Programación Distribuida y Tiempo Real
Facultad de Informática - Universidad Nacional de La Plata

Práctica 3

Integrantes: Ana Mariela Cossio Aquino, Leandro David Svetlich

1) Utilizando como base el programa ejemplo 1 de gRPC:

a.- Mostrar experimentos donde se produzcan errores de conectividad del lado del cliente y del lado del servidor. Si es necesario realice cambios mínimos para, por ejemplo, incluir sleep() o exit(), de forma tal que no se reciban comunicaciones o no haya receptor para las comunicaciones.

Experimento 1:

La primera prueba consistió en llamar al método remoto desde un nuevo thread, e interrumpir dicho thread antes de que retorne la respuesta del servidor.
Nos encontramos con que del lado del cliente se arroja la excepción StatusRuntimeException con estado de cancelado.
Del lado del servidor no se observó ningún error, sino que finalizó normalmente.

```java
@Override
public void operacion1(
        ServiceOuterClass.Request request,
        StreamObserver<ServiceOuterClass.Response> responseObserver
) {
    System.out.println("Servidor --> Operacion 1");
    try {
        System.out.println("Servidor --> Se duerme durante 5s");
        Thread.sleep( millis: 5000);
        System.out.println("Servidor --> Me desperte para seguir atendiendo al pedido");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    responseObserver.onNext( v: null);
    responseObserver.onCompleted();
}
```

```
Cliente --> crea un hilo
Cliente --> llama a operacion1
Cliente --> interrumpe al hilo
[WARNING]
io.grpc.StatusRuntimeException: CANCELLED
    at io.grpc.Status.asRuntimeException (Status.java:517)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:121)
    at pdytr.example.grpc.ServiceGrpc$ServiceBlockingStub.operacion1 (ServiceGrpc.java:186)
    at pdytr.example.grpc.client.Client$1.run (Client.java:39)
Caused by: java.lang.InterruptedException
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.reportInterruptAfterWait (AbstractQueuedSynchronizer.java:2014)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await (AbstractQueuedSynchronizer.java:2048)
    at java.util.concurrent.LinkedBlockingQueue.take (LinkedBlockingQueue.java:442)
    at io.grpc.stub.ClientCalls$ThreadlessExecutor.waitAndDrain (ClientCalls.java:589)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:118)
    at pdytr.example.grpc.ServiceGrpc$ServiceBlockingStub.operacion1 (ServiceGrpc.java:186)
    at pdytr.example.grpc.client.Client$1.run (Client.java:39)
```

Experimento 2:
Consistió en hacer que el cliente llame a un método remoto, el cual finaliza sin retornar nada ni llamar a onNext(). En este caso, el cliente se queda

indeterminadamente esperando por el retorno. En ambas partes no se observa que se arroje algún error.

Servidor

```
@Override
public void operacion2(
        ServiceOuterClass.Request request,
        StreamObserver<ServiceOuterClass.Response> responseObserver
) { }
```

Cliente

```
private static void operacion2(){
    System.out.println("Cliente --> llama a operacion2");
    stub.operacion2( request: null);
}
```

Experimento 3:
En este caso se invocó a la operación del experimento anterior, pero ahora se hizo sin antes haber iniciado el servidor. Como resultado, se arrojó la excepción StatusRuntimeException con el estado UNAVAILABLE.

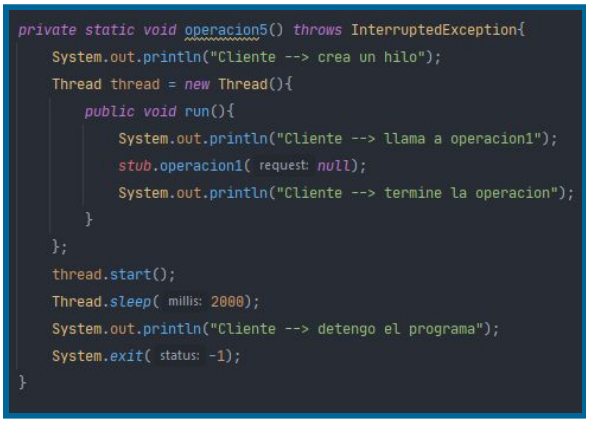
```
Cliente --> llama a operacion2
[WARNING]
io.grpc.StatusRuntimeException: UNAVAILABLE
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:210)
    at io.grpc.stub.ClientCalls.getUnchecked (ClientCalls.java:191)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:124)
    at pdytr.example.grpc.ServiceGrpc$ServiceBlockingStub.operacion2 (ServiceGrpc.java:193)
    at pdytr.example.grpc.client.Client.operacion2 (Client.java:68)
    at pdytr.example.grpc.client.Client.main (Client.java:36)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:254)
    at java.lang.Thread.run (Thread.java:748)
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused: localhost/127.0.0.1:8080
```

Experimento 4:
En este experimento, el cliente llama a una operación del servidor, la cual se duerme durante dos segundos, y luego llama a System.exit(-1). Además se arroja un IllegalThreadStateException.

```
Cliente --> llama a operacion4
[WARNING]
io.grpc.StatusRuntimeException: UNAVAILABLE: Network closed for unknown reason
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:210)
```

```
java.lang.IllegalThreadStateException
    at java.lang.ThreadGroup.destroy (ThreadGroup.java:778)
    at org.codehaus.mojo.exec.ExecJavaMojo.execute (ExecJavaMojo.java:293)
```

Experimento 5:
De manera similar al experimento 1, el cliente llama al método remoto en un nuevo hilo, pero a diferencia del anterior, en vez de interrumpir al hilo, detiene al programa mediante un System.exit(-1). No se arrojó ninguna excepción de parte del servidor.



```
private static void operacion5() throws InterruptedException{
    System.out.println("Cliente --> crea un hilo");
    Thread thread = new Thread(){
        public void run(){
            System.out.println("Cliente --> llama a operacion1");
            stub.operacion1( request: null);
            System.out.println("Cliente --> termine la operacion");
        }
    };
    thread.start();
    Thread.sleep( millis: 2000);
    System.out.println("Cliente --> detengo el programa");
    System.exit( status: -1);
}
```

2) Describir y analizar los tipos de API que tiene gRPC. Desarrolle una conclusión acerca de cuál es la mejor opción para los siguientes escenarios:

- Un sistema de pub/sub
- Un sistema de archivos FTP
- Un sistema de chat

Grpc cuenta con cuatro tipos diferentes de métodos de servicio:

RPC unario: el cliente envía un solo requerimiento al servidor, y recibe una sola respuesta.

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

Server streaming RPC: el cliente envía un solo requerimiento al servidor, y recibe como respuesta un stream de donde leer una secuencia de mensajes. El cliente continúa leyendo mensajes de dicho stream hasta que no haya más mensajes.

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

Client streaming RPC: en este caso es el cliente quien escribe una secuencia de mensajes y las envía al servidor mediante un stream. Una vez el cliente escribe los mensajes, queda a la espera de que el servidor los lea y retorne su respuesta.

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

RPC con streaming bidireccional: en este último son ambas partes, cliente y servidor, quienes envían una secuencia de mensajes mediante un stream de lectura-escritura. Estos streams son independientes entre sí, por lo que la lectura/escritura puede darse en el orden que cada uno desee. El servidor podría esperar a recibir todos los mensajes del cliente y luego escribir, alternar lecturas y escrituras , o hacerlo en

alguna otra combinación. En todos los casos de streaming, gRPC asegura que se preserva el orden de los mensajes.

Ya que en un sistema publish-subscribe los publicadores y los suscriptores se encuentran desacoplados, se puede adoptar una estrategia diferente para cada uno. Una buena opción sería que el publicador utilice RPC unario o bien Client streaming; esto dependerá de la frecuencia con la que este publique mensajes, si publica mensajes a una frecuencia muy alta la opción elegida sería Client streaming, de otra forma alcanzaría con utilizar el clásico RPC unario.
Para los suscriptores la opción a utilizar sería Server streaming, ya que de otra forma deberían preguntar periódicamente al servidor si hay nuevas publicaciones. No sería necesario usar la opción de streaming bidireccional ya que los suscriptores solo reciben mensajes del servidor. Si necesitan comunicarse con el servidor pueden hacerlo pero por otra vía y ya no con el rol de suscriptores.

En un sistema de archivos FTP, podrían diferenciarse las operaciones de lectura y escritura, y adoptar una estrategia diferente para una. Cuando un cliente quiere escribir un archivo en el servidor, la opción a utilizar sería Client streaming RPC, ya que es el cliente quien tendrá que ir mandando el archivo de a bloques, mientras que el servidor solo tiene que recibir estos bloques y almacenarlos en su file system.
Para la operación lectura los roles se invierten, ahora es el servidor quien tendrá que enviar el archivo en bloques, y el cliente tendrá que leer cada uno de estos mensajes para almacenarlos en su file system.

En una aplicación de chat, cada cliente podrá tanto enviarle mensajes a otros usuarios, como recibir mensajes de ellos. Para empezar a recibir mensajes es necesaria una operación de conexión, cada cliente debe autenticarse, y una vez autenticados, recibir un stream de mensajes de parte del servidor. El servidor deberá escribir los mensajes dirigidos a dicho usuario en el stream, para que este pueda leerlos en tiempo real. Por este motivo se utilizará server streaming RPC.
El envío de mensajes de parte del cliente puede hacerse mediante client streaming RPC o bien simplemente RPC unario. Esto puede depender de la cantidad de clientes que maneje el sistema ya que hay que tener en cuenta que el servidor tiene que mantener muchas conexiones abiertas.

3) Analizar la transparencia de gRPC en cuanto al manejo de parámetros de los procedimientos remotos. Considerar lo que sucede en el caso de los valores de retorno. Puede aprovechar el ejemplo provisto.

Por defecto gRPC utiliza Protocol Buffers, el cual se trata de un mecanismo para serializar estructuras de datos. También puede usarse con otros formatos de datos como por ejemplo JSON.

Cuando se utiliza protocol buffers, se debe definir la estructura de los mensajes y la interfaz de los servicios en un archivo de texto .proto. A esta descripción se la llama contrato, y se puede usar tanto en servicios como en clientes, aún estando estos sobre distintas plataformas de desarrollo.

Al usar el archivo proto, el compilador protoc genera tanto el código del cliente y el servicio, el cual incluye los mensajes que intercambian, una clase base sobre la cual puede extender el servicio remoto, y un esqueleto del cliente que contiene lo necesario para hacer invocaciones al servicio remoto.

Los métodos del servicio gRPC deben tener solo un mensaje de entrada y uno de salida obligatoriamente. Normalmente estos mensajes se usan como entrada y salida de un solo método, ya que de esta manera se facilita el hecho de añadir nuevos campos a los mensajes en el futuro, manteniendo la compatibilidad con versiones anteriores. Por lo tanto, en el caso de que no se requiera ningún parámetro, simplemente puede definirse un mensaje vacío, y en el caso de que en un futuro surja la necesidad de uno, simplemente puede agregarse a la definición en el contrato.

En tiempo de ejecución, cada mensaje se serializa con su representación Protobuf estándar, y se intercambia mediante el cliente y el servicio remoto. A diferencia de otros formatos como JSON o XML, los mensajes son serializados como bytes binarios compilados.

Por estos motivos podría decirse que el manejo de parámetros en gRPC no es totalmente transparente, ya que el cliente debe ser consciente de estas restricciones.

4) Con la finalidad de contar con una versión muy restringida de un sistema de archivos remoto, en el cual se puedan llevar a cabo las operaciones enunciadas informalmente como

- leer: dado un nombre de archivo, una posición y una cantidad de bytes a leer, retorna 1) los bytes efectivamente leídos desde la posición pedida y la cantidad pedida en caso de ser posible, y 2) la cantidad de bytes que efectivamente se retornan leídos.
- escribir: dado un nombre de archivo, una cantidad de bytes determinada, y un buffer a partir del cual están los datos, se escriben los datos en el archivo dado. Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos. En todos los casos se retorna la cantidad de bytes escritos.

a.- Defina e implemente con gRPC un servidor. Documente todas las decisiones tomadas.

Para implementar el servidor se tomó como base el código de ejemplo provisto por la cátedra.

Comenzamos por especificar los tipos de mensajes, que reflejan los parámetros de entrada y salida de cada operación, que se detallan en el enunciado.
Además se especifica la interfaz del servicio de archivos, es decir del servidor. gRPC funciona particularmente bien con protocol buffers, permitiendo generar código importante de RPC directamente a través de archivos .proto mediante un plugin.

```
message ReadRequest {
  string name = 1;
  int32 pos = 2;
  int32 count = 3;
}

message ReadResponse {
  int32 count = 1;
  bytes data = 2;
}

message WriteRequest {
  string name = 1;
  int32 count = 3;
  bytes data = 2;
}

message WriteResponse {
  int32 count = 1;
}

service FileService {
    rpc read(ReadRequest) returns (ReadResponse);
    rpc write(WriteRequest) returns (WriteResponse);
}
```

Para ambas operaciones se utilizaron operaciones RPC unarias, ya que las operaciones solicitadas operan sobre pequeñas porciones del archivo, y no transfieren archivos en su totalidad.

En general la implementación no difiere demasiado a la realizada con RMI; se mantienen las operaciones de lectura y escritura sobre los archivos y las estructuras de control. Obviamente ahora se está utilizando gRPC así que hay algunas diferencias.
De parte del cliente, lo más notable además de la inicialización del canal de comunicación, es el pasaje de parámetros a los métodos remotos ya que ahora deben pasarse a través de un solo objeto que los contenga.

Cliente:

```
while (response.getCount() > -1) {
    request = FileServiceOuterClass.ReadRequest.newBuilder(request)
            .setCount(bufferSize)
            .setPos((int)offset)
            .build();

    response = stub.read(request);
    if (response.getCount() > -1) {
        output.write(response.getData().toByteArray(), 0, response.getCount());
        offset = offset + response.getCount();
    }
}
```

Servidor:

```
if (!file.exists()) {
  System.out.println("Servidor --> El archivo solicitado no existe.");
  response =
    FileServiceOuterClass.ReadResponse.newBuilder().setCount(-1).build();
} else {
  try {
    InputStream input = new FileInputStream(file);
    int readCount; // cantidad leida en un read
    byte[] buffer = new byte[bufferSize];
    input.skip(request.getPos());
    readCount = input.read(buffer, 0, request.getCount());
    response =
      FileServiceOuterClass.ReadResponse.newBuilder().setCount(readCount).setData(ByteString.copyFrom(buffer))
        .build();
    input.close();
  } catch (Exception e) {
    System.out.println("Servidor -> Se produjo un error.");
    e.printStackTrace();
  }
}
responseObserver.onNext(response);
responseObserver.onCompleted();
```

Adicionalmente se implementó una versión del método leer que utiliza Server Streaming RPC. A diferencia de lo solicitado en el enunciado, este método recibe solo el nombre del archivo a leer, y devuelve el archivo en su totalidad en forma de stream de ReadResponse al cliente. Una vez invocado el método, el cliente itera sobre la estructura recibida y escribe los datos del archivo leído en su file system.

```
while(responses.hasNext()){
    response = responses.next();

    System.out.println(response.getCount());
    if(response.getCount() > -1) {
        output.write(response.getData().toByteArray(), 0, response.getCount());
    }
}
```

El cliente se bloquea en responses.hasNext() hasta recibir una nueva respuesta, o bien hasta que retorna falso.
Del lado del servidor, este lee el archivo hasta que se lean -1 bytes, y cada vez que lo hace llama a streamObserver.onNext(response) para que el cliente reciba en el stream esa nueva respuesta en el stream.

```
while (readCount > -1) {
    readCount = input.read(buffer, 0, bufferSize);
    response =
                FileServiceOuterClass.ReadResponse.newBuilder()
                        .setData(ByteString.copyFrom(buffer))
                        .setCount(readCount).build();
    streamObserver.onNext(response);
}
```

Al utilizar Server Streaming RPC hubo una importante mejora en cuanto a la velocidad de transferencia. Hay que tener en cuenta que si se trabaja con archivos muy grandes, es necesario implementar un control de flujo del lado del servidor ya que pueden producirse errores por la alta utilización de memoria. La prueba se realizó con un archivo mp4 de 48.608KB.

RPC unario

```
Cliente  -> Nombre del archivo: server-nature.mp4
Cliente  -> Operacion: read
Cliente  -> Transferencia completada.
Cliente  -> Tiempo: 6134ms
```

Server Streaming RPC

```
Cliente  -> Nombre del archivo: server-nature.mp4
Cliente  -> Operacion: readServerStreaming
Cliente  -> Tiempo: 1545ms
Cliente  -> Transferencia completada.
```

En cuanto a la operación escribir, se procedió de manera similar, el cliente debe iterar leyendo bloques del archivo a transferir desde su file system, y llamar a la operación write del servidor, pasando como parámetro un WriteRequest con la cantidad de bytes a escribir y los bytes en sí. También hubiera sido posible utilizar Client Streaming RPC para realizar la escritura de un archivo en el servidor. En dicho caso, la operación write del servidor deberá retornar al cliente un StreamObserver, sobre el cual podrá llamar a la operación onNext(WriteRequest).

b.- Investigue si es posible que varias invocaciones remotas estén ejecutándose concurrentemente y si esto es apropiado o no para el servidor de archivos del ejercicio anterior. En caso de que no sea apropiado, analice si es posible proveer una solución (enunciar/describir una solución, no es necesario implementarla). Nota : diseñe un experimento con el que se pueda demostrar fehacientemente que dos o más invocaciones remotas se ejecutan concurrentemente o no.

Es posible hacer muchas invocaciones a métodos remotos y que se ejecuten concurrentemente, ya que cada invocación se ejecuta en un hilo diferente.

En el servidor de archivos implementado en el ejercicio anterior, la ejecución simultánea de métodos que operan sobre los mismos archivos, ya sea leyendo o escribiendo puede ocasionar errores de sobreescritura y pérdida de información. Esto se debe a dos motivos; El primero es que la operación de escritura del FileOutputStream no es atómica, por lo que puede haber varios hilos escribiendo o leyendo el archivo al mismo tiempo. Lo mismo sucede con el read del

FileInputStream. El otro motivo es que para guardar o leer un archivo en el servidor, el cliente debe hacer reiterados llamados los métodos remoto escribir y leer, entonces, aunque las operaciones write/read del FileOutputStream/FileInputStream fueran atómicas, todavía habría problemas de sincronización que resolver.
En principio, si se trata de clientes diferentes que guardan los archivos leídos en distintos directorios, no hay ningún problema y pueden hacerlo concurrentemente. El problema es que puede que suceda que un cliente está leyendo un archivo, mientras otro lo está escribiendo. Esta situación no se encuentra controlada en la implementación del punto anterior.
Una posible solución al primer problema sería utilizar File locks para garantizar la exclusión mutua en la escritura/lectura de los archivos.

El experimento realizado consiste en ejecutar el servidor del inciso anterior, y correr tres clientes que leen y escriben los mismos archivos de manera concurrente. De esta manera se ve claramente que se están ejecutando de manera concurrente, y además se exponen algunos de los problemas de nuestra implementación mencionados anteriormente

5) Timeouts en gRPC:
a.- Desarrollar un experimento que muestre el timeout definido para las llamadas gRPC y el promedio de tiempo de una llamada gRPC.

En gRPC no hay un timeout definido por defecto, lo cual aplica para todos los lenguajes que soporta.
Para calcular el tiempo promedio de una llamada gRPC, se desarrolló un experimento que consiste en hacer llamadas a un método del servidor que retorna sin hacer nada.
El tiempo de retorno es contabilizado por parte del cliente, quien se encarga de almacenar cada uno de ellos para luego calcular el promedio y la desviación estándar.

```
for (int i = 0; i < sampleSize; i++) {
    start = System.currentTimeMillis();
    stub.metodoInutil(null);
    end = System.currentTimeMillis();
    diff = end - start;
    samples[i] = diff;
    total = total + diff;
}
final double std = calcularDE(samples);
System.out.println("Total: " + total + "ms");
System.out.println("Promedio: " + total / sampleSize + "ms");
System.out.println("Desviacion estandar: " + std + "ms");
```

Resultados obtenidos:

Tamaño de muestra	10000
Total	5100.0ms
Promedio	0.51ms
Desviación estándar	3.039720381876984ms

b.- Reducir el timeout de las llamadas gRPC a un 10% menos del promedio encontrado anteriormente. Mostrar y explicar el resultado para 10 llamadas.

El experimento realizado consistió en establecer un timeout del lado del cliente, indicando como tiempo 459000 nanosegundos que es el 90% de los 51 milisegundos del promedio obtenido en el experimento anterior.

Al ejecutar el programa, vemos que en ciertas ocasiones, del lado del servidor se arroja la excepción:

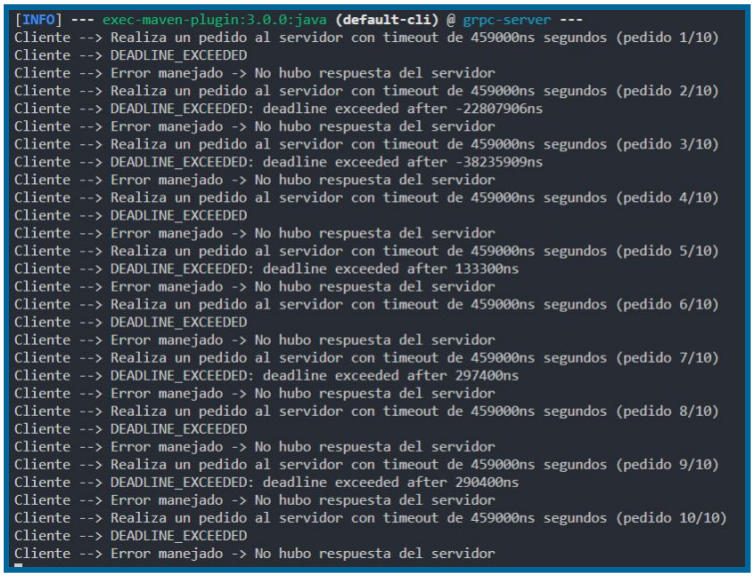
```
io.grpc.netty.NettyServerHandler onStreamError WARNING: Stream Error

io.netty.handler.codec.http2.Http2Exception$StreamException: Received DATA frame for an unknown stream 5
```

Es posible que este comportamiento resulte de que el servidor reacciona al deadline establecido por el cliente, antes que el propio cliente. Cuando se desencadena la cancelación en consecuencia del deadline, el servidor se olvida de la información que estaba recibiendo en el stream, por lo que si aún había datos en camino, cuando llegan al servidor, este no lo recuerda. Por eso la excepción indica que se recibieron datos para un stream desconocido.

Log del servidor:

```
Nov 08, 2020 9:28:41 PM io.grpc.netty.NettyServerHandler onStreamError
WARNING: Stream Error
io.netty.handler.codec.http2.Http2Exception$StreamException: Received DATA frame for an unknown stream 3
        at io.netty.handler.codec.http2.Http2Exception.streamError(Http2Exception.java:129)
        at io.netty.handler.codec.http2.DefaultHttp2ConnectionDecoder$FrameReadListener.shouldIgnoreHeadersOrDataFrame(DefaultHttp2C
onnectionDecoder.java:535)
        at io.netty.handler.codec.http2.DefaultHttp2ConnectionDecoder$FrameReadListener.onDataRead(DefaultHttp2ConnectionDecoder.jav
a:187)
        at io.netty.handler.codec.http2.Http2InboundFrameLogger$1.onDataRead(Http2InboundFrameLogger.java:48)
```

Log del cliente:



c.- Desarrollar un cliente/servidor gRPC de forma tal que siempre se supere el tiempo de timeout. Una forma sencilla puede utilizar el tiempo de timeout como parámetro del procedimiento remoto, donde se lo utiliza del lado del servidor en una llamada a sleep( ), por ejemplo.

Para desarrollar dicho programa, comenzamos por definir los datos del protocol buffer, donde especificamos el tipo de mensaje Request, que posee un campo timeout.
El cliente simplemente genera un valor aleatorio que hará de deadline, y dicho valor será enviado al servidor para que este se duerma durante ese mismo tiempo al cual le incrementamos un segundo más.
De esa manera se asegura que siempre se superará el tiempo de timeout ya que en ocasiones, si el servidor se duerme el mismo tiempo que el cliente, este llega a retornar y el cliente no arroja la excepción StatusRuntimeException, que es lo que se espera.

Cliente:

```java
for (int i = 1; i <= 10; i++) {
    timeout = getRandomTimeout();
    System.out.println("Cliente --> Realiza un pedido al servidor con deadline de "+timeout+" segundos (pedido " + i + "/10)");
    request = ServiceOuterClass.Request.newBuilder().setId(i).setTimeout(timeout).build();
    try {
        stub.withDeadlineAfter(timeout , TimeUnit.SECONDS).hacerAlgoConTimeout(request);
    } catch (io.grpc.StatusRuntimeException e) {
        printError(e);
    }
}

channel.shutdownNow();
```

Servidor:

```java
try {
    System.out.println("Servidor --> Se duerme durante  "+(request.getTimeout() + 1) +"s");
    Thread.sleep((request.getTimeout() + 1) * 1000);
    System.out.println("Servidor --> Me desperte para seguir atendiendo al pedido "+request.getId()+"...");
} catch (InterruptedException e) {
    e.printStackTrace();
}

if (Context.current().isCancelled()) {
    System.out.println("Servidor --> El cliente cancelo la operacion " + request.getId());
    System.out.println("Servidor --> Puedo dejar de ejecutar el pedido " + request.getId());
    responseObserver.onError(Status.CANCELLED.withDescription("Cancelled by client").asRuntimeException());
    return;
}
```

Cuando el servidor se despierta, puede verificar si el cliente aún está esperando su retorno. Esto es muy útil cuando se trabaja con grandes cantidades de información, ya que puede evitarle al servidor hacer mucho trabajo de gusto, en caso de que el cliente haya cancelado la operación.