1;
}

service DBRecord {
   rpc getRecords(PatientName) returns
(stream PatientRecord){}
}
```

The Remote Service

```
public class DBRecordService extends DBRecordImplBase{
   private List<PatientRecord> records;

   public void getRecords(PatientName request, StreamObserver<PatientRecord> responseObserver) {
              Iterator<PatientRecord> record = records.iterator();

               while(record.hasNext()) {
                     PatientRecord r = record.next();
                     if(r.getPatientName().equals(request.getPatientName()))
                              responseObserver.onNext(r);
               }

               responseObserver.onCompleted();
          }
```

The Server

```
public static void main(String[] args) throws Exception{
        Server srv = ServerBuilder.forPort(8080)
                                .addService(new DBRecordService())
                                .build();
        srv.start();
        srv.awaitTermination();
}
```

The Client

```
public class DBClient {
  public static void main(String [] args) {
      ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost",
8080).usePlaintext().build();
      DBRecordBlockingStub stub = DBRecordGrpc.newBlockingStub(channel);
      PatientName patient = PatientName.newBuilder().setPatientName(args[0]).build();

      Iterator<PatientRecord> stream = stub.getRecords(patient);
      while(stream.hasNext()) {
            PatientRecord record = stream.next();
            System.out.printf("Name: %s --- Test: %s\n", record.getPatientName(),
record.getTestResult());
      }
```

Languange Interoperability: A Python Client

```
with grpc.insecure_channel('localhost:8080') as channel:
    stub = patient_records_pb2_grpc.DBRecordStub(channel)
    responses = stub.getRecords(
      patient_records_pb2.PatientName(patient_name='Minni'))
    for response in responses:
      print("Patient name: %s, Test Result: %s\n" %
          (response.patient_name, response.test_result))
```

The compiler invocation:

```
python -m grpc_tools.protoc -I../proto --python_out=. --grpc_python_out=.
../proto/patient-records.proto
```

Conclusion

- Defining an service via Protocol Buffers
- Implementing the service
- Invoking the service
- Data Streaming
- Interoperability

References:

- gRPC Java Quickstart
- gRPC Basics: Java