

Programación distribuida y tiempo real

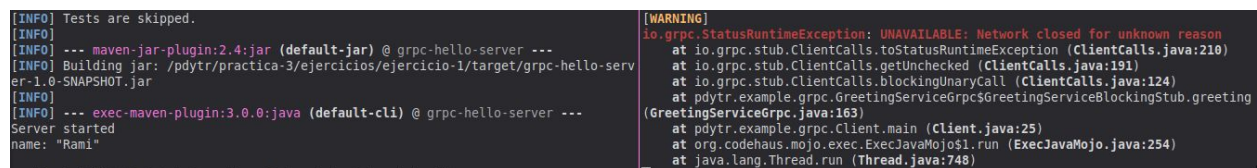
Entrega 3

Ejercicio 1	2
Ejercicio 2	4
Ejercicio 3	7
Ejercicio 4	8
Inciso A	8
Inciso B	11
Ejercicio 5	12
Inciso A	12
Inciso B	13
Inciso C	14

Ejercicio 1

Experimentos para hacer fallar al cliente

1º experimento: Agregamos la línea **System.exit(77)** en la implementación del servicio en la interfaz, esto provocará que el servidor termine y el cliente nunca reciba una respuesta lanzando una excepción.



```
[INFO] Tests are skipped.
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ grpc-hello-server ---
[INFO] Building jar: /pdytr/practica-3/ejercicios/ejercicio-1/target/grpc-hello-server-1.0-SNAPSHOT.jar
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ grpc-hello-server ---
Server started
name: "Rami"

[WARNING] io.grpc.StatusRuntimeException: UNAVAILABLE: Network closed for unknown reason
    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:25)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:254)
    at java.lang.Thread.run (Thread.java:748)
```

Este experimento se puede ver en el paquete *experimento1* del proyecto *ejercicios/ejercicio-1*.

Ejecutar servidor:

```
mvn -DskipTests package exec:java -Dexec.mainClass=pdytr.example.experimento1.App
```

Ejecutar cliente:

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.example.experimento1.Client
```

2º experimento: Agregamos la línea **Thread.sleep(600000)** en la implementación de la interfaz, esto provocará que el servidor se duerma 10 minutos y el cliente espere 10 minutos. Como resultado, luego de los 10 minutos el cliente obtuvo la respuesta satisfactoriamente sin ninguna excepción.

Este experimento se puede ver en el paquete *experimento2* del proyecto *ejercicios/ejercicio-1*.

Ejecutar servidor:

```
mvn -DskipTests package exec:java -Dexec.mainClass=pdytr.example.experimento2.App
```

Ejecutar cliente:

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.example.experimento2.Client
```

Experimentos para hacer fallar al servidor

3º experimento: Agregamos un thread del lado del cliente que ejecute la línea **System.exit(1)** luego de hacer el llamado al servicio. Mientras que al servidor le agregamos la línea **Thread.sleep(6000)** para que luego de 6 segundos envíe la respuesta y se encuentre con el

cliente cerrado. El resultado fue que el servidor envía la respuesta satisfactoriamente sin rastros de errores.

Este experimento se puede ver en el paquete *experimento3* del proyecto *ejercicios/ejercicio-1*.

Ejecutar servidor:

```
mvn -DskipTests package exec:java -Dexec.mainClass=ptytr.example.experimento3.App
```

Ejecutar cliente:

```
mvn -DskipTests exec:java -Dexec.mainClass=ptytr.example.experimento3.Client
```

4º experimento: Cambiamos el stub del cliente a asincrónico y luego de realizar una petición cerramos la conexión del cliente. Mientras tanto, el servidor hará un sleep de 6 segundos (como en el experimento anterior) y cuando envíe la respuesta el cliente estará cerrado. El resultado fue que el servidor envía la respuesta satisfactoriamente sin rastros de errores.

Este experimento se puede ver en el paquete *experimento4* del proyecto *ejercicios/ejercicio-1*.

Ejecutar servidor:

```
mvn -DskipTests package exec:java -Dexec.mainClass=ptytr.example.experimento4.App
```

Ejecutar cliente:

```
mvn -DskipTests exec:java -Dexec.mainClass=ptytr.example.experimento4.Client
```

Ejercicio 2

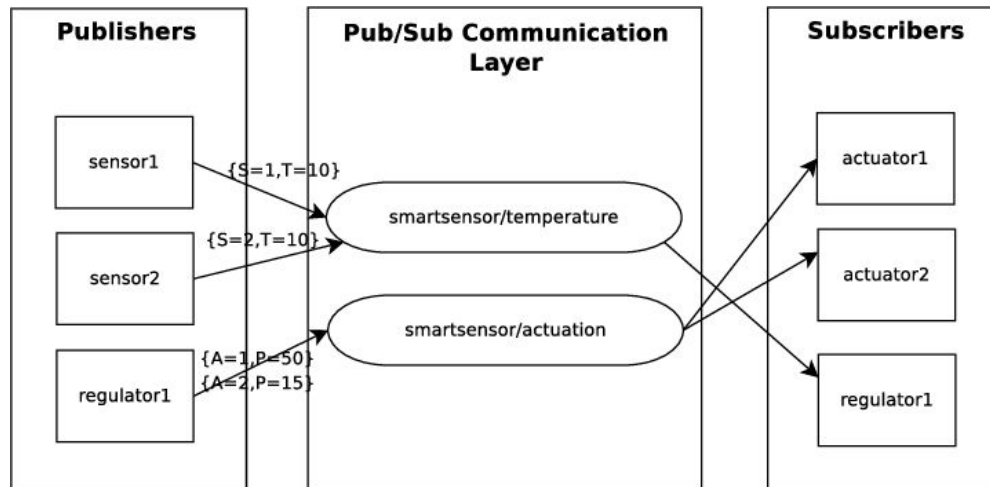
Los tipos de API que ofrece gRPC son:

- **Unary RPCs:** La comunicación es punto a punto, es decir, el cliente envía un requerimiento y el server envía una respuesta.
- **Server streaming RPCs:** El cliente manda un requerimiento y recibe un stream del servidor, por lo tanto, el cliente va a leer hasta que el stream se marque como completado. Esto le permite al cliente recibir muchos mensajes del servidor en 1 comunicación establecida.
- **Client streaming RPCs:** El cliente manda un stream como requerimiento y el server lo lee hasta que el stream se marque como completado. Esto le permite al servidor recibir muchos mensajes del cliente en 1 comunicación establecida.
- **Bidirectional streaming RPC:** Tanto cliente como el server se comunican los datos a través de streams, esto le permite a ambos intercambiar mensajes en la comunicación establecida.

Fuente: <https://grpc.io/docs/what-is-grpc/core-concepts/>

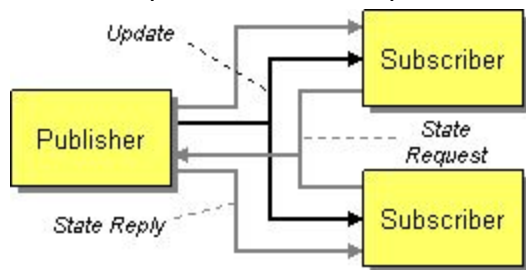
Publishers-Subscribers

En el esquema en donde tenemos un servidor que atiende, tanto pedidos de los publishers, como los subscribers:



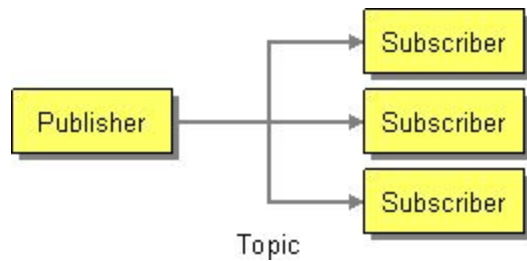
Se podría utilizar el esquema **Server streaming** para los clientes *suscribers*, los mismos enviarán una solicitud al servicio *subscribe* y el server retornará el stream de eventos. Por lo tanto, el *suscriber* monitoreará el stream y en caso de que surja un evento nuevo, el cliente lo podrá leer desde el stream recibido sin necesidad de establecer una nueva conexión. Por otra parte, los clientes *publishers* enviarán una solicitud al servicio *publish* como **Client streaming**, esto le permitirá intercambiar un stream de eventos y cada vez que se genere un nuevo evento se lo comunicará al servidor a través del stream.

En otro esquema, el servidor podría ser el publisher, tal como se ve en la siguiente imagen:



Para este esquema alcanzaría sólo con usar **Server streaming** para los clientes *suscribers*, y al igual que en el anterior, los clientes pedirán suscribirse a través de un servicio, luego el servidor *publisher* retornará el stream de eventos y el cliente podrá monitorear este stream para saber si ocurrió un evento nuevo.

También puede se puede optar el esquema inverso, es decir, los *suscribers* pueden verse como servidores y el *publisher* como cliente:



En este caso se podría utilizar **Client streaming**, en donde el publisher llamará a un método *publish* e intercambiará su stream de eventos con todos los *subscribers*. Por lo tanto, cuando se genere un nuevo evento, el *publish* lo escribirá en dicho stream y los *subscribers* lo podrán leer.

FTP

Como vimos en las entregas anteriores, para mandar un archivo se necesitan muchas comunicaciones dado a que se va pasando de a porciones de bytes. Por lo tanto, si el cliente quiere escribir un archivo conviene utilizar **Client streaming** para pasar esas porciones en un stream en la misma conexión establecida.

En cambio, si se quiere leer desde el servidor conviene utilizar **Server streaming** ya que el cliente recibe un stream del archivo del servidor y puede leerlo desde allí en la misma comunicación establecida.

Optamos por usar **streamings** en vez de una comunicación **unidireccional**, ya que permite hacer menor cantidad de establecimientos de conexión. Es decir, si por ejemplo el cliente quiere leer un archivo muy grande de 1GB con una ventana de 1MB, tendrá que llamar 1000 veces a un servicio gRPC. En cambio, al trabajar con streamings se realiza solo 1 llamada gRPC y luego se va leyendo del streaming que fue intercambiado, lo que provocará un overhead menor a realizar 1000 llamadas gRPC.

También, los streamings de gRPC aseguran atomicidad y el orden de los datos. Por lo tanto, no vamos a tener inconsistencias en la comunicación.

- Fuente: <https://grpc.io/docs/what-is-grpc/core-concepts/>

Chat

En un sistema de chat ambas partes intercambian datos en la misma conexión, por lo tanto conviene utilizar una comunicación bidireccional, es decir, **Bidirectional streaming** ya que le permitirá a ambas partes compartir los datos a través de streams en la misma comunicación.

Ejercicio 3

Para especificar los parámetros en gRPC se utiliza el protocolo *Protocols-Buffer*, el cual los agrupa mediante una estructura llamada mensajes (**message**). Cada mensaje contiene la especificación diferentes parámetros, en donde lo más importante que deberemos especificar es:

- La **cardinalidad**, es decir, si son obligatorios, opcionales o repetidos (0 o más ocurrencias), entre otros.
- El **tipo de dato**, desde tipos simples como **int** hasta tipos complejos como **Enum**.
- El **id** de cada parámetro.
- Entre otras opciones.

Por un lado esta especificación permite una gran transperencia para el programador, ya que siempre enviará como parámetro una estructura de tipo **message** y no se deberá preocupar por la representación de ellos en memoria. Pero por otra parte es un poco limitado, ya que el programador no podrá pasar tipos primitivos como parámetro (ya que si o sí deben estar encapsulados en una estructura **message**), ni tampoco parámetros que no sean por valor. Por lo tanto, esto último le quita algo de transperencia al manejo de parámetros.

Para los valores de retorno se utiliza el mismo mecanismo, se deben especificar a través de *Protocols-Buffers* por cada servicio y siempre se retorna un tipo **message** o **Void** en el caso de que no se quiera retornar nada.

Fuente: <https://developers.google.com/protocol-buffers/docs/overview>

Ejercicio 4

Inciso A

Implementación del servicio **leer**

Servidor

Primero definimos el servicio en *Ftp.proto*. El mismo recibirá como argumento un message **LecturaRequest**, el cual tendrá el nombre del archivo, la posición del archivo y el offset hasta donde se desea leer. Retornará un mensaje **LecturaResponse**, que se compone por el contenido del archivo en bytes y la cantidad de bytes leídos.

Luego, implementamos el servicio **leer** en la interfaz *Ftp.java* de la siguiente manera:
Se comienza abriendo un stream del archivo requerido por el cliente, y se lee desde la posición hasta el offset que le fue indicado por el cliente. Luego, encapsula el contenido y los bytes que fueron leídos en un message **LecturaResponse**.
Finalmente el servidor envía el **message** a través de `onNext()`.

Cliente

Para mostrar el funcionamiento del servicio leer, desarrollamos un cliente que funciona de la siguiente manera:

El cliente comienza estableciendo una ventana fija de bytes que representa las porciones en la cual va a ir leyendo el archivo. Luego, creará un archivo en donde irá guardando los bytes recibidos por el servidor.

Invocamos al servicio leer con un **message** de tipo **LecturaRequest**, el cual se compone de los siguientes parámetros:

1. El nombre del archivo que se quiere leer en el servidor.
2. La posición que va a estar dada por la cantidad de bytes leídos del archivo del servidor (inicialmente será 0).
3. La cantidad de bytes a leer será la ventana de 1024 bytes (*siempre fija*).

De la anterior llamada obtenemos un **message** de tipo **LecturaResponse**, el cual va a contener:

1. Los bytes leídos por el servidor
2. El contenido en bytes de la porción del archivo que pedimos

El cliente escribirá la cantidad de bytes leídos en su propio archivo.

Finalmente todos estos pasos se harán hasta que los bytes leídos recibidos sean menor o igual a 0.

Implementación del servicio **escribir**

Servidor

El servicio leer se declaró en el archivo *Ftp.proto*. Recibe un **message** llamado **EscrituraRequest**, el cual se compone de los siguiente parámetros:

1. Nombre: Será el nombre del archivo que se quiere escribir.
2. Offset: Indica hasta dónde escribirá el contenido.
3. Contenido: Representa el contenido en bytes que se recibe del cliente.
4. Destructiva: Indica si se quiere hacer una escritura destructiva o no.

Retorna un **message** llamado **EscrituraResponse**, el cual está compuesto por los siguientes parámetros:

1. Bytes escritos: Retorna la cantidad total de bytes escritos en el archivo.

Luego, se implementó el servicio **escribir** en la interfaz *Ftp.java* de la siguiente forma:

Se abrirá el archivo especificado por el cliente en el modo que fue indicado por el mismo (destructivo o agregar al final).

Luego, escribe el contenido de bytes en el archivo abierto previamente con el offset que le fue indicado por parámetro. Finalmente, calcula los bytes escritos y los envía a través de un **message EscrituraResponse** al cliente.

Cliente

Para demostrar el funcionamiento del servicio **escribir**, se desarrolló un cliente que funciona de la siguiente manera:

Primero se abre el archivo que se desea transferir al servidor, y se lo va pasando al servidor mediante porciones de una ventana fija de bytes. En cada iteración se invoca al servicio **escribir** con un **message EscrituraRequest** con los siguientes parámetros:

1. El **nombre** del archivo que se desea escribir.
2. Los bytes de la porción del **contenido** del archivo.
3. El **offset**, que será el mínimo entre la ventana y la cantidad de bytes faltantes. En la mayoría de los casos va a ser la ventana si se trata de un archivo muy grande, pero en el caso del último fragmento, si la cantidad faltante no es múltiplo de la ventana

entonces se envían la cantidad de bytes faltantes (es provisto por la librería de Java que usamos para leer el archivo en el cliente).

4. El **modo** de escribir, en dónde en la primera iteración va a ser destructivo, esto es para eliminar el contenido si en el servidor existe un archivo con igual nombre. En el resto de las iteraciones se irá agregando al final.

La iteración se dará hasta que el total de bytes escritos en el archivo del servidor sea igual al tamaño del archivo en el cliente.

Ejecución

La implementación se encuentra en el directorio *ejercicios/ejercicio-4*.

Para ejecutar el servidor:

```
mvn -DskipTests package exec:java -Dexec.mainClass=ptytr.example.grpc.App
```

Para ejecutar el cliente:

```
mvn -DskipTests exec:java  
-Dexec.mainClass=ptytr.example.grpc.Client  
-Dexec.args="<operacion> <path_server> <path_cliente> <ventana>"
```

Se deben especificar los parámetros:

- **operacion:** La operación que se desea realizar: [-leer | -escribir].
- **path_server:** El nombre del path en el lado del servidor.
- **path_cliente:** El nombre del path en el lado del cliente.
- **ventana:** El tamaño de la ventana que se desea usar. Debe ser mayor que 0.

Ejemplo:

```
mvn -DskipTests exec:java  
-Dexec.mainClass=ptytr.example.grpc.Client  
-Dexec.args="-leer database/server/fondo.jpg  
database/client/wallpaper-copia.jpg 1024"
```

Inciso B

Sí, es posible que varias invocaciones remotas se estén ejecutando concurrentemente, pero no es apropiado para el servidor, debido a que puede haber interferencia entre las múltiples comunicaciones simultáneas que tenga el servidor con los clientes. Esta interferencia puede provocar que por ejemplo un archivo del servidor al ser escrito por varios clientes concurrentemente, tenga información errónea (es decir que los bytes enviados por los clientes se mezclen).

Como nuestra implementación escribe el archivo en pequeñas porciones de una ventana de bytes no alcanza con que sea atómica sólo la operación de lectura o escritura, ya que puede haber interferencias incluso si son atómicas. Por lo tanto, debe ser atómica la sesión del cliente hasta que se termine de leer/escribir el archivo deseado.

Entonces llegamos a que la solución es crear una sesión de lectura o escritura para cada cliente y hacer que dicha sesión sea atómica.

Para probar la concurrencia realizamos un experimento en donde 2 clientes escriben un archivo con el mismo nombre en el servidor. Ambos clientes llaman al método **escribir** con diferentes archivo origen (de igual contenido) y con el mismo nombre de archivo destino. Los archivos son:

Del lado de los clientes:

- `/ejercicio-4/database/client/experimento1.txt` → Contiene 20 caracteres 'a'.
- `/ejercicio-4/database/client/experimento2.txt` → Contiene 20 caracteres 'b'

Del lado del servidor:

- `/ejercicio-4/database/server/experimento.txt`

El script que realiza el experimento se encuentra en `ejercicios/scripts/script-4b.sh`.

Realiza 10 ejecuciones donde 2 clientes escriben el mismo archivo de forma concurrente con una ventana de 2 bytes.

Ejecutando el script obtuvimos la siguiente salida (se encuentra en el archivo

`/ejercicios/outputs/ejercicio-4b.txt`):

-----	aaaaaaaaaaaaaaaaaaaaa	Ejecución 5:
Ejecución 0:	-----	bbbbbbbbbbbbbbbbbbbb
aabababbababaabbaabaab	Ejecución 3:	-----
-----	bbbbbbbbbbbbbbbbbbbb	Ejecución 6:
Ejecución 1:	-----	aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa	Ejecución 4:	-----
-----	baaaaabababa bbbbbbbb	Ejecución 7:
Ejecución 2:	-----	abbbbaababababbabababa

Ejecución 8:

aaaaaaaaaaaaaaaaaaaaa

Ejecución 9:

baaaabaabbaabaabbababa

Ejercicio 5

Inciso A

Según miembros de gRPC el tiempo por default de timeout (deadline) es infinito. Es decir, gRPC no pone un límite de timeout si no se lo especifica.

Fuente: <https://github.com/grpc/grpc-java/issues/1495>

El valor del **deadline** es una fecha UTC. Cuando el cliente supera el tiempo de esa fecha se lanza una excepción de tipo **DEADLINE_EXCEEDED**.

Para calcular el promedio de tiempo de una llamada gRPC hicimos lo siguiente:

- Removimos las sentencias del lado del servidor para que no hubiese procesamiento que haga que la comunicación dure más tiempo.
- Modificamos el cliente para que cuente el tiempo en milisegundos utilizando un método de java llamado `System.currentTimeMillis()`. La implementación se encuentra en *ejercicios/ejercicio-5a*.
- Realizamos un script que levanta un servidor y ejecuta varios clientes, los cuales retornan el tiempo que tardan en ejecutarse en milisegundos. El script se encuentra en *scripts/script-5a.sh* (usarlo con la opción **-time**). La salida del script se guarda en *outputs/time-test.csv*.

Calculamos mediante el script, el tiempo de comunicación de **50** llamadas y nos dio un promedio de **497** milisegundos por llamada. Este valor lo usamos luego para setear el deadline del cliente, el cual se configura agregando la siguiente sentencia a la llamada al servicio: `withDeadlineAfter(497, TimeUnit.MILLISECONDS)`

Tiempo de llamada gRPC

Promedio = 497,14 | Desviación estándar = 69,37



Ejecutamos nuevamente el script (con el parámetro **-ejercicio5a**) para que calcule el tiempo de comunicación de **10** llamadas, con el fin de ver cuántas llamadas se completaba con éxito y cuantas finalizaban con la excepción **DEADLINE_EXCEEDED**. Como resultado obtuvimos que **4** de las **10** llamadas finalizaron con error.

```
ramirolopescanadell:outputs:[master]$ cat ejercicio-5a-test.csv | grep "DEADLINE_EXCEEDED" | wc -l  
4
```

Inciso B

Utilizando nuevamente la implementación que se encuentra en *ejercicios/ejercicio-5a* y el script que se encuentra en *scripts/script-5a.sh* (con la opción **-ejercicio5b**).

El deadline tenía seteado el valor **497** milisegundos. Al reducirlo un **10%** nos queda:

- $497 - (497 / 10) = 497 - 49.7 = 447.3$

El cliente nos queda con la sentencia `withDeadlineAfter(447, TimeUnit.MILLISECONDS)`

Ejecutamos el script para que calcule el tiempo de comunicación de **10** llamadas. Como resultado obtuvimos que **6** de las **10** llamadas finalizaron con error.

```
ramirolopescanadell:outputs:[master]$ cat ejercicio-5b-test.csv | grep "DEADLINE_EXCEEDED" | wc -l
6
```

Inciso C

Para desarrollar un cliente/servidor gRPC de forma tal que siempre se supere el tiempo de timeout implementamos lo siguiente:

- Del lado del servidor, hicimos que se realice un loop infinito por lo que nunca se retornará una respuesta al cliente.
- Del lado del cliente, llamamos al servicio con el método:
`withDeadlineAfter(4000, TimeUnit.MILLISECONDS)`

Por lo tanto, siempre que el cliente llame el servicio va a terminar en timeout porque el servidor estará en un loop infinito:

```
[INFO] -----
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.0.0:java (default-cli) on project grpc-hello-server: An exception occurred while executing the Java class. DEADLINE_EXCEEDED: deadline exceeded after 3970988789ns -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

La implementación se encuentra en *ejercicios/ejercicio-5c*.

Para ejecutar el server:

```
mvn -DskipTests package exec:java -Dexec.mainClass=ptytr.example.grpc.App
```

Para ejecutar el cliente:

```
mvn -DskipTests exec:java -Dexec.mainClass=ptytr.example.grpc.Client
```