



gRPC - Streaming

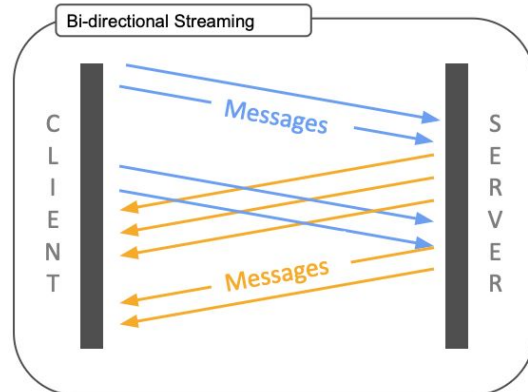
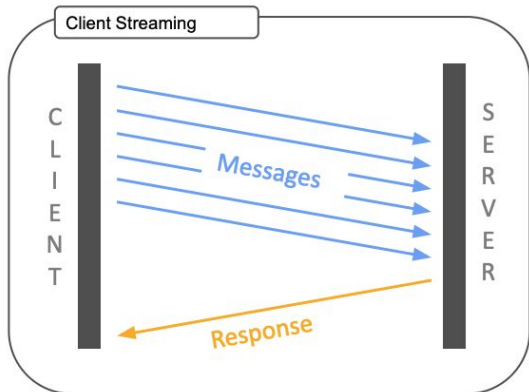
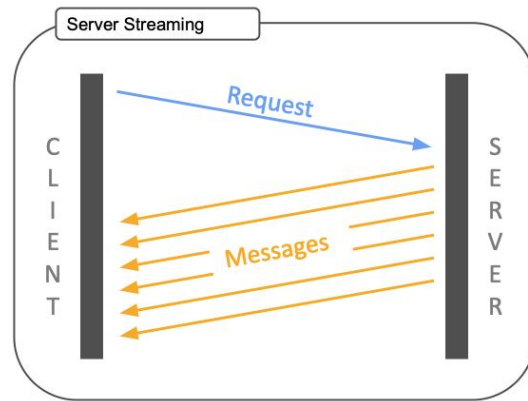
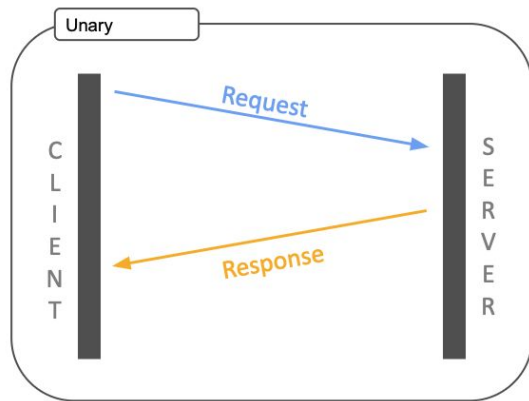
Sistemas Distribuidos



Streaming

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 `train_ticket_service.proto` al módulo `api`
3. Incorporar `TicketRepository.java` y `Train.java` al módulo `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);  
}
```

Server Streaming

Client Streaming

Bi-directional Streaming

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.



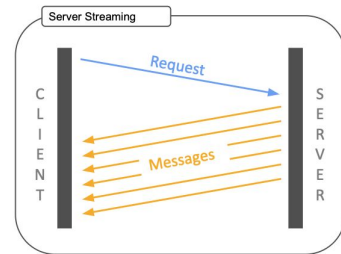
gRPC - Streaming

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 requests/responses.

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



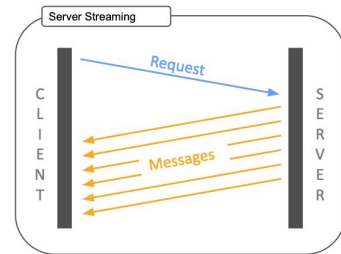
gRPC - Server Side



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

```
service TrainTicketService {  
    rpc GetTrainsForDestination(google.protobuf.StringValue) returns (stream Train);  
}  
  
message Train {  
    string id = 1;  
    string destination = 2;  
    string time = 3;  
    int32 available_count = 4;  
}
```

gRPC - Server Side



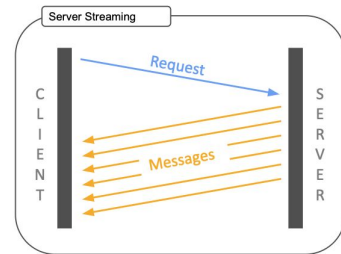
El servidor se implementa similar al unario pero el servidor llama varias veces al **onNext**

```
@Override
public void getTrainsForDestination(StringValue request,
                                   StreamObserver<Train> responseObserver) {
    String destination = request.getValue();

    List<ar.edu.itba.pod.server.ticket.repository.Train> toReturn =
        ticketRepository.getAvailability(destination);

    toReturn.forEach(train → responseObserver.onNext(Train.newBuilder()
        .setId(train.id())
        ...
        .build()));
    responseObserver.onCompleted();
}
```

gRPC - Server Side



El Cliente recibe un **iterador** de respuestas

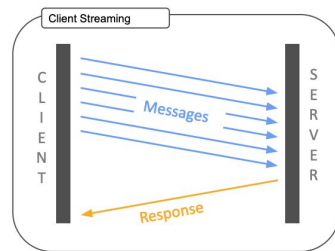
```
TrainTicketServiceGrpc.TrainTicketServiceBlockingStub blockingStub =  
    TrainTicketServiceGrpc.newBlockingStub(channel);  
  
StringValue request = StringValue.of("Mar del Plata");  
  
Iterator<Train> trainsForDestination =  
    blockingStub.getTrainsForDestination(request);  
  
while (trainsForDestination.hasNext()) {  
    System.out.println(trainsForDestination.next());  
}
```

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

Ejercicio 2 - Server Side Streaming

1. Implementar el método `GetTrainsForDestination` y un cliente que lo consuma

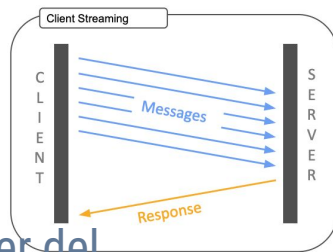
gRPC - Client Side



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

```
service TrainTicketService {  
  rpc PurchaseTicket(stream Ticket) returns (Reservation);  
}  
message Ticket {  
  string id = 1;  
  string trainId = 2;  
  string passenger_name = 3;  
}  
message Reservation {  
  string id = 1;  
  int32 ticket_count = 2;  
}
```

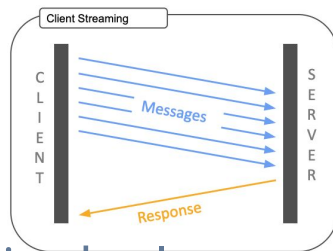
gRPC - Client Side



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.

```
@Override
public StreamObserver<Ticket> purchaseTicket(StreamObserver<Reservation> responseObserver) {
    return new StreamObserver<>() {
        private final List<Ticket> tickets = new ArrayList<>();
        @Override public void onNext(Ticket ticket) {
            tickets.add(ticket);
        }
        @Override public void onCompleted() {
            String reservationId = ticketRepository.addReservation(tickets);
            Reservation reservation = Reservation.newBuilder()
                .setId(reservationId)
                ...
                .build();
            responseObserver.onNext(reservation);
            responseObserver.onCompleted();
        }
        @Override public void onError(Throwable throwable) { ... }
    };
}
```

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

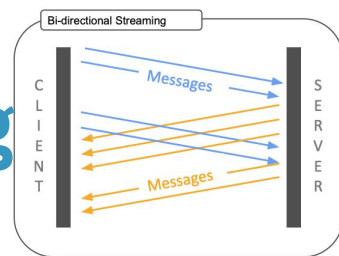
```
TrainTicketServiceGrpc.TrainTicketServiceStub stub = TrainTicketServiceGrpc.newStub(channel);
CompletableFuture<Reservation> reservation = new CompletableFuture<>();
StreamObserver<Reservation> purchaseResponse = new StreamObserver<>() {
    @Override public void onNext(Reservation r) {
        reservation.complete(r);
    }
    @Override public void onCompleted() { ... }
    @Override public void onError(Throwable throwable) { ... }
};
StreamObserver<Ticket> ticketStreamObserver = stub.purchaseTicket(purchaseResponse);
Arrays.asList("John", "Paul", "Ringo").forEach(name -> {
    ticketStreamObserver.onNext(Ticket.newBuilder().setPassengerName(name).build());
});
ticketStreamObserver.onCompleted();
Reservation toReturn = reservation.get();
System.out.println(toReturn);
String reservationId = toReturn.getId();
```

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 lo consuma

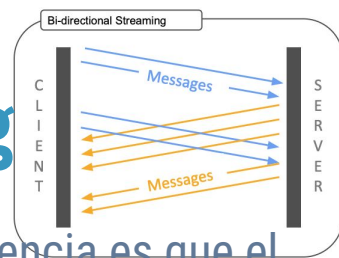
gRPC - Bidirectional Streaming



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

```
service TrainTicketService {  
    rpc GetTicketsForReservations(stream Reservation) returns (stream Ticket);  
}
```

gRPC - Bidirectional Streaming

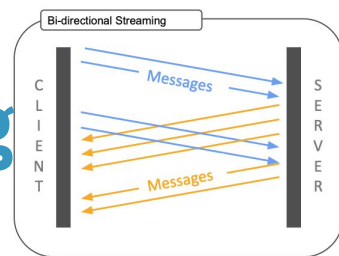


En el servidor el código es igual que en el streaming desde el cliente, la diferencia es que el servicio puede enviar más de una respuesta por cada elemento en vez de esperar a recibir todos los elementos del request

```
@Override
public StreamObserver<Reservation> getTicketsForReservations(
    StreamObserver<Ticket> responseObserver) {

    return new StreamObserver<>() {
        @Override
        public void onNext(Reservation reservation) {
            Optional<List<Ticket>> tickets =
                ticketRepository.getReservation(reservation.getId());
            if (tickets.isPresent()) {
                tickets.get().forEach(ticket → responseObserver.onNext(ticket));
            } else { ... }
        }
        @Override public void onCompleted() { responseObserver.onCompleted(); }
        @Override public void onError(Throwable throwable) { ... }
    };
}
```

gRPC - Bidirectional Streaming



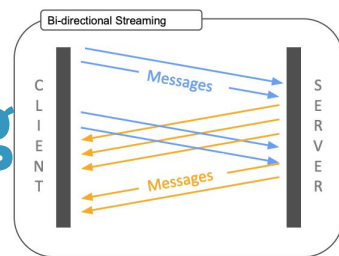
El cliente define un observer para las respuestas y llama recibiendo el observer para enviar los requests.

```
List<Ticket> tickets = new ArrayList<>();

CountDownLatch finishLatch = new CountDownLatch(1);

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

gRPC - Bidirectional Streaming



```
StreamObserver<Reservation> reservations = stub.getTicketsForReservations(observer);  
  
List<String> reservationIds = List.of(reservationId);  
  
reservationIds.forEach(id → {  
    Reservation r = Reservation.newBuilder().setId(id).build();  
    reservations.onNext(r);  
});  
  
reservations.onCompleted();  
  
finishLatch.await();  
  
tickets.forEach(System.out::println);
```

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

Ejercicio 4 - Bidirectional Streaming

1. Implementar el método `GetTicketsForReservations` y un cliente que lo consuma



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());
```

```
Exception in thread "main" io.grpc.StatusRuntimeException:  
INVALID_ARGUMENT: The commodity is not supported
```

```
try {  
    int aux = 3 / 0;  
} catch (Exception ex) {  
    throw Status.fromThrowable(ex)  
        .withDescription("Something Happened")  
        .asRuntimeException();  
}
```

```
Exception in thread "main" io.grpc.StatusRuntimeException:  
UNKNOWN
```


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(com.google.rpc.Code.NOT_FOUND.getNumber())  
    .setMessage("Product id not found")  
    .addDetails(Any.pack(errorInfo))  
    .build();  
  
responseObserver.onError(  
    io.grpc.protobuf.StatusProto.toStatusRuntimeException(status)  
);
```

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

```
try {  
    // ...  
} catch (StatusRuntimeException ex) {  
    Status status = StatusProto.fromThrowable(ex);  
    if (status != null) {  
        logger.error("Message: {}", status.getMessage());  
        for (Any detail : status.getDetailsList()) {  
            if (detail.is(ErrorInfo.class)) {  
                ErrorInfo errorInfo = detail.unpack(ErrorInfo.class);  
                logger.error("Reason: {}", errorInfo.getReason());  
                logger.error("Domain: {}", errorInfo.getDomain());  
            }  
        }  
    } else {  
        ...  
    }  
}
```


```
2025-04-16 11:49:53 ERROR Client:49 - Message: Product id not found  
2025-04-16 11:49:53 ERROR Client:53 - Reason: Resource not found  
2025-04-16 11:49:53 ERROR Client:54 - Domain: Product
```



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.





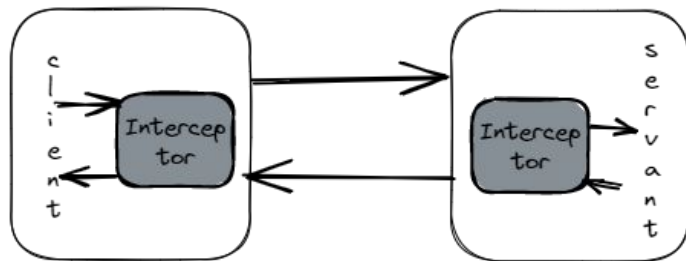
Interceptors



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.






gRPC - Interceptors

Algunos de los casos de uso que se pueden utilizar un interceptor en un llamado pueden 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 ClientLoggerInterceptor implements ClientInterceptor {
    private static final Logger logger = ...

    @Override
    public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(
        MethodDescriptor<ReqT, RespT> methodDescriptor,
        CallOptions callOptions, Channel channel) {
        // Here goes your code #1
        String rpcId = UUID.randomUUID().toString();
        String serviceName = methodDescriptor.getServiceName();
        String methodName = methodDescriptor.getBareMethodName();
        logger.info("Call {}: {}#{}", rpcId, serviceName, methodName);
        // Here goes your code #2
        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 class ClientResponseLogger implements ClientInterceptor {
    private static final Logger logger = ...

    public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(MethodDescriptor<ReqT, RespT>
methodDescriptor, CallOptions callOptions, Channel channel) {
        return new ForwardingClientCall.SimpleForwardingClientCall<>(
            channel.newCall(methodDescriptor, callOptions)) {
            @Override
            public void start(Listener<RespT> responseListener, Metadata headers) {
                logger.info("Setting userToken in header");
                // Here goes your code
                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)
    .usePlaintext()
    .intercept(new ClientLoggerInterceptor(), new ClientResponseLogger())
    .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 = ...

    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(
        ServerCall<ReqT, RespT> call, Metadata headers,
        ServerCallHandler<ReqT, RespT> next) {
        // Here goes your code #1
        String rpcId = UUID.randomUUID().toString();
        MethodDescriptor<ReqT, RespT> methodDescriptor = call.getMethodDescriptor();
        String serviceName = methodDescriptor.getServiceName();
        String methodName = methodDescriptor.getBareMethodName();
        logger.info("Call {}: {}#{}", rpcId, serviceName, methodName);
        // Here goes your code #2
        return next.startCall(call, headers);
    }
}
```

gRPC - Server Side Interceptor

Para interceptar la respuesta, similar al cliente

```
public class GrpcServerResponseInterceptor implements ServerInterceptor {
    private static final Logger logger = ...


    @Override
    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(new TrainTicketServant(ticketRepository))
    .intercept(new ServerRequestLoggerInterceptor())
    .intercept(new GrpcServerResponseInterceptor())
    .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.



CREDITS

Content of the slides:

- » POD - ITBA

Images:

- » POD - ITBA
- » Or obtained from: commons.wikimedia.org

Slides theme credit:

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](https://slidescarnival.com)
 - » Photographs by [Unsplash](https://unsplash.com)
- 