

Labo Distributed Systems 3: google Remote Procedure Call (gRPC)

Dit labo bestaat uit 3 delen. In deel 1 wordt de IDEA ontwikkelomgeving geïnstalleerd en het project geïmporteerd vanaf een git repository. In het 2de deel wordt er een kort voorbeeld gegeven van gRPC waarbij er vertrokken wordt van een bestaand project waarbij uitleg gegeven wordt over bepaalde aspecten van de technologie. Deel 3 bestaat uit een opdracht waarin gRPC gebruikt moet worden.

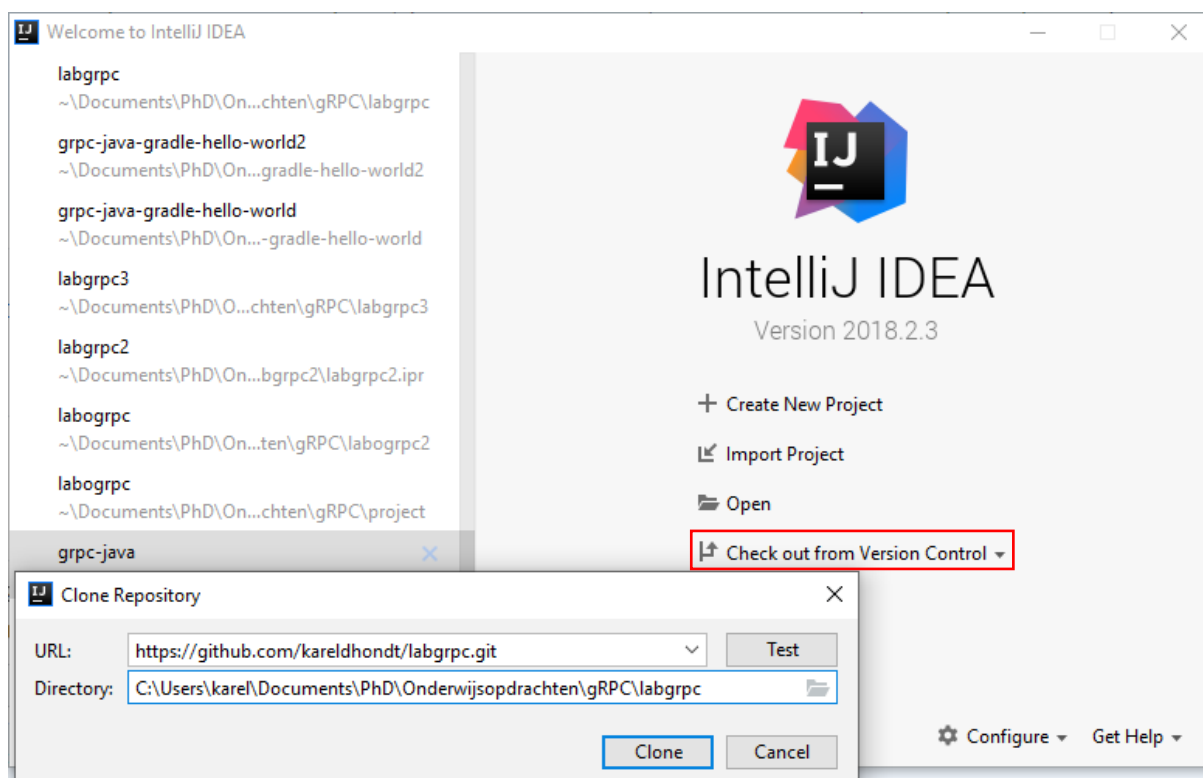
DEEL 1: IntelliJ IDEA klaarmaken

Voor dit labo over gRPC maken we gebruik van de IntelliJ IDEA community ontwikkelomgeving. De installer kan worden gevonden op de volgende site: <https://www.jetbrains.com/idea/download/>

Optioneel kan de [Protobuf Support plugin](#) worden geïnstalleerd voor code completion en syntax highlighting in de IDE.

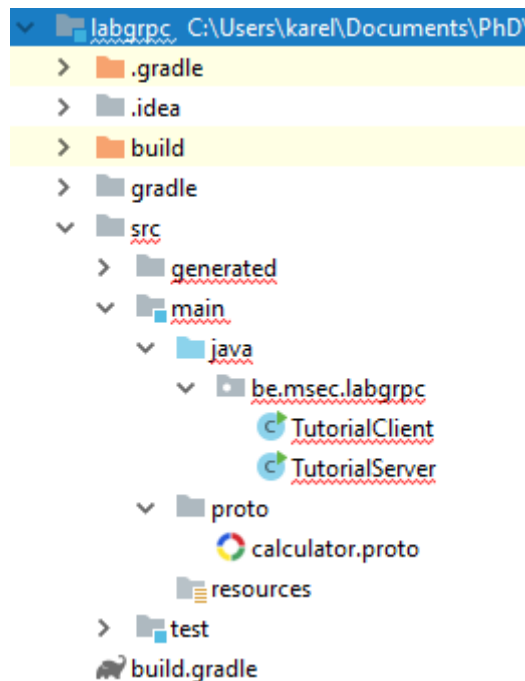
Vervolgens importeren we het project vanaf volgende git repository:

<https://github.com/kareldhondt/labgrpc.git>



DEEL 2: Voorbeeld van gRPC

Onderstaand voorbeeld demonstreert een aantal mogelijkheden van het gRPC framework. Het voorbeeld bestaat uit een client en een server. De server voorziet 3 methodes voor de client: *calculateSum*, *streamingSum* en *calculatorHistory*. De client zal met behulp van gRPC een verbinding opzetten met de server en deze methodes aanspreken.



2.1 gRPC start

Dit project maakt gebruik van het Gradle build systeem. We beginnen met de inhoud van het project te verkennen. Het eerste bestand dat we bekijken is de *gradle.build*. Dit bestand bevat de configuratie die ervoor zorgt dat gRPC gebruikt kan worden door Gradle. We zien de volgende dependencies in het bestand.

```
dependencies {  
    testCompile group: 'junit', name: 'junit', version: '4.12'  
    compile 'io.grpc:grpc-all:1.12.0'  
    compile 'javax.annotation:javax.annotation-api:1.2'  
}
```

Deze dependencies laden de gRPC code in voor het Gradle project. Als we *gradle.build* openen zien we dat er naast de dependencies ook nog een protobuf object in staat. Dit object zorgt voor nieuwe build regels die code genereert op basis van het *calculator.proto* bestand om communicatie over gRPC mogelijk te maken.

Dit is dan ook het volgende bestand dat we bekijken. Het *calculator.proto* bestand begint met een aantal configuratie settings om aan te geven dat we java code willen genereren op basis van deze file. Onze service is gedefinieerd in de .proto file.

```
service Calculator{  
    ...  
}
```

In deze service vinden we de functies terug die de server zal aanbieden.

- De functie *calculateSum* is een eenvoudige RPC functie. De server geeft het antwoord op de som die doorgegeven wordt. Er wordt een request opgestuurd en de server antwoordt met een response.

```
// Bereken een som  
rpc calculateSum(Sum) returns (CalculatorReply) {}
```

- De functie *calculatorHistory* is een server-side streaming functie. Deze functie krijgt een leeg request en stuurt alle uitgevoerde berekeningen als output. Aangezien we niet weten hoeveel berekeningen er aan deze functie vooraf zijn gegaan wordt er een stream teruggestuurd. De client blijft deze stream uitlezen tot er geen berichten meer zijn. Dit soort functie wordt gespecificeerd door het woord **stream** voor het response type te plaatsen.

```
// Vraag de uitgevoerde berekeningen op  
rpc calculatorHistory(Empty) returns (stream Calculation) {}
```

- *StreamingSum* is een client-side streaming functie. Hier stuurt de client een onbepaald aantal getallen op die dan opgeteld worden door de server. De client stuurt een opeenvolging van berichten naar de server met behulp van een stream. Eenmaal de client alle berichten verzonden heeft, wacht hij op de server om alles te lezen en te antwoorden. Dit soort methode wordt gespecificeerd door **stream** voor het request type te plaatsen.

```
// Doe een som met onbepaalde input  
rpc streamingSum(stream Sum) returns (CalculatorReply) {}
```

De .proto file bevat ook de definities van de berichten types voor alle request en response types die onze methodes gebruiken. Zo is er het bijvoorbeeld het **Sum** type

```
// De aanvraag voor een som  
message Sum{  
    int32 a = 1;  
    int32 b = 2;  
}
```

Op basis van deze definities zal gRPC client en server interfaces maken. Gradle zorgt ervoor dat deze code genereerd als deel van het bouw proces.

De volgende klassen worden gegenereerd op basis van de service definitie.

- Sum.java, CalculatorReply.java, Calculation.java en Empty.java. Deze klassen bevatten alle code om onze request en response bericht types in te vullen en gebruiken.
- CalculatorGrpc.CalculatorImplBase. Deze bevat:
 - Een basis klasse voor Calculator servers om te implementeren, CalculatorGrpc.CalculatorImplBase, met alle methodes gedefinieerd in de Calculator service.
 - Stub klassen die clients gebruiken om een Calculator server te contacteren.

2.2 Server implementatie

De server implementeren bestaat uit 2 delen. Ten eerste moet de gegenereerde klasse overschreven worden om een implementatie te voorzien voor de functies die de service aanbiedt. Ten tweede moet de server zelf draaien en luisteren naar requests van clients.

In TutorialServer.java zien we een CalculatorService klasse die de gegenereerde CalculatorGRPC klasse extend.

```
private static class CalculatorService
    extends CalculatorGrpc.CalculatorImplBase {
    ...
}
```

In deze CalculatorService vinden we de implementaties van de functies gedefinieerd in tutorial.proto terug. De eenvoudigste functie *calculateSum* telt de meegegeven getallen op en stuurt het antwoord terug.

```
@Override
public void calculateSum(Sum request, StreamObserver<CalculatorReply>
    responseObserver) {
    responseObserver.onNext(sum(request));
    responseObserver.onCompleted();
}

...

private CalculatorReply sum(Sum request) {
    int solution = request.getA() + request.getB();
    CalculatorReply reply =
        CalculatorReply.newBuilder().setSolution(solution).build();
    history.add(Calculation.newBuilder().setSum(request)
        .setSolution(reply).setIndex(history.size()).build());
    return reply;
}
```

Deze functie heeft twee parameters:

- **Sum:** het request
- **StreamObserver<CalculatorReply>:** Een speciale interface die de server moet aanroepen om te antwoorden.

Om de functie correct te doen verlopen doorlopen we de volgende stappen. Eerst creëren we een CalculatorReply object als antwoord zoals gedefinieerd in de proto file. We gebruiken de onNext()

methode om het object terug te sturen naar de client. Ten slotte roepen we de *onCompleted()* methode aan om aan te tonen dat we klaar zijn met de RPC.

De functie *calculatorHistory* is de server-side streaming functie die een onbepaald aantal berekeningen terugstuurt als antwoord.

```
@Override
public void calculatorHistory(Empty nul, StreamObserver<Calculation>
    responseObserver) {
    for(int i = 0; i < history.size(); i++){
        responseObserver.onNext(history.get(i));
    }

    responseObserver.onCompleted();
}
```

Net als bij de eenvoudige RPC, heeft deze methode een request object (in dit geval is het een leeg veld) en een StreamObserver.

Bij deze functie nemen we het aantal Calculation objecten dat we nodig hebben en sturen we elk ervan naar de client met de *onNext()* methode. Uiteindelijk gebruiken we de *onCompleted()* methode opnieuw om aan te geven dat we klaar zijn met antwoorden te versturen.

Ten slotte is er de *streamingSum* methode. Hierbij nemen we een willekeurig aantal getallen en tellen deze allemaal op. De grootte van de input is hier niet bekend en we zullen gebruik maken van client-side streaming om deze correct door te geven. Deze methode is een stuk gecompliceerder dan de vorige.

```
@Override
public StreamObserver<Sum> streamingSum(final
    StreamObserver<CalculatorReply> responseObserver){
    return new StreamObserver<Sum>() {
        int sumCount;
        CalculatorReply tempSolution;

        @Override
        public void onNext(Sum sum) {
            if(sumCount == 0)
                tempSolution = sum(sum);
            else
                tempSolution = sum(sum, tempSolution);
        }

        @Override
        public void onError(Throwable t){
            logger.log(Level.WARNING, "Encountered error in
                StreamingSum", t);
        }

        @Override
        public void onCompleted() {
            responseObserver.onNext(tempSolution);
            responseObserver.onCompleted();
        }
    };
}
```

```
}
```

We zien dat ook deze methode een `StreamObserver` heeft om de antwoorden door te sturen maar deze keer antwoordt de server ook een `StreamObserver` zodat de client zijn requests erin kan schrijven. In de methode zelf instantiëren we een nieuwe `StreamObserver`. Hierin wordt de

- `onNext()` overschreven om de nieuwe getallen op te tellen bij de vorige oplossing elke keer de client een nieuwe `Sum` in de stream schrijft.
- `OnCompleted()` overschreven om de uiteindelijke oplossing naar de client te sturen met de `onNext()` van de `StreamObserver` die de antwoorden beheert en vervolgens de `onCompleted()` methode aan te roepen om de RPC af te werken aan de server kant.

Eenmaal alle methodes geïmplementeerd zijn, moeten we de gRPC server nog opstarten zodat clients de service kunnen gebruiken. Het volgende stuk code geeft aan hoe we onze `Calculator` kunnen starten.

```
public TutorialServer(int port) throws IOException {
    this(ServerBuilder.forPort(port), port);

public TutorialServer(ServerBuilder<?> serverBuilder, int port){
    this.port = port;
    if(history == null)
        history = new ArrayList<Calculation>();
    server = serverBuilder.addService(new CalculatorService()).build();
}

...

public void start() throws IOException{
    server.start();
    logger.info("Server started, listening on " + port);
    ...
}
```

In deze code bouwen en starten we de server met behulp van een `ServerBuilder`. Om dit te bereiken speciëren we een adres en port waarop de server luistert naar client requests met de `forPort()` methode. Daarnaast creëren we een instantie van onze `CalculatorService` klasse en geven deze door aan met de `addService()` methode. Ten slotte roepen we `build()` en `start()` aan om de server te bouwen en op te starten.

2.3 Client implementatie

Om de methodes van de service aan te kunnen roepen hebben we twee stubs nodig.

- Een blocking/synchrone stub: dit houdt in dat de RPC call wacht tot de server antwoord, en zal of een response geven of een exception opwerpen.
- Een non-blocking/asynchrone stub: deze maakt non-blocking calls naar de server. Het antwoord wordt hier asynchroon teruggestuurd. Bepaalde streaming calls kunnen enkel gemaakt worden met een asynchrone stub.

```
public TutorialClient(String host, int port){
    this(ManagedChannelBuilder.forAddress(host,port).usePlaintext(true));
}
```

```

public TutorialClient(ManagedChannelBuilder<?> channelBuilder) {
    channel = channelBuilder.build();
    blockingStub = CalculatorGrpc.newBlockingStub(channel);
    asyncStub = CalculatorGrpc.newStub(channel);
}

```

De `ManagedChannelBuilder` creëert een kanaal. Dit kanaal wordt dan gebruikt om de stubs te maken met de `newStub` en `newBlockingStub` methode uit de gegenereerde `RouteGuideGrpc` klasse.

Nu kijken we hoe we de methodes van de server kunnen aanspreken. We beginnen opnieuw met de simpele `calculateSum`.

```

public void calculateSum(int a, int b){
    info("Calculating sum of {0} and {1}", a, b);

    Sum request = Sum.newBuilder().setA(a).setB(b).build();
    CalculatorReply reply;
    try{
        reply = blockingStub.calculateSum(request);
    } catch (StatusRuntimeException e) {
        logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
        return;
    }

    info("Solution of the {0} + {1} = {2}", a, b, reply.getSolution());
}

```

Er wordt een request object, hier `Sum`, gecreëerd en ingevuld. Dit object wordt dan doorgegeven aan de `calculateSum()` method op onze blocking stub en we krijgen een `CalculatorReply` terug.

De volgende functie is de server-side streaming call `calculatorHistory` die een stream aan `Calculations` teruggeeft.

```

public void calculatorHistory(){
    info("Requesting calculator history");

    Iterator<Calculation> calculations;
    try{
        calculations = blockingStub.calculatorHistory(
            Empty.newBuilder().build());
    } catch (StatusRuntimeException e) {
        logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
        return;
    }

    StringBuilder responseLog = new StringBuilder("History:\n");
    while(calculations.hasNext()){
        Calculation calculation = calculations.next();
        responseLog.append(calculation);
    }

    info(responseLog.toString());
}

```

We zien dat deze functie zeer gelijkaardig is aan de eenvoudige RPC met als voornaamste verschil dat we hier een Iterator als response krijgen die de client kan gebruiken om alle geretournde Calculations uit te lezen.

Ten slotte hebben we de client-side streaming functie *streamingSum*. Deze is opnieuw een stuk gecompliceerder dan de andere functies. We genereren een aantal willekeurige getallen om op te tellen en blijven deze naar de server sturen om uiteindelijk een antwoord te krijgen. Deze methode maakt gebruik van een asynchrone stub.

```
public void streamingSum(int operands) throws InterruptedException {
    info("Streaming sum");
    final CountDownLatch finishLatch = new CountDownLatch(1);
    StreamObserver<CalculatorReply> responseObserver = new
        StreamObserver<CalculatorReply>() {
            @Override
            public void onNext(CalculatorReply reply) {
                info("Finished streaming sum. The solution is {0}",
                    reply.getSolution());
            }

            @Override
            public void onError(Throwable t){
                Status status = Status.fromThrowable(t);
                logger.log(Level.WARNING, "StreamingSum failed:{0}",
                    status);
                finishLatch.countDown();
            }

            @Override
            public void onCompleted(){
                info("Finished StreamingSum");
                finishLatch.countDown();
            }
        };

    StreamObserver<Sum> requestObserver =
        asyncStub.streamingSum(responseObserver);
    try{
        int a;
        int b;
        Sum request;
        for(int i = 0; i < (operands+1)/2; i++){
            a = (int) (Math.random() * 10);
            b = (int) (Math.random() * 10);
            if(i == ((operands+1)/2-1) && (operands % 2) != 0){
                b = 0;
                info("Adding new summand {0}", a);
            } else
                info("Adding new summands {0} and {1} ", a, b);

            request = Sum.newBuilder().setA(a).setB(b).build();

            requestObserver.onNext(request);

            Thread.sleep(1000);
            if (finishLatch.getCount() == 0){
                // RPC completed or errored before we finished sending.
                // Sending further request won't error, but they will
                // just be thrown away.
            }
        }
    }
```



```

        return;
    }

    }
} catch (RuntimeException e){
    requestObserver.onError(e);
    throw e;
}

// Mark the end of requests
requestObserver.onCompleted();

// Receiving happens asynchronously
finishLatch.await(1, TimeUnit.MINUTES);
}

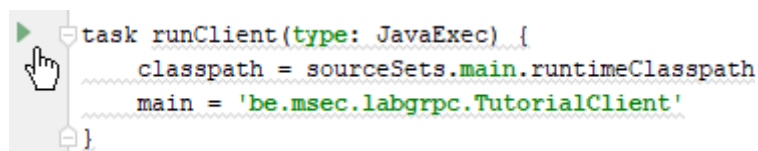
```

In deze methode creëren we een `StreamObserver` die een speciale interface aanbiedt die de server kan aanspreken met zijn antwoord. Deze `StreamObserver` heeft een overschreven `onNext()` methode die het antwoord van de server uitprint en een overschreven `onCompleted()` methode om te kijken of de server klaar is met schrijven.

De `StreamObserver` wordt doorgegeven aan de `streamingSum()` methode van de asynchrone stub en dan krijgen we zelf een `StreamObserver` object terug om onze `Sum` in te schrijven. Eenmaal we alle `Sum` objecten hebben doorgegeven, gebruiken we de `onCompleted` methode om gRPC te vertellen dat de client side klaar is met data te schrijven. De `CountDownLatch` wordt gebruikt om te controleren of de server van zijn kant klaar is.

2.4 De code uitvoeren

Eenmaal alles geïmplementeerd is moet de code nog uitgevoerd worden. Hiervoor openen we opnieuw het `build.gradle` bestand en navigeren we naar de `run tasks`:



```

task runClient(type: JavaExec) {
    classpath = sourceSets.main.runtimeClasspath
    main = 'be.msec.labgrpc.TutorialClient'
}

```

De client en server kunnen vanuit hier worden opgestart met een druk op de groene play knop. De twee onderstaande `tasks` dienen om startscripts te genereren voor de IDEA ontwikkelomgeving om niet steeds naar het `build.gradle` bestand te moeten navigeren alvorens de programma's op te starten.

Extra informatie over gRPC kan gevonden worden op <http://www.grpc.io/>

DEEL 3: Opdracht

Pas je opdracht van vorige week aan om alle communicatie over gRPC te laten verlopen. Denk voor je begint goed na over welke methoden je nodig hebt om dit te realiseren. Uiteraard kan een groot deel van de code van vorig labo hergebruikt worden (vb. GUI).

Stuur de code (volledig project in zip formaat, en voorzien van een *readme*-file met instructies om uit te voeren) op het einde van het labo door naar karel.dhondt@cs.kuleuven.be.

Evaluatie:

Na de eerste drie labosessies wordt een kort verslag (2 pagina's) verwacht over de ervaring met de geziene communicatietechnologieën, de aanpak van de gerealiseerde programma's, en een vergelijkende analyse van de geziene technologieën (Sockets, RMI, gRPC). Dit verslag wordt dan samen met de geproduceerde code beoordeeld.