



gRPC - Streaming

Sistemas Distribuidos

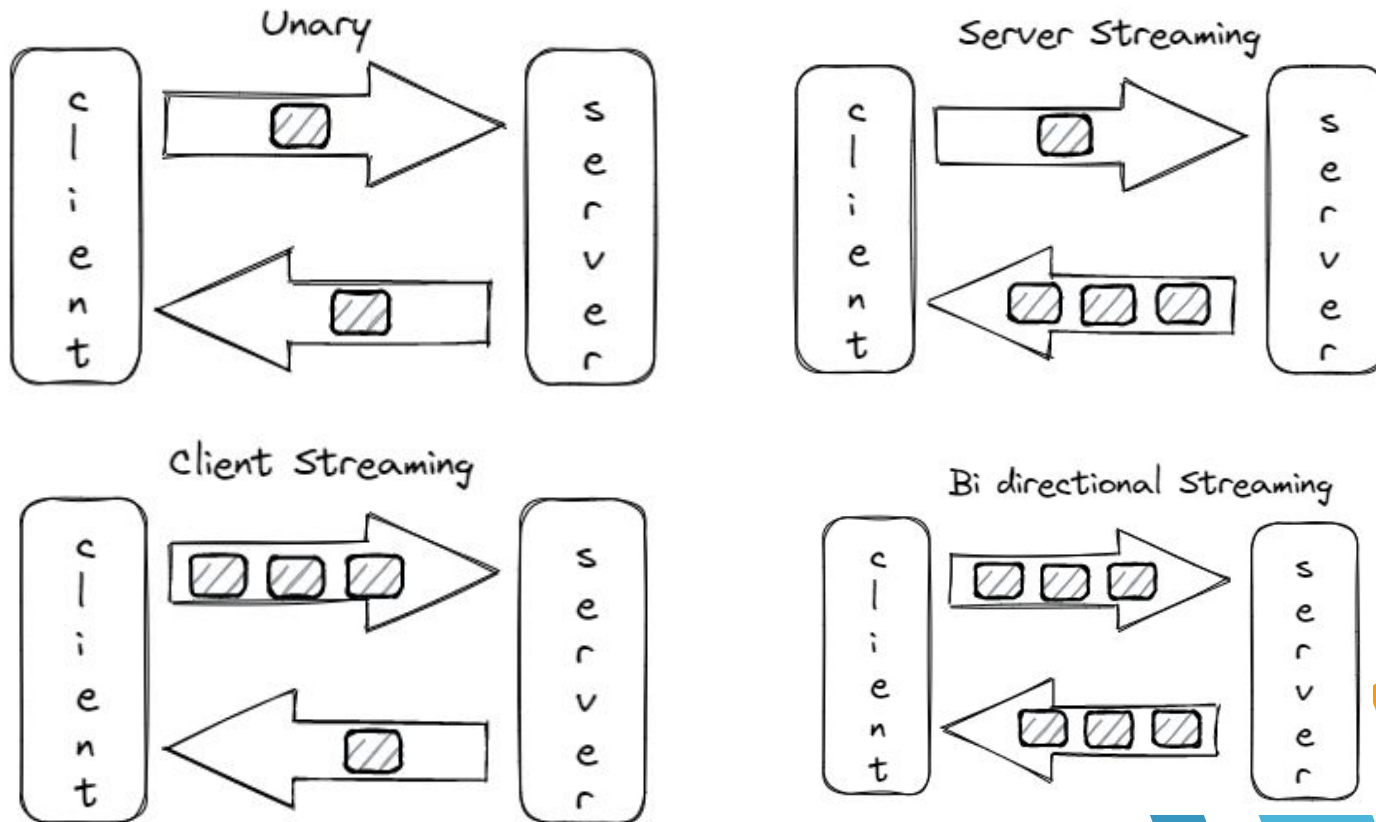


Streaming

gRPC puede mantener una conexión abierta, mandando mas de un solo mensaje, ya que esta basado en HTTP2.

gRPC - Tipos métodos de servicios

gRPC permite definir **4 tipos de métodos de servicios** de acuerdo al patrón de comunicación. Básicamente se toman en cuenta cuántos elementos se mandan del objeto request y se reciben del objeto response.



gRPC - RPC Unario

RPC Unario es cuando el cliente envía un objeto request y el server responde con un objeto.

Todos los ejemplos vistos hasta el momento utilizan este patrón de comunicación.

```
service TrainTicketService{  
    rpc GetDestinations (google.protobuf.Empty) returns (Destinations);  
}  
  
message Destinations {  
    repeated string destinations = 1;  
}
```

Ejercicio 1 - Unary

1. Crear el proyecto grpc-streaming
2. Incorporar el archivo **trainTicketService.proto** al api.
3. Incorporar el archivo **TicketRepository.java** al proyecto server.
4. Implementar **GetDestinations** y un cliente que lo consuma

gRPC - Streaming

gRPC con streaming es una variante de comunicación donde al menos uno de los elementos (Request o Response) envía más de un elemento del tipo definido.

Para indicar esta repetición se utiliza la palabra reservada **stream** en el elemento que será enviado más de una vez.

```
service TrainTicketService{  
    rpc GetTrainsForDestination(google.protobuf.StringValue) returns (stream  
    Train);  
    rpc PurchaseTicket(stream Ticket) returns (Reservation);  
    rpc GetTicketsForReservations(stream Reservation) returns (stream Ticket);  
}
```

gRPC - Streaming vs Repeated

Streaming y repeated son dos maneras de enviar más de un elemento en la comunicación.

Se pueden intercambiar pero no son lo mismo.

- **repeated**: es un modificador para un campo dentro de un mensaje.
- **stream**: es un modificador para un mensaje (de entrada y/o salida).

```
service TrainTicketService{  
    rpc GetDestinations (google.protobuf.Empty) returns (Destinations);  
    rpc GetTrainsForDestination (google.protobuf.StringValue)  
        returns (stream Train);  
}  
  
message Destinations {  
    repeated string destinations = 1;  
}
```


Técnicamente **repeated** se usa para listas que se obtienen inmediatamente y **stream** se usa más cuando se tarda en recuperar los elementos.



gRPC - Streaming

Internamente la posibilidad de establecer una comunicación vía streaming se basa en las características de HTTP/2 de poder establecer una conexión de larga duración y multiplexar los request/responses.

De esta manera cliente y servidor permanecen conectados por un largo período y a medida que se calculan los valores a enviar o devolver se van enviando por el cable.



gRPC - Server Side

En streaming server side el cliente envía un elemento y el servidor retorna más de un elemento.

```
rpc GetTrainsForDestination(google.protobuf.StringValue)  
  returns (stream Train);
```

```
message Train {  
  string id = 1;  
  string destination = 2;  
  string time = 3;  
  int32 availableCount = 4;  
}
```

gRPC - Server Side

El Servidor se implementa similar al unario pero el servidor llama varias veces al `onNext`

```
public void getTrainsForDestination(final StringValue destination,
                                    final StreamObserver<Train> responseObserver)
{
    List<> availability = ...
    availability.forEach(train -> {
        responseObserver.onNext(Train.newBuilder()
            ...
            .build());
    });

    responseObserver.onCompleted();
}
```

gRPC - Server Side

El Cliente recibe un **iterador** de respuestas

```
final StringValue request = StringValue.newBuilder()
    .setValue(destination).build();

final Iterator<Train> trainsForDestination =
    trainTicketServiceBlockingStub.getTrainsForDestination(request);

final List<Train> trains = new ArrayList<>();

while (trainsForDestination.hasNext()) {
    trains.add((trainsForDestination.next()));
}

return trains;
```

En este caso también se puede utilizar el *stub y pasarle un observer.



Ejercicio 2 - Server Side Streaming

1. Implementar el método **GetTrainsForDestination**
- 

gRPC - Client Side

Streaming client side significa que el cliente envía muchos objetos en el request y el servidor retorna un solo valor

```
rpc PurchaseTicket(stream Ticket) returns (Reservation);
```

```
message Ticket {  
    string id = 1;  
    string trainId = 2;  
    string passengerName = 3;  
}
```

```
message Reservation {  
    string id = 1;  
    int32 ticketCount = 2;  
}
```

gRPC - Client Side

El servidor además de recibir el observer para la respuesta retorna un observer del request para ir aplicando las operaciones “a medida” que el cliente envía un elemento.

```
public StreamObserver<Ticket> purchaseTicket(final StreamObserver<Reservation>
responseObserver) {
    return new StreamObserver<Ticket>() {
        final List<Ticket> tickets = new ArrayList<>();

        @Override
        public void onNext(final Ticket ticket) {
            tickets.add(ticket);
        }

        @Override
        public void onCompleted() {
            final String reservationId = ...
            Reservation reservation = Reservation.newBuilder().build();
            responseObserver.onNext(reservation);
            responseObserver.onCompleted();
        }
    };
}
```

Antes, el metodo solo "recibia lo que devuelve" (responseObserver). Ahora, el metodo tambien "devuelve lo que recibe"

gRPC - Client Side

El cliente define el observer para la respuesta y llama al método recibiendo el observer del request que se usa para enviar los elementos

Uso CompletableFuture para reservation para hacer que el metodo sea bloqueante

```
final CompletableFuture<Reservation> reservation = new CompletableFuture<>();
final StreamObserver<Reservation> response = new StreamObserver<Reservation>() {
    @Override
    public void onNext(final Reservation r) {
        reservation.complete(r);
    }
};
```

Aca se implementa el onNext pero no el onComplete.
Se va a llamar al default onComplete del observer,
que solo corta la comunicacion.

Cuando el cliente llama al server, recibe un StreamObserver por el cual puede empezar a mandar parametros. Obvio que aca no se puede usar un blocking stub porque se quedaria en deadlock.

```
final StreamObserver<Ticket> ticketStreamObserver =
    trainTicketServiceStub.purchaseTicket(response);
```

```
names.forEach(name -> {
    final Ticket ticket = Ticket.newBuilder().build();
    ticketStreamObserver.onNext(ticket);
});
```

```
ticketStreamObserver.onCompleted();
return reservation.get();
```

En este caso el único cliente que se puede utilizar es el *stub.



Ejercicio 3 - Client Side Streaming

1. Implementar **purchaseTicket** y un cliente que consuma este servicio.
- 



gRPC - Bidirectional Streaming

Bidirectional streaming se da cuando tanto cliente y servidor envían más de un valor entre ellos

```
rpc GetTicketsForReservations (stream Reservation) returns (stream Ticket);
```

gRPC - Bidirectional Streaming

En el servidor el código es igual que en el streaming desde el cliente, la diferencia es que el servicio puede enviar más de una respuesta por cada elemento en vez de esperar a recibir todos los elementos del request

```
public StreamObserver<Reservation> getTicketsFor(final StreamObserver<Ticket>
responseObserver) {
    return new StreamObserver<Reservation>() {
        @Override
        public void onNext(final Reservation reservation) {
            Optional<List<Ticket>> tickets = ...
            if (tickets.isPresent()) {
                tickets.get().forEach(ticket -> {
                    responseObserver.onNext(ticket);
                });
            }
        }

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

gRPC - Bidirectional Streaming

El cliente define un observer para las respuestas y llama recibiendo el observer para enviar los requests.

En este ejemplo usamos un latch

```
final List<Ticket> tickets = new ArrayList<>();  
final CountDownLatch finishLatch = new CountDownLatch(1);  
final StreamObserver<Ticket> observer = new StreamObserver<Ticket>() {  
    @Override  
    public void onNext(final Ticket ticket) {  
        tickets.add(ticket);  
    }  
  
    @Override  
    public void onError(final Throwable throwable) {  
        finishLatch.countDown();  
    }  
  
    @Override  
    public void onCompleted() {  
        finishLatch.countDown();  
    }  
};
```

En este caso solo se puede utilizar el *stub

gRPC - Bidirectional Streaming

```
final StreamObserver<Reservation> reservations =  
    trainTicketServiceStub.getTickets(observer);  
  
reservationIds.forEach(id -> {  
    final Reservation reservation = Reservation.newBuilder().setId(id).build();  
    reservations.onNext(reservation);  
});  
  
reservations.onCompleted();  
finishLatch.await();
```

Usamos el latch porque en este caso nos convenia mas que el CompletableFuture



Ejercicio 4 - Bidirectional Streaming

1. Implementar el método **GetTicketsForReservations**
- 



Error Handling



gRPC - Errors


Todo mensaje gRPC puede devolver o un valor o un error.

Para devolver errores el framework define mensajes de Status que cada lenguaje implementa.

En principio hay dos modelos de status para utilizar.

- **io.grpc.Status**: el modelo básico del framework.
- **com.google.rpc.Status**: un modelo más rico

Y para enviar los mensajes de manera sincrónica se utiliza `StreamObserver::OnError`, a menos que se esté transmitiendo en Streaming



gRPC - io.grpc.Status

En el modelo básico el error se puede crear a partir de un código de status o una excepción.

```
responseObserver.onError(io.grpc.Status.INVALID_ARGUMENT
    .withDescription("The commodity is not supported")
    .asRuntimeException());
```

Estas son las dos formas de hacerlos

```
try {
    ...
} catch (Exception exception) {
    throw Status.fromThrowable(exception)
        .withDescription("something happened")
        .asRuntimeException();
}
```


gRPC - com.google.rpc.Status

Al usar el modelo enriquecido se pueden agregar elementos definidos en [error_details.proto](#) lo que permite dar más valores

```
ErrorInfo errorInfo = ErrorInfo.newBuilder()
    .setReason("Resource not found")
    .setDomain("Product")
    .build();

com.google.rpc.Status status = com.google.rpc.Status.newBuilder()
    .setCode(Code.NOT_FOUND.getNumber())
    .setMessage("Product id not found")
    .addDetails(Any.pack(errorInfo))
    .build();


responseObserver.onError(
    StatusProto.toStatusRuntimeException(status)
);
```



gRPC - Streams

Con streams hay que tener cuidado porque al llamar al `onError` se va a cortar la comunicación del cliente con el servidor.

En caso de querer mantener la conexión abierta pero ir enviando el error que tal vez uno de los mensajes provocó, se recomienda incorporar el error como parte de la respuesta del servicio.





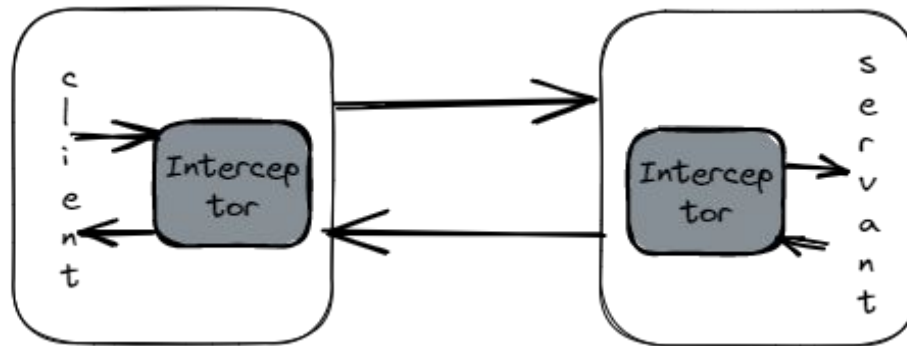
Interceptors



gRPC - Interceptors

Los Interceptors son **elementos opcionales** que se pueden definir tanto en el cliente como en el servidor.

La intención es **interceptar cada llamado y agregar funcionalidad “cross” a todos los llamados**. (como el middleware, se usa para autenticacion, logueo, etc)



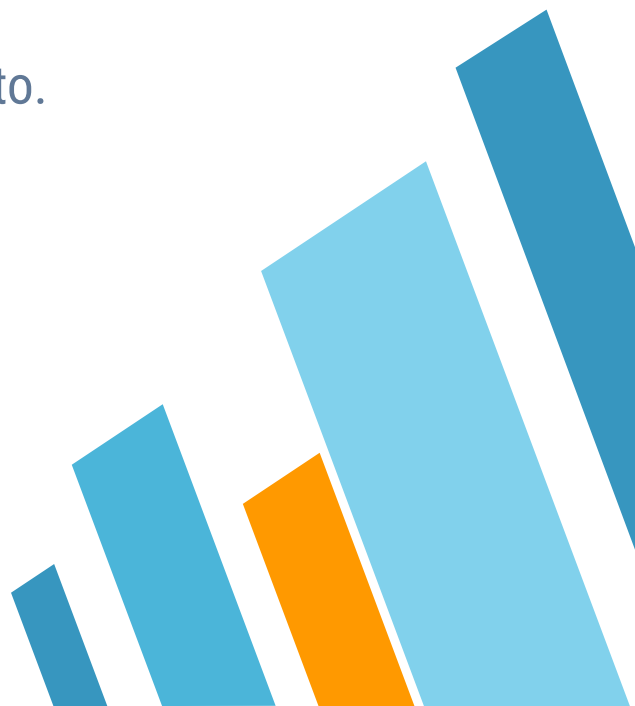


gRPC - Interceptors

Alguna de los **casos de uso** que se puede utilizar un interceptor en un llamado puede ser:

- Métricas
- Log de los llamados
- Autenticación
- Autorización
- Control de errores
- Agregar Headers especiales o Metadata de contexto.

Los interceptores pueden encadenarse por lo que se puede agregar uno por cada responsabilidad



gRPC - Client Interceptor

Para interceptar el mensaje en el cliente se implementa `ClientInterceptor` y se procesa antes de que se ejecute el llamado (`channel.newCall`)

```
public class LoggerClientInterceptor implements ClientInterceptor {

    private static final Logger logger =
        LoggerFactory.getLogger(LoggerClientInterceptor.class);

    @Override
    public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(
        final MethodDescriptor<ReqT, RespT> methodDescriptor,
        final CallOptions callOptions, Channel channel) {
        // Here goes your codes.
        final String rpcId = UUID.randomUUID().toString();
        final String serviceName = methodDescriptor.getServiceName();
        final String methodName = methodDescriptor.getBareMethodName();
        logger.info("Call {}: {}#{}", rpcId, serviceName, methodName);
        // Here goes your codes.
        return channel.newCall(methodDescriptor, callOptions);
    }
}
```

gRPC - Client Interceptor

Para interceptar la respuesta en el cliente se puede utilizar un `SimpleForwardingClientCall` y overridear el `start`.

```
public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(
    final MethodDescriptor<ReqT, RespT> methodDescriptor,
    final CallOptions callOptions,
    final Channel channel) {

    return new ForwardingClientCall.SimpleForwardingClientCall<ReqT,
RespT>(
        channel.newCall(methodDescriptor, callOptions)) {

        @Override
        public void start(ClientCall.Listener<RespT> responseListener,
Metadata headers) {
            logger.info("Setting userToken in header");
            //headers.put(Metadata.Key.of("JWT",
super.start(responseListener, headers);
        }
    };
}
```



gRPC - Client Interceptor

Para agregar un interceptor al cliente se lo agrega a la definición del channel

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress("localhost", 50051)
    .intercept(clientLoggerInterceptor, responseLogger)
    .usePlaintext().build();
```


gRPC - Server Side Interceptor

Para el servidor se implementa una instancia del ServerInterceptor y se puede interceptar el request antes de que se lo envíe al servidor.

```
public class ServerRequestLoggerInterceptor implements ServerInterceptor {
    private static final Logger logger =
        LoggerFactory.getLogger(ServerRequestLoggerInterceptor.class);

    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(
        ServerCall<ReqT, RespT> call, Metadata headers,
        ServerCallHandler<ReqT, RespT> next) {
        // Here goes your codes.
        final String rpcId = UUID.randomUUID().toString();
        final MethodDescriptor<ReqT, RespT> methodDescriptor =
            call.getMethodDescriptor();
        final String serviceName = methodDescriptor.getServiceName();
        final String methodName = methodDescriptor.getBareMethodName();
        logger.info("Call {}: {}#{}", rpcId, serviceName, methodName);
        // Here goes your codes.
        return next.startCall(call, headers);
    }
}
```

gRPC - Server Side Interceptor

Para interceptar la respuesta, similar al cliente


```
public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(  
    ServerCall<ReqT, RespT> serverCall, Metadata metadata,  
    ServerCallHandler<ReqT, RespT> next) {  
  
    return next.startCall(  
        new  
        ForwardingServerCall.SimpleForwardingServerCall<>(serverCall) {  
            @Override  
            public void sendMessage(RespT message) {  
                logger.info("Message being sent to client : "  
+ message);  
  
                super.sendMessage(message);  
            }  
        },  
        metadata);  
    }  
}
```



gRPC - Server Side Interceptor

Para agregar un interceptor al servidor se lo agrega a la definición del channel

```
io.grpc.Server server = ServerBuilder.forPort(port)
    .addService(
        ServerInterceptors.intercept(
            new TrainTicketServer(new TicketRepository()),
            serverLoggerInterceptor
        )
    )
    .build();
```



Ejercicio 4 - Interceptor

1. Agregar el interceptor de logueo para el request en el cliente y en el servidor.
2. Descargar el **GlobalExceptionHandlerInterceptor.java** y analizar su uso.

El `GlobalExceptionHandlerInterceptor` nos deja simplificar el manejo de excepciones del servidor, haciendo handle de las excepciones que se van tirando por los distintos servicios (sin necesidad de hacer try-catch dentro de cada uno). Hay que agregar al mapa 'errorCodesByException' las excepciones y los handler.