gRPC 소개

출처: https://medium.com/@goinhacker/microservices-with-grpc-d504133d191d

1. gRPC 란?

gRPC의 역사는 구글의 데이터센터에서 실행되는 수많은 마이크로서비스들이 10년이상 사용한 단일 범용 RPC 인프라인 Stubby에서 시작됩니다. 구글의 내부시스템은 오래전부터 마이크로서비스 아키텍처를 받아들여 연구해왔고, 내부 서비스들을 연결하기 위해서 Stubby를 만들었습니다. 2015년 3월에 Stubby의 다음 버전을 계획하면서 소스를 오픈하기로 결정하였고, 구글의 내외부 서비스뿐만 아니라 모바일, IOT 등 다양한 엔드포인트(endpoint)에서도 활용하기 위해서 gRPC를 만들었습니다. gRPC는 높은 성능의 오픈소스 범용 RPC 프레임워크입니다.

2. gRPC 를 사용해야 하는 이유

2.1 높은 생산성과 효율적인 유지보수

gRPC는 서비스와 메시지를 정의하기 위해서 Protocol Buffers(이하 ProtoBuf)를 사용합니다. 아래와 같은 ProtoBuf의 IDL만 정의하면 높은 성능을 보장하는 서비스와 메세지에 대한 소스코드가 자동으로 생성됩니다.

```
// 헤더 부분 생략..
// Greeter 서비스 정의
service Greeter {
  // 서비스의 RPC 정의
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
// 요청 메시지 정의
message HelloRequest {
  string name = 1;
}
// 응답 메시지 정의
message HelloReply {
  string message = 1;
}
```

자세한 예제는 뒷 부분에서 다시 다루겠지만, 위와 같은 서비스 정의만으로 서버와 클라이언트(Stub) 코드가 자동으로 생성됩니다. 위 예제의 경우는 sayHello()라는 서비스의 비지니스 로직과 클라이언트에서 Stub을 사용하여 호출해주는 부분만 구현해주면 됩니다.

2.2 다양한 언어와 플랫폼 지원

ProtoBuf의 IDL을 활용한 서비스 정의 한개로 다양한 언어와 플랫폼에서 동작하는 서버와 클라이언트 코드가 생성됩니다. 공식적으로 지원하는 언어 및 플랫폼은 Officially Supported Platforms를 참고하시면 됩니다.

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+ Linux Mac	.NET Core, .NET 4.5+ .NET Core, Mono 4+ .NET Core, Mono 4+
Dart *	Windows/Linux/Mac	Dart 1.24.3+
Go	Windows/Linux/Mac	Go 1.6+
Java	Windows/Linux/Mac	JDK 8 recommended. Gingerbread+ for Android
Node.js	Windows/Linux/Mac	Node v4+
PHP *	Linux/Mac	PHP 5.5+ and PHP 7.0+
Python	Windows/Linux/Mac	Python 2.7 and Python 3.4+
Ruby	Windows/Linux/Mac	

2.3 HTTP/2 기반의 양방향 스트리밍

gRPC는 HTTP/2를 기반으로 통신합니다. HTTP/2의 특징과 장점에 대해서는 뒷 부분에서 자세히 다루도록 하겠습니다. HTTP/2에서는 양방향 스트리밍이 가능합니다. 즉, 일반적인 요청/응답 방식이 아니고 서버와 클라이언트가 서로 동시에 데이터를 스트리밍으로 주고 받을 수 있다는 것 입니다. 이 부분도 뒤에서 예제를 통해서 자세히 다루도록 하겠습니다.

2.4 높은 메시지 압축률과 성능

HTTP/2의 또다른 장점중에 하나는 HTTP를 사용하는 전송보다 높은 헤더 압축률을 보장한다는 점입니다. gRPC에서는 HTTP/2에 의한 압축뿐만 아니라 protoBuf에 의한 메시지 정의에 의해서 메시지 크기를 획기적으로 줄일 수 있습니다. 메시지의 크기가 줄어드는 것은 곧 네트워크 트래픽이 줄어드는 의미하기 때문에 시스템 리소스를 절약하고 성능을 높일 수 있습니다.

2.5 다양한 gRPC 생태계

gRPC는 필요에 따라 활용할 수 있는 Authentication, Tracing, Load Balancing, Health Checking, API Gateway 등의 생태계를 가지고 있습니다. 벡엔드 서버를 개발 하다보면 필요한 다양한 추가 기능이나 도구를 꽤 많이 갖추고 있어서 새로운 기술 스택을 추가로 리서치해서 사용할 일이 많지 않습니다.

3. 누가 gRPC 를 사용 하는가?

gRPC는 기본적으로 구글에서 사용하고 있습니다. 구글의 인프라 내부에서 돌아가는 수많은 서비스들이 gRPC 또는 gRPC의 전신인 Stubby를 사용하여 서비스되고 있습니다. 또한 Square, Netflix, CoreOS 등 회사의 다양한 언어와 플랫폼에서 개발된 수백개의 서비스들로 구성된 클라우드 환경에서 gRPC가 활용되고 있습니다. 즉, 다양한 회사의 마이크로서비스 아키텍쳐에 채택되고 있습니다.

4. 어디에 gRPC 를 사용 하는가?

gRPC는 단일 인스턴스로 돌아가는 CRUD 웹 애플리케이션에서부터 수백개의 서비스 인스턴스가 상호작용하는 마이크로서비스 아키텍쳐(이하 MSA)까지 거의 모든 서버 시스템 개발에 적합합니다. 마이크로서비스에 대해서는 아래 포스팅에서 자세히 다룬바 있으니 참고하시기 바랍니다.

gRPC가 MSA에 적합한 이유는 아래와 같습니다.

gRPC를 활용하면 비니지스 로직에 집중하여 빠른 서비스 개발이 가능하고, 간단한 설치와 빠른 배포가 가능합니다. 또한, 다양한 언어 및 플랫폼 지원으로 폴리글랏 언어와 기술스택을 지향하는 MSA의 철학과도 일맥상통합니다.

ProtoBuf가 지원하는 IDL 활용한 서비스 및 메시지 정의는 MSA의 다양한 기술 스택의 공존으로인한 중복 발생의 단점을 보완하고, 수많은 서비스간의 API 호출로 인한 성능 저하를 개선합니다.

MSA를 기반으로 하면서 분산처리를 위해서 필요한 Security, API Gateway, Tracing, Monitoring, Health Checking 등의 기능들을 Pluggable하게 포함하고 있습니다.

ProtoBuf에 의한 높은 메시지 압축률은 시스템 전체의 네트워크 트래픽을 획기적으로 줄여줍니다. 이것은 동일한 자원 제약에서 더 많은 서비스 인스턴스를 띄울 수 있다는 것을 의미합니다.

5. gRPC 를 사용 하지 말아야 할 곳

gRPC는 RPC로 통신하기 때문에 간단한 REST API를 제공하는 서비스 개발에는 적합하지 않습니다. 하지만 MSA에서 외부에 REST API를 제공할때는 grpc-gateway를 활용할 수 있습니다.

gRPC의 튜토리얼이나 가이드는 상당히 쉽고 빠르게 습득이 가능합니다. 하지만 실제로 제품에 적용하고 운영할때는 ProtoBuf, HTTP2 등 이해해야 할 기반 기술들이 많습니다. 따라서 어느 정도의 Learning Curve는 각오해야 합니다.

6. 어떻게 gRPC 를 사용 하는가?

gRPC를 사용하는 것은 간단하지만, 제대로 사용하기 위해서는 몇가지 기반 기술 및 도구에 대한 이해가 필요합니다. 여기서는 간단한 예제를 포함해서 고려해야할 요소 및 기술들에 대한 설명을 포함합니다.

6.1 RPC System

RPC(Remote Procedure Call)는 원격 컴퓨터나 프로세스에 존재하는 함수를 호출하는데 사용하는 프로토콜 이름입니다.

물론 RPC라는 개념이 존재하기 전부터 소켓 프로그래밍을 통해서 네크워크 상에 존재하는 서비스를 호출할 수는 있었습니다. 그러나 네크워크 상의 원격 통신은 느려지거나 서버가 응답하지 않는 등의 다양한 장애가 발생할 수 있습니다. 소켓 프로그래밍으로 직접 구현한다면 네크워크에서 발생 가능한 다양한 예외상황들을 개발자가 직접 핸들링해야만 합니다.

RPC는 이러한 네크워크 통신과 관련된 작업들을 대신해 줍니다. 개발자는 원격 컴퓨터나 프로세스에 존재하는 함수를 동일 프로세스에 존재하는 함수를 호출하는 것 처럼 호출할 수 있습니다.

6.2 Protocol Buffers

gRPC는 ProtoBuf를 IDL 및 기본 메시지 교환 포멧으로 사용할 수 있습니다(JSON 등 다른 직렬화 방식을 사용할 수도 있음). ProtoBuf는 구글에서 만들고 사용하는 데이터 직렬화 라이브러리 입니다. ProtoBuf 3.0부터 gRPC를 지원하며 IDL로 메시지뿐만 아니라 서비스도 정의가 가능합니다.

```
// Person.proto
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

IDL로 정의된 메시지를 기반으로 데이터 접근을 위한 코드를 생성하기 위해서 protoc 컴파일러를 사용합니다. 간단한 컴파일러 명령어로 원하는 언어의 코드를 생성할 수 있고, Person 클래스(자바의 경우)에 setter/getter 등 필요한 모든 메서드가 만들어 집니다.

```
syntax = "proto3";   // ProtoBuf 버전 3부터 gRPC 서비스 정의 가능.

option java_multiple_files = true;                    // 여러개의 자바 소스코드로
생성
option java_package = "io.grpc.examples.helloworld";   // 생성되는 코드의 자바 패키
지명 설정
option java_outer_classname = "HelloWorldProto";       // 생성되는 클래스명 설정
option objc_class_prefix = "HLW";                      // Object-C에서 생성되는 클
래스명의 접두어 설정

package helloworld;   // ProtoBuf로 정의된 메시지간 이름 충돌을 방지하기 위한 패키지

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
```

```
  string message = 1;
}
```

위 예제는 간단한 gRPC helloworld 예제의 풀 버전입니다. proto 파일을 작성하기 위해서 여러가지 ProtoBuf 옵션 및 IDL에 대한 공부가 필요한 것을 알 수 있습니다. 자세한 내용은 Protocol Buffers 공식 사이트에서 확인할 수 있습니다. gRPC와 함께 사용하기 위한 문법은 proto3 가이드를 참고해야 합니다.

6.3 HTTP/2

gRPC에서 사용하는 통신 프로토콜인 HTTP/2에 대해서 살펴보겠습니다.

HTTP/1은 기본적으로 클라이언트가 서버에 요청을 보내고, 서버가 요청에 대한 응답을 보내는 구조입니다. 따라서 요청 단위로 클라이언트와 서버를 왕복해야 합니다. 또한 쿠키를 포함한 헤더 크기는 불필요하게 큽니다. 이런 특징때문에 HTTP/1은 느립니다. 성능을 개선하기 위해서 구글은 SPDY를 개발하고, 이를 기반으로 HTTP/2 표준이 만들어집니다.

성능이 개선된 HTTP/2의 주요 특징은 아래와 같습니다.

6.3.1 Header Compression

Header Table과 Huffman Encoding 기법을 사용하여 HTTP/2 헤더정보를 압축하였습니다.

6.3.2 Multiplexed Streams

HTTP/1에서 요청마다 새로운 커넥션을 자주 만드는 것과는 달리 HTTP/2는 한개의 커넥션으로 동시에 여러개의 메시지를 주고 받을 수 있습니다.

6.3.3 Server Pus

HTTP/2에서는 클라이언트의 요청없이도 서버가 리소스를 보낼 수 있습니다. 클라이언트 요청이 최소화되기 때문에 성능이 향상될 수 있습니다.

6.3.4 Stream Priority

요청에 우선순위를 지정하여 중요한 리소스를 먼저 전달받을 수 있습니다.

gRPC는 이런 HTTP/2의 특징을 기반으로 하기때문에 양방향 스트리밍이 가능하고, 기본적인 통신 속도가 빠릅니다. on-connection 상태에서 비동기 통신의 구현이 용이합니다.

7. gRPC Example

간단한 gRPC 자바 예제를 통해서 개발자의 작업 플로우를 살펴보겠습니다. (원본 소스코드)

7.1 .proto 파일에 서비스를 정의합니다.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.routeguide";
option java_outer_classname = "RouteGuideProto";
option objc_class_prefix = "RTG";

package routeguide;

// RouteGuide 서비스 정의합니다.
// 서비스가 제공하는 RPC 메서드를 선언하고, 각 메서드의 요청/응답 메시지를 정의합니다.
service RouteGuide {

  // 단순한 RPC
  // 클라이언트에서 요청를 보내고 서버의 응답을 리턴합니다
  rpc GetFeature(Point) returns (Feature) {}

  // 서버에서 클라이언트로 스트리밍하는 RPC
  // 클라이언트에서 요청을 보내고 서버로 부터 더이상 받을 메시지가 없을때까지
  // 스트림(sequence of messages)을 읽습니다
  // 스트림을 사용하기 위해서 stream 키워드를 사용합니다
  rpc ListFeatures(Rectangle) returns (stream Feature) {}

  // 클라이언트에서 서버로 스트리밍하는 RPC
  // 클라이언트에서 스트림을 모두 서버에 쓰고 끝나면 서버의 응답을 리턴합니다
  rpc RecordRoute(stream Point) returns (RouteSummary) {}

  // 양방향 스트리밍 RPC
  // 클라이언트와 서버가 서로 독립적으로 스트림을 읽고 씁니다.
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}

// 서비스의 요청/응답 메시지와 타입 정의를 작성합니다.
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}

message Rectangle {
  Point lo = 1;
  Point hi = 2;
}

message Feature {
  string name = 1;
  Point location = 2;
}

message FeatureDatabase {
  repeated Feature feature = 1;   // repeated를 사용하면 자바에서 List<Feature>로 생
성
}

message RouteNote {
  Point location = 1;
```

```
  string message = 2;
}

message RouteSummary {
  int32 point_count = 1;  // <-- 추후 메시지 변화에 따른 하위 호환성을 위해서 숫자를 지정해야함.
  int32 feature_count = 2;
  int32 distance = 3;
  int32 elapsed_time = 4;
}
```

gRPC는 일반적인 RPC뿐만 아니라 server-to-client streaming RPC, client-to-server streaming RPC, bidirectional streaming RPC를 만들어서 사용할 수 있습니다.

7.2 서버와 클라이언트 코드를 생성합니다.

코드를 생성하기 위해서는 ProtoBuf의 컴파일러인 protoc를 사용합니다. gradle이나 maven을 사용하고 있다면 protoc 빌드 플러그인을 사용하여 코드를 생성할 수 있습니다. 관련 내용은 여기에 자세하게 설명되어 있습니다.

ProtoBuf 메시지 정의에 따라서 Feature.java, Point.java, Rectangle.java 등의 메시지 파일이 생성되고 각 필드에 대한 getter, setter, 직렬화 코드 등이 생성됩니다.

gRPC 서비스 정의에 따라서 RouteGuideGrpc.java 파일이 생성되고, RouteGuideGrpc.RouteGuideImplBase를 상속한 서비스 코드에 비지니스 코드를 작성하면 됩니다.

RouteGuide 서비스의 메서드를 호출하기 위한 클라이언트(stub) 코드들이 생성됩니다. 클라이언트에서 이 Stub을 사용하여 서비스의 함수를 호출하면 됩니다.

7.3 서비스를 구현합니다.

```
// 자동 생성된 RouteGuideGrpc.RouteGuideImplBase를 상속하여 서비스를 구현합니다.
class RouteGuideService extends RouteGuideGrpc.RouteGuideImplBase {

    /**
     * proto에 정의된 getFeature() 함수입니다. (Simple RPC)
     * Point 메시지는 proto에 정의된 타입으로 자동 생성되어 입력으로 들어옵니다.
     * StreamObserver<Feature>는 응답(Feature)에 대한 옵져버 인터페이스 입니다.
     */
    @Override
    public void getFeature(Point request, StreamObserver<Feature> responseObserver) {
        // 자동 생성된 Feature은 기본적으로 빌더 패턴으로 객체를 만들 수 있도록 코드가 생성됩니다.
        Feature feature = Feature.newBuilder().setName("").setLocation(request).build();
        // 서버에서 onNext()를 호출하여 Feature 메시지를 클라이언트 보냅니다.
        responseObserver.onNext(feature);
        // 서버에서 onCompleted()를 호출하여 메시지 전송이 끝났음을 알립니다.
```

```
        responseObserver.onCompleted();
    }

    /**
     * server-to-client streaming RPC
     * 함수의 프로토타입은 Simple RPC와 차이가 없습니다.
     * 하지만, 서버에서 클라이언트로 여러개의 Feature 메시지를 보낼 수 있습니다.
     */
    @Override
    public void listFeatures(
        Rectangle request, StreamObserver<Feature> responseObserver) {

        for (int i = 0; i < 10; i++) {
            Feature feature = Feature.newBuilder()
                .setName(valueOf(i))
                .setLocation(request.getLo())
                .build();
            // 10개의 Feature 메시지 스트림을 클라이언트에 하나씩 전송합니다.
            responseObserver.onNext(feature);
        }

        // 서버에서 onCompleted()를 호출하여 메시지 전송이 끝났음을 알립니다.
        responseObserver.onCompleted();
    }

    /**
     * client-to-server streaming RPC
     * 입력 파라메터로는 클라이언트에 응답을 보내기위한 StreamObserver 인터페이스만 있습니다.
     * 리턴 타입은 클라이언트에서 보내는 메시지 스트림을 핸들링하는 StreamObserver로
     * 함수 구현부에서 바로 인스턴스화 합니다.
     */
    @Override
    public StreamObserver<Point> recordRoute(
        final StreamObserver<RouteSummary> responseObserver) {

        return new StreamObserver<Point>() {    // 클라이언트에서 보내는 스트림을 핸들링합니다.
            int pointCount;

            // 클라이언트에서 onNext(point)를 호출할때마다 실행됩니다.
            @Override
            public void onNext(Point point) {
                // 클라이언트에서 보낸 Point가 들어옵니다.
                logger.log(Level.TRACE, "Point message received {}", point)
                pointCount++;
            }

            // 클라이언트에서 onError()를 호출하거나, 내부 에러가 발생하면 호출됩니다.
            @Override
            public void onError(Throwable t) {
                logger.log(Level.WARNING, "recordRoute cancelled");
            }

            // 클라이언트에서 모든 메시지 스트림 전송을 마치고 onComplete()를 호출하면 실행됩니다.
            // 클라이언트는 onComplete()를 호출한 후, 응답(RouteSummary)가 리턴될때까지 기다립니다.
```

```
            @Override
            public void onCompleted() {
                RouteSummary summary = RouteSummary.newBuilder()
                    .setPointCount(pointCount)
                    .build()
                responseObserver.onNext(summary);   // 서버는 클라이언트에 응답을 보
내고,
                responseObserver.onCompleted();     // onCompleted()를 호출하여 통
신을 종료합니다.
            }
        };
    }

    /**
     * Bidirectional streaming RPC
     * 서버와 클라이언트가 독립적인 채널로 메시지 스트림을 보내고 받습니다.
     * 함수의 프로토타입은 client-to-server streaming RPC와 동일합니다.
     */
    @Override
    public StreamObserver<RouteNote> routeChat(
        final StreamObserver<RouteNote> responseObserver) {

        return new StreamObserver<RouteNote>() {
            // 클라이언트에서 onNext(note)로 메시지를 보낼때마다 호출됩니다.
            @Override
            public void onNext(RouteNote note) {
                for (int i = 0; i < 10; i++) {
                    // 클라이언트의 메시지를 전송중에도 서버에서는 클라이언트의
onNext(note)를 호출하여
                    // 메시지 스트림을 보낼 수 있습니다. 서버와 클라이언트 각각 코드에 작
성된 순서대로
                    // 메시지를 가져오지만 순서에 상관없이 읽고 쓸 수 있습니다.
                    // 즉, 스트림은 완전히 독립적으로 작동합니다.
                    responseObserver.onNext(RouteNote.newBuilder(note).build());
                }
            }

            // 클라이언트에서 onError()를 호출하거나, 내부 에러가 발생하면 호출됩니다.
            @Override
            public void onError(Throwable t) {
                logger.log(Level.WARNING, "routeChat cancelled");
            }

            // 클라이언트에서 onCompleted()를 호출하여 스트리밍을 종료합니다.
            @Override
            public void onCompleted() {
                // 서버에서도 onCompleted()를 호출하여 스트리밍 종료를 알려줍니다.
                responseObserver.onCompleted();
            }
        };
    }
}
```

7.4 서버를 실행 코드를 작성합니다.

서비스의 비니지스 로직은 위에서 작성했지만, 서비스가 실행될 서버 코드가 필요합니다. 서버 코드에서는 포트를 설정하고, 작성된 서비스를 실행하기 위한 부분입니다. gRPC에서는 ServerBuilder를 사용하여 쉽게 서버 코드를 작성할 수 있습니다.

```
public class RouteGuideServer {

    private final int port;
    private final Server server;

    public RouteGuideServer(int port) throws IOException {
        this.port = port;
        // io.grpc.ServerBuilder를 사용하여 코드를 설정하고, 서비스를 추가합니다.
        this.server = ServerBuilder.forPort(port)
            .addService(new RouteGuideService())
            .build();
    }

    public void start() throws IOException {
        server.start();     // 서버를 시작합니다.
        // 서버가 SIGTERM을 받았을때 종료하기 위한 shutdown hook을 추가합니다.
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                RouteGuideServer.this.stop();   // SIGTERM을 받으면 서버를 종료합니
다.
            }
        });
    }

    public void stop() {
        if (server != null) {
            server.shutdown();      // 서버를 종료합니다.
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();  // 서버가 SIGTERM을 받아서 종료될 수 있도록
await 합니다.
        }
    }

    public static void main(String[] args) throws Exception {
        // 8980 포트로 서버를 생성합니다.
        RouteGuideServer server = new RouteGuideServer(8980);
        server.start();                 // 서버를 시작합니다.
        server.blockUntilShutdown();    // 서버가 데몬으로 실행될 수 있도록 블럭합니
다.
    }
}
```

7.5 클라이언트를 구현합니다.

gRPC에서는 클라이언트(Stub) 코드도 생성됩니다. 클라이언트에서 생성된 코드를 사용하여 gRPC Stub을 생성하고 서비스의 각 메서드를 호출하는 코드를 구현합니다.

```
public class RouteGuideClinet {

    private static final Logger logger =
        Logger.getLogger(RouteGuideClient.class.getName());

    // gRPC Channel 인터페이스 입니다. 통신 채널의 설정을 관리할 수 있습니다.
    private final ManagedChannel channel;
    // gRPC 클라이언트는 blocking/sync Stub과 non-blocking/async Stub이 있습니다.
    private final RouteGuideBlockingStub blockingStub;
    private final RouteGuideStub asyncStub;

    public RouteGuideClient(String host, int port) {
        // ManagedChannel은 ServiceProvider에 default 등록된 네트워크 프레임워크을 사
용합니다.
        // 대개는 NettyChannelBuilder나 OkHttpChannelBuilder를 사용해서 지정하여 생성
합니다.
        this.channel = ManagedChannelBuilder.forAddress(host, port)
            .usePlaintext(true)
            .build();
        // blocking/sync stub을 생성합니다.
        this.blockingStub = RouteGuideGrpc.newBlockingStub(channel);
        // non-blocking/async stub을 생성합니다.
        this.asyncStub = RouteGuideGrpc.newStub(channel);
    }

    public void shutdown() throws InterruptedException {
        // 클라이언트 사용이 완료된 후, 5초가 지나면 채널을 닫아줍니다.
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
    }

    /**
     * Simple RPC
     */
    public void getFeature(int lat, int lon) {
        // 서버에 보낼 요청 메시지를 생성합니다.
        Point request =
Point.newBuilder().setLatitude(lat).setLongitude(lon).build();

        Feature feature;
        try {
            // blocking으로 서버의 getFeature() 메서드를 호출합니다.
            feature = blockingStub.getFeature(request);
        } catch (StatusRuntimeException e) {
            // gRPC 통신에 문제가 생기면 StatusRuntimeException이 발생합니다.
            return;
        }
    }

    /**
     * server-to-client streaming RPC
     */
    public void listFeatures(int lowLat, int lowLon, int hiLat, int hiLon) {
```

```
        // 서버에 보낸 요청 메시지를 생성합니다.
        Rectangle request = Rectangle.newBuilder()

.setLo(Point.newBuilder().setLatitude(lowLat).setLongitude(lowLon).build())

.setHi(Point.newBuilder().setLatitude(hiLat).setLongitude(hiLon).build())
            .build();

        Iterator<Feature> features;
        try {
            // blocking으로 서버의 listFeatures() 메서드를 호출하고 메시지 스트림을 받
아서 리턴합니다.
            features = blockingStub.listFeatures(request);
        } catch (StatusRuntimeException e) {
            return;
        }
    }

    /**
     * client-to-server streaming RPC
     */
    public void recordRoute(
        List<Feature> features, int numPoints) throws InterruptedException {

        // 비동기 처리를 위해서 CountDownLatch를 사용하였습니다.
        final CountDownLatch finishLatch = new CountDownLatch(1);

        // 서버에서 보내는 메시지를 처리하기 위한 옵져버를 생성합니다.
        StreamObserver<RouteSummary> responseObserver =
            new StreamObserver<RouteSummary>() {

            @Override
            public void onNext(RouteSummary summary) {
                // 여기서는 양방향 스트림이 아니기때문에 정상적인 경우, onNext()가 한번씩
만 호출됩니다.
                logger.info("A message received from server: {}", summary);
            }

            @Override
            public void onError(Throwable t) {
                // 서버에서 onError()를 호출하거나 내부 통신장애가 발생하면 호출됩니다.
                logger.error("RecordRoute Failed: {}", Status.fromThrowable(t));
                finishLatch.countDown();
            }

            @Override
            public void onCompleted() {
                // 메시지 전송이 완료되면 서버에서 onCompleted()를 호출합니다.
                finishLatch.countDown();
            }
        };

        // non-blocking/async stub을 사용하여 서버의 recordRoute()를 호출하였습니다.
        // 서버의 응답을 기다리기 위해서 비동기 stub을 사용해야 합니다.
        StreamObserver<Point> requestObserver =
asyncStub.recordRoute(responseObserver);
        try {
            for (int i = 0; i < numPoints; ++i) {
```

```
                Point point = features.get(i).getLocation();
                // 클라이언트에서 서버로 메시지 스트림을 전송합니다.
                requestObserver.onNext(point);

                // 서버에서 종료하여 finishLatch.countDown()가 호출되면 종료합니다.
                if (finishLatch.getCount() == 0) {
                    return;
                }
            }
        } catch (RuntimeException e) {
            // 메시지 전송과정에서 에러가 발생하면 onError()를 호출하여 서버에 에러 상황
을 알립니다.
            requestObserver.onError(e);
            throw e;
        }
        // 메시지 전송이 완료되면 onCompleted()를 호출하여 서버에 종료 상황을 알립니다.
        requestObserver.onCompleted();

        // finishLatch를 사용하여 최대 1분간 blocking 하도록 합니다.
        if (!finishLatch.await(1, TimeUnit.MINUTES)) {
            logger.warn("recordRoute can not finish within 1 minutes");
        }
    }

    public void routeChat() {
        final CountDownLatch finishLatch = new CountDownLatch(1);
        StreamObserver<RouteNote> requestObserver =
            asyncStub.routeChat(new StreamObserver<RouteNote>() {
                @Override
                public void onNext(RouteNote note) {
                    // 서버로부터 오는 메시지 스트림을 처리합니다.
                    logger.info("A message received from server: {}", note);
                }

                @Override
                public void onError(Throwable t) {
                    // 서버에서 에러가 발생하면 호출합니다.
                    logger.info("RouteChat Failed: {}",
Status.fromThrowable(t));
                    finishLatch.countDown();
                }

                @Override
                public void onCompleted() {
                    // 서버에서 메시지 전송이 완료되면 호출합니다.
                    logger.info("Finished RouteChat");
                    finishLatch.countDown();
                }
            });

        try {
            RouteNote[] requests = {
                RouteNote.newBuilder().setMessage("First").build(),
                RouteNote.newBuilder().setMessage("Second").build()
            }

            for (RouteNote request : requests) {
                logger.info("Sending message: {}", request);
```

```
                // 클라언트에서 서버로 메시지 스트림을 전송합니다.
                requestObserver.onNext(request);
            }
        } catch (RuntimeException e) {
            // 에러가 발생하면 서버에 에러 상황을 알립니다.
            requestObserver.onError(e);
            throw e;
        }
        // 메시지 전송이 끝나면 서버에 종료 상황을 알립니다.

        requestObserver.onCompleted();

        // finishLatch를 사용하여 최대 10분간 blocking 하도록 합니다.
        if (!finishLatch.await(10, TimeUnit.MINUTES)) {
            logger.warn("routeChat can not finish within 10 minutes");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // 클라이언트를 생성합니다. 이때 서버의 주소와 포트번호를 입력합니다.
        RouteGuideClient client = new RouteGuideClient("localhost", 8980);
        try {
            // 클라이언트의 메서드를 호출하여 서버의 메서드들을 테스트합니다.
            client.getFeature(409146138, -746188906);
            client.listFeatures(400000000, -750000000, 420000000, -730000000);
            client.recordRoute(features, 10);
            client.routeChat();
        } finally {
            // 클라이언트 사용이 완료된 후, 채널을 종료합니다.
            client.shutdown();
        }
    }
}
```