Practica 3 - gRPC

Alumno: Cazorla, Ignacio Nicolás
N° de alumno: 13926/1

Contenidos

1) Utilizando como base el programa ejemplo 1 de gRPC: Mostrar experimentos donde se produzcan errores de conectividad del lado del cliente y del lado del servidor.

a) Si es necesario realice cambios mínimos para, por ejemplo, incluir exit(), de forma tal que no se reciban comunicaciones o no haya receptor para las comunicaciones.

1) Conexión terminada por el cliente antes de obtener la respuesta.

Para el primer caso, se modifican el cliente y el servidor, de forma tal que la operación no sea bloqueante y permita ejecutar un *exit()* antes de obtener la respuesta (para el cliente), y estableciendo una espera del lado del servidor.

La solicitud del cliente se realiza de manera asincrónica con un callback, que imprimirá el mensaje cuando sea recibido desde el servidor.

La forma en que se llevó a cabo esta operatoria, fue utilizando las interfaces **Future**[1] y **ListenableFuture**[2], que permiten realizar tareas asincrónicas y el manejo de estas.

Tanto cliente como servidor implementan una espera (2 y 8 segundos respectivamente) con el método *sleep()*, la cual dará un margen para cerrar la conexión del lado del cliente antes de que el servidor pueda responder.
El resultado es que el servidor recibe el mensaje que le envió el cliente y envía una respuesta que no será recibida.

2) Conexión terminada por el servidor antes de tiempo.

Se modifica solo el servidor original, para que luego de imprimir el mensaje del cliente, termine su ejecución con la operación *exit()*.

A continuación podemos ver la salida del cliente:

```
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.0.0:java (default-cli) on project gr
pc-hello-server: An exception occured while executing the Java class. UNAVAILABLE: Network closed for unk
nown reason -> [Help 1]
```

3) Tiempo de espera

Utilizando el método *sleep()* que había sido usado en 1), podemos establecer un tiempo determinado de espera para un proceso.

Para el experimento se utilizaron esperas del lado del servidor con tiempos de 8 y 18 segundos. Son tiempos de respuesta largos, respecto a lo que sería deseable en una situación real.

---

[1] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html?is-external=true
[2] https://guava.dev/releases/21.0/api/docs/com/google/common/util/concurrent/ListenableFuture.html

Como resultado, el cliente queda esperando el tiempo que sea necesario hasta recibir una respuesta del servidor.

En caso de cortar la ejecución con **ctrl + c** el servidor envía la respuesta y nuevamente no será recibida por el cliente como en el caso 1).

4) No se inicia el servidor.

Utilizando el cliente original y no encendiendo el servidor, el cliente avisa que no puede establecer la conexión.

```
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.GreetingServiceGrpc$GreetingServiceBlockingStub.greeting (GreetingServiceGrpc.java:163)
    at pdytr.example.grpc.Client.main (Client.java:30)
    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
```

Se puede decir entonces, que gRPC maneja una serie de excepciones generales.

b) Configure un DEADLINE y cambie el código (agregando la función sleep()) para que arroje la excepción correspondiente.

En gRPC un **deadline** permite indicar el tiempo que está dispuesto a esperar un cliente por una respuesta del servidor. Por defecto, el valor es un número muy grande y es dependiente del lenguaje en el que está implementado.

Es conveniente setear este valor, de otra manera el cliente se verá esperando por una respuesta indefinidamente.

El método *withDeadlineAfter()* en Java nos permite definir un deadline:

```
GreetingServiceOuterClass.HelloResponse response =
       stub.withDeadlineAfter(5000,
TimeUnit.MILLISECONDS).greeting(request);
```

El primer argumento es el tiempo que se desea esperar, y el segundo indica la unidad usada para expresar el tiempo.

En este caso, el plazo de vencimiento es de 5 segundos a partir del requerimiento (expresado en milisegundos).

Por parte del servidor, se establece una espera de 18 segundos con el método *sleep()*, tiempo más que suficiente para recibir el error que representa un vencimiento del plazo indicado:

```
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.0.0:java (
default-cli) on project grpc-hello-server: An exception occured while executing
 the Java class. DEADLINE_EXCEEDED: deadline exceeded after 4955706187ns -> [He
lp 1]
```

c) Reducir el deadline de las llamadas gRPC a un 10% menos del promedio encontrado anteriormente. Mostrar y explicar el resultado para 10 llamadas.

Utilizando el promedio del ejercicio 5a de 9002 microsegundos, calculamos un 10% menos:

9002 * 0,1 = 900,2 → 9002 - 900,2 = 8101,2

```
stub.withDeadlineAfter(8101, TimeUnit.MICROSECONDS).greeting(request);
```

Configurando el deadline de esta manera se obtienen 10 ejecuciones de 10 que exceden el deadline.

```
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli)
Server started
```

En la salida del servidor se puede apreciar que no llega ningún mensaje.

Reduciendo el tiempo de deadline por debajo del tiempo promedio, las comunicaciones se ven muy afectadas y pueden no realizarse por su vencimiento.

2) Describir y analizar los tipos de API que tiene gRPC.

Los tipos de API que ofrece gRPC son:

Unary RPC: es el caso más simple, dónde un cliente le manda un solo mensaje a un servidor y recibe un solo mensaje por parte de este.

Server streaming RPC: es similar al caso anterior, solo que en este caso el servidor envía una serie de respuestas al cliente y luego un status code y otros mensajes de status opcionales, de esta manera finaliza la ejecución del servidor. La ejecución del cliente finaliza cuando termina de recibir toda la información del servidor

Client streaming RPC: el cliente envía muchos mensajes al servidor.

Bidirectional streaming RPC: tanto cliente como servidor pueden enviar varios mensajes sin importar el orden. Se pueden ir turnando para mandar mensajes o pueden hacer un echo-reply.

Desarrolle una conclusión acerca de cuál es la mejor opción para los siguientes escenarios:
a) Un sistema de pub/sub

Los sistemas publish/subscribe están basados en la ocurrencia de eventos. Por un lado tenemos a los agentes **publishers**, que informan eventos (de manera asincrónica). Por el otro, tenemos los **subscribers**, que toman estos eventos del servicio, siempre y cuando, como su nombre lo indica, estén suscritos a dicho servicio.

En general, una **notificación de evento** habrá sido producida por un publisher y será recibida por muchos subscribers. La tarea de un sistema de este tipo será entregar las notificaciones de un evento a los suscriptores que tengan interés en ellas.

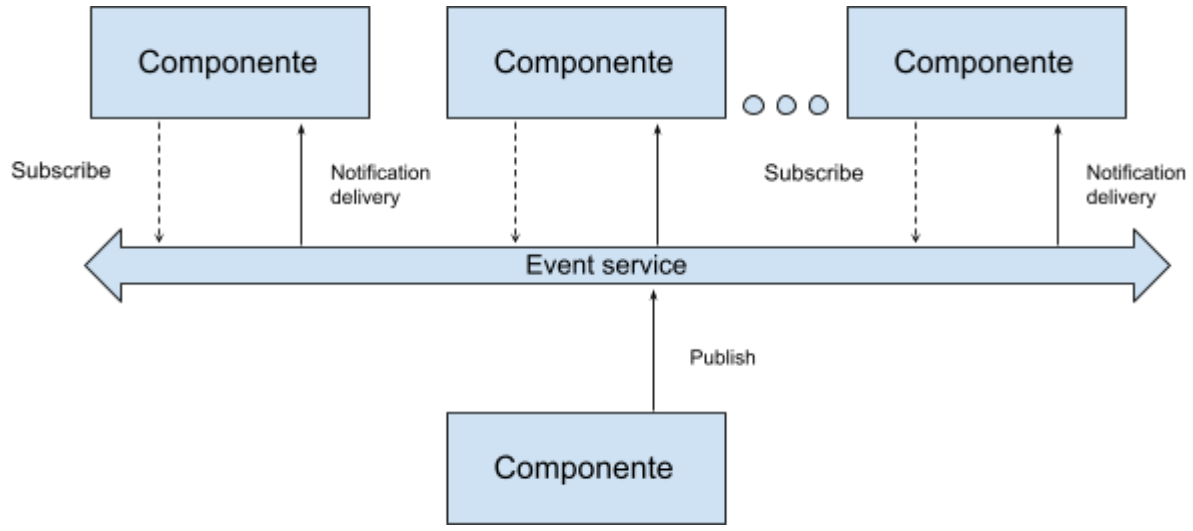


Figura 2.a

Es importante destacar que esta arquitectura produce eventos que deben ser consumidos en ese momento, es decir, una vez enviados deben ser consumidos ya que no habrá otra oportunidad de hacerlo, porque los eventos no se almacenarían. De todas maneras, es posible implementar el sistema con persistencia para contar con eventos que se produjeron al menos por cierta cantidad de tiempo.

En esta oportunidad tomaremos la implementación sin persistencia. Los componentes involucrados serán un publisher, un servidor y ***n*** subscribers.
En base a lo planteado, para construir un sistema con esta arquitectura y usando gRPC, podemos pensar las interacciones de los componentes de la siguiente manera :

- **Subscribers**: necesitan enviar un mensaje para suscribirse y luego recibir todas las notificaciones que se produzcan mientras estén suscriptos.
- **Publishers**: van a enviar mensajes cada vez que haya eventos, es decir que es posible que envíe muchos mensajes.
- **Servidor**: recibe todos los mensajes del publisher y los envía a los subscribers.

Utilizaremos tres de las APIs, mensajes unarios por parte de los publishers (cliente) hacia el servidor, streaming por parte del publisher (cliente) hacia el servidor y streaming por parte del servidor hacia los subscribers (clientes). Si observamos la **Figura 2.a** podemos interpretar los componentes superiores como suscriptores donde una flecha punteada es el mensaje unario de suscripción y una flecha continua representa un streaming desde el servidor donde se recibirán una o varias notificaciones. Luego, el componente inferior es el publisher y la flecha continua representa nuevamente un streaming, pero esta vez del cliente hacia el servidor.

La finalidad de utilizar dichas APIs de esta manera es la de poder brindarle las interacciones necesarias a los componentes. Puede ser útil, tener streamings desde el publisher hacia el servidor porque no es necesario recibir ningún mensaje del último y además el publisher idealmente envía varios mensajes al servidor. Siguiendo con esto último, es útil para el servidor realizar un streaming de los datos que recibe del publisher para mantener la esencia de esta arquitectura y dando la impresión de que el publisher manda directamente a los subscribers por la manera en la que se decidió enviar los mensajes desde el servidor, y generando cierta abstracción.

b) Un sistema de archivos FTP

El escenario para un sistema de este tipo va a estar representado por dos componentes, un cliente y un servidor. El servidor tendrá el sistema de archivos, del cual el cliente puede realizar una lectura o una escritura. Se puede apreciar que este sistema es muy sencillo.

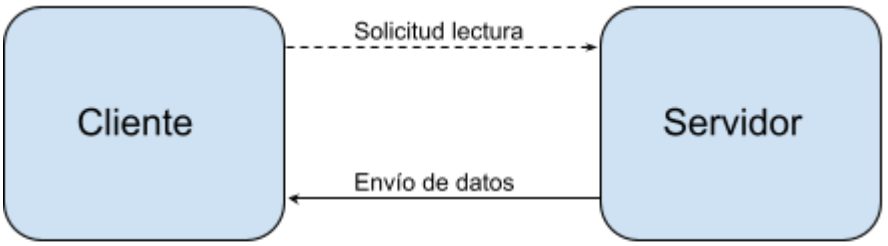


Figura 2.b.1

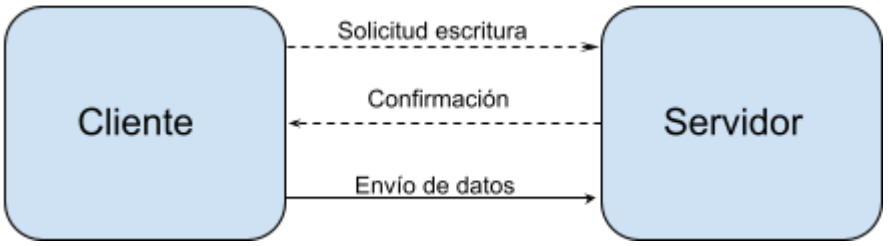


Figura 2.b.2

El cliente siempre inicia la interacción. En las figuras, se tiene una operación de lectura y una de escritura. Nuevamente como en el inciso anterior podemos interpretar las flechas continuas como un streaming y las punteadas como un único mensaje entre cliente y

servidor. Entonces, desde el punto de vista de la cantidad de interacciones que es necesaria para llevar a cabo las operaciones, se opta por tener la mínima indispensable para lograr el objetivo.
El cliente solo debe indicar qué operación desea realizar y esperar a recibir los datos en caso de lectura; o esperar una confirmación en caso de operación de escritura para luego comenzar a escribir. Es deseable que las operaciones de transferencia se realicen completas sin intercambiar mensajes mientras tanto, para obtener mayor performance.

c) Un sistema de chat

Para este sistema contaremos con los siguientes componentes: dos clientes y un servidor. En este caso queremos proveer una abstracción similar a la del inciso 2a). Es deseable que los clientes perciban la comunicación como si fuese directa. Por este motivo, utilizaremos streaming tanto del lado de los clientes como del servidor. Además, esta situación provee una interacción respecto del orden y la cantidad de mensajes, mucho más natural.

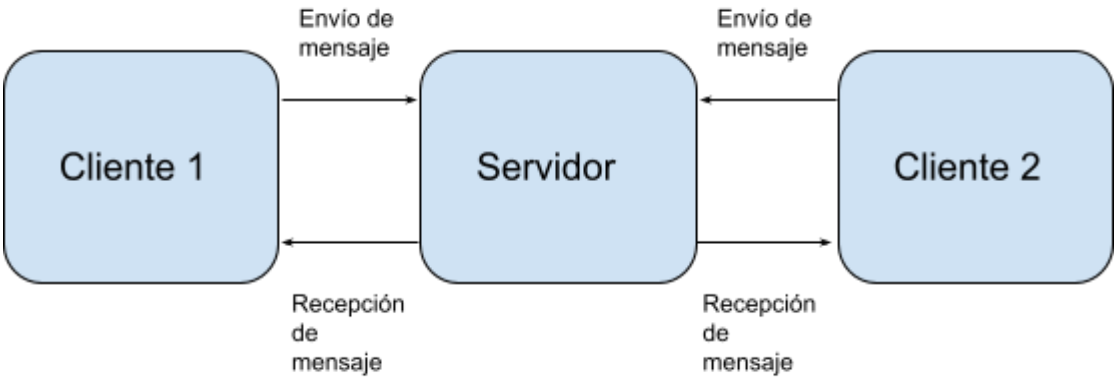


Figura 2.c

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 usa Protocol buffers como mecanismo de serialización. Este tiene un lenguaje para definir la estructura de los datos (**message**) y servicios que van a utilizarse. Podemos definir para cada dato de la estructura el tipo (escalares, enumerativos, etc), un id por campo, reglas (para indicar si es opcional, requerido o si se puede repetir).

Para el programador, el manejo de parámetros es transparente, ya que siempre se envía una estructura y no es necesario realizar ninguna operación adicional para manejar la representación de los datos.

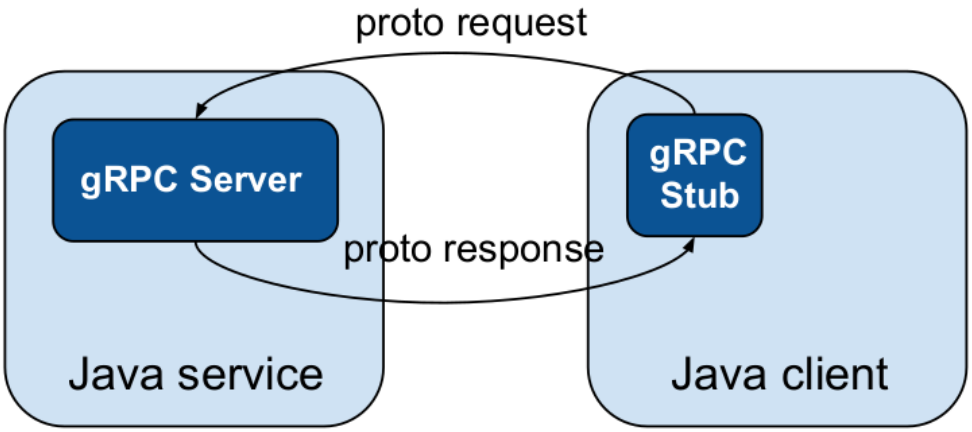
Para los valores de retorno también se usa este mecanismo y se recibe un **message**.

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.

La implementación del sistema de archivos remoto con gRPC consta básicamente de un cliente, un servidor y un archivo **.proto** que define las interacciones entre ambos agentes.



- El cliente tiene definidas requests : WriteRequest y ReadRequest.
- El servidor tiene definidas responses : WriteResponse y ReadResponse.

Los servicios provistos por el servidor son **leer()** y **escribir()**.

Ejecución

Se indican a continuación, los pasos para leer el archivo **cancion.mp3** desde el servidor y luego escribirlo allí como **cancion_copia.mp3**.

Para ejecutar el servidor:

```
$ mvn -DskipTests package exec:java
```

```
-Dexec.mainClass=pdytr.example.grpc.App
```

Para ejecutar un cliente que lee del servidor:

```
$ mvn -DskipTests exec:java -Dexec.mainClass=pdytr.example.grpc.Client
-Dexec.args="-r ./database/server/cancion.mp3
./database/cliente/cancion.mp3"
```

Para escribir en el servidor:

```
$ mvn -DskipTests exec:java -Dexec.mainClass=pdytr.example.grpc.Client
-Dexec.args="-w ./database/cliente/cancion.mp3
./database/server/cancion_copia.mp3"
```

Los archivos transferidos son iguales, lo comprobamos con el comando **diff**:

```
root:database#
root:database# diff cliente/cancion.mp3 server/cancion.mp3
root:database#
```

```
root:database#
root:database# diff server/cancion.mp3 server/cancion_copia.mp3
root:database#
```

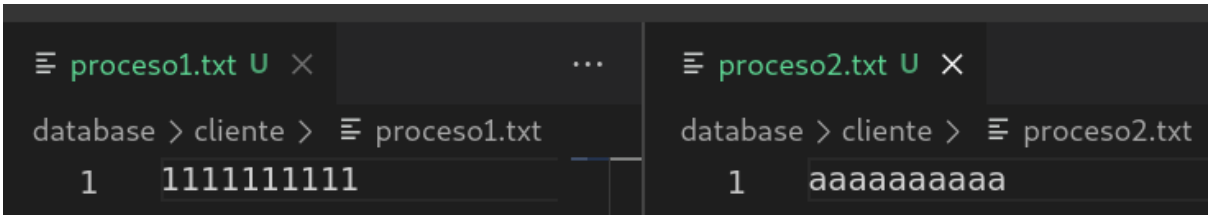
a) Defina e implemente con gRPC un servidor. Documente todas las decisiones tomadas.

Para el servidor implementado se tuvieron presentes las siguientes decisiones:

1. El servidor realizará, en base a lo requerido, una sola respuesta por petición.
2. Para la operación de escritura, no se tiene en cuenta ninguna condición fuera de lo solicitado, “Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos.”, es decir que el directorio deberá estar preparado para realizar esta operación. Utilizar el script **ej4_cleanup.sh** para limpiar los directorios antes de ejecutar el programa.

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.

Para realizar el experimento se van a utilizar dos clientes. Cada uno de ellos va a intentar escribir desde un archivo distinto (**proceso1.txt** y **proceso2.txt**), el mismo archivo en el servidor. Esta operatoria se repetirá 10 veces.



Contenido de los archivos que se intentarán escribir en el servidor.

Además se va a reducir el tamaño de los datos que se escriben por cada invocación, a una ventana de 2 (dos) bytes. Este tamaño se indica como parámetro adicional al programa. No es deseable el uso de este parámetro bajo ninguna otra circunstancia.

```
$ ./ejercicio4b.sh
```

```
Ejecucion 10
[INFO] Termino la operacion de escritura.
[INFO] Termino la operacion de escritura.

Termino la ejecucion. Ver los resultados en output/ejercicio4.txt.
```

Final de output del script ejecutado.

Ejecucion 1 1111aa1111aa Ejecucion 2 aaaaaaaaaa11 Ejecucion 3 1111aa1111aa Ejecucion 4 aaaaaaaaaa11 Ejecucion 5 1111111111aa	Ejecucion 6 aaaaaaaaaa11 Ejecucion 7 aa11aaaaaa11 Ejecucion 8 1111111111aa Ejecucion 9 aaaaaaaaaa11 Ejecucion 10 1111111111aa

Se puede apreciar en la tabla anterior que hay una interferencia cuando ambos clientes intentan escribir a la vez en el mismo archivo. Esto no es apropiado para un servidor de archivos tan sencillo como el que fue implementado.

Una posible solución para este inconveniente podría ser que no se permita escribir a más de un cliente por vez en el mismo archivo. Cuando un cliente quiere escribir un archivo, solicita permiso. Si el archivo no está siendo utilizado por nadie más se le asigna a dicho cliente, y hasta que termine en su totalidad la operación que está realizando ningún otro cliente podrá utilizarlo.

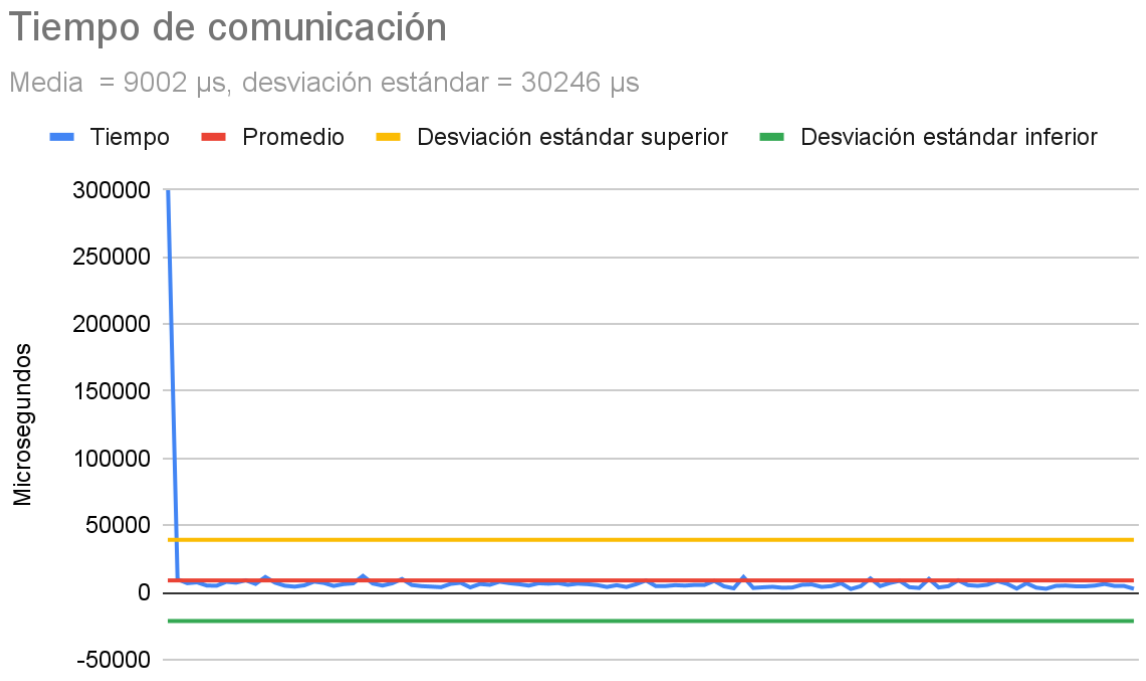
5) Tiempos de respuesta de una invocación.

a) Diseñe un experimento que muestre el tiempo de respuesta mínimo de una invocación con gRPC. Muestre promedio y desviación estándar de tiempo respuesta.

Se modifica la clase Cliente del programa de ejemplo.
Para medir los tiempos de respuesta, se toma un punto previo a la invocación del método *greeting()* y el tiempo posterior a obtener la respuesta.

Ejecutar con:

```
$ mvn -DskipTests exec:java -Dexec.mainClass=pdytr.example.grpc.Client
-Dexec.args="100"
```



b) Utilizando los datos obtenidos en la Práctica 1 (Socket) realice un análisis de los tiempos y sus diferencias. Desarrollar una conclusión sobre los beneficios y complicaciones tiene una herramienta sobre la otra.

En relación a los tiempos obtenidos para socket, donde tenemos un promedio de 97 microsegundos, el tiempo de respuesta es mucho mayor en gRPC.

Desde el punto de vista de gRPC, como beneficio podemos mencionar el uso de Protocol buffers, ya que provee abstracción suficiente como para que cada agente que interactúa en un sistema, pueda estar implementado en distintos lenguajes. Por el contrario con sockets, contamos solo con la abstracción necesaria para enviar y recibir información, pero la forma en la que esta se interpreta debe realizarse “a mano”.

Bibliografía

gRPC, https://grpc.io/docs/what-is-grpc/introduction/
https://grpc.io/docs/what-is-grpc/core-concepts/ .

Deadlines en gRPC, https://grpc.io/blog/deadlines/ .

Protocol Buffers, https://developers.google.com/protocol-buffers/docs/overview .

Capítulo 6, *Distributed systems - Concepts and design.*