



Facultad de
INFORMÁTICA



UNIVERSIDAD
NACIONAL
DE LA PLATA

Programación Distribuida y Tiempo Real

Trabajo Práctico 3

- Joaquin Tartaruga (17023/7)
- Manuel Rubiano (17711/6)

1- A) Como el servidor esta hecho para quedarse esperando el cierre, es decir se queda activo hasta que se finalice manualmente su ejecución, poner un `exit()` del lado del cliente no afecta al flujo del programa, ya que cada interacción entre cliente y servidor se ejecuta como una transacción independiente y única. Esto significa que, una vez que el cliente envía su solicitud y el servidor responde, la conexión se cierra y no se mantiene ninguna sesión activa entre cliente y servidor..

Ahora bien, del lado del servidor podemos agregar un `exit()` antes del `start()` para que no reciba mensajes del cliente (es decir no está disponible). O terminarlo una vez el cliente se conecta, haciendo que este se quede dormido y en ese tiempo hacer que el servidor se caiga. En ambos casos se termina produciendo el siguiente error:

```
io.grpc.StatusRuntimeException: UNAVAILABLE: io exception
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:268)
    at io.grpc.stub.ClientCalls.getUnchecked (ClientCalls.java:249)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:167)
    at app.pdytr.BookServiceGrpc$BookServiceBlockingStub.addBook (BookServiceGrpc.java:174)
    at app.pdytr.client.BookClient.main (BookClient.java:47)
    at org.codehaus.mojo.exec.ExecJavaMojo.doMain (ExecJavaMojo.java:358)
    at org.codehaus.mojo.exec.ExecJavaMojo.doExec (ExecJavaMojo.java:347)
    at org.codehaus.mojo.exec.ExecJavaMojo.lambda$execute$0 (ExecJavaMojo.java:269)
    at java.lang.Thread.run (Thread.java:833)
Caused by: io.grpc.netty.shaded.io.netty.channel.AbstractChannel$AnnotatedConnectException: finishConnect(..) failed: Connection refused: localhost/127.0.0.1:8080
Caused by: java.net.ConnectException: finishConnect(..) failed: Connection refused
    at io.grpc.netty.shaded.io.netty.channel.unix.Errors.newConnectException0 (Errors.java:166)
    at io.grpc.netty.shaded.io.netty.channel.unix.Errors.handleConnectErrno (Errors.java:131)
    at io.grpc.netty.shaded.io.netty.channel.unix.Socket.finishConnect (Socket.java:359)
    at io.grpc.netty.shaded.io.netty.channel.epoll.AbstractEpollChannel$AbstractEpollUnsafe.doFinishConnect (AbstractEpollChannel.java:710)
    at io.grpc.netty.shaded.io.netty.channel.epoll.AbstractEpollChannel$AbstractEpollUnsafe.finishConnect (AbstractEpollChannel.java:687)
    at io.grpc.netty.shaded.io.netty.channel.epoll.AbstractEpollChannel$AbstractEpollUnsafe.epollOutReady (AbstractEpollChannel.java:567)
    at io.grpc.netty.shaded.io.netty.channel.epoll.EpollEventLoop.processReady (EpollEventLoop.java:489)
    at io.grpc.netty.shaded.io.netty.channel.epoll.EpollEventLoop.run (EpollEventLoop.java:397)
    at io.grpc.netty.shaded.io.netty.util.concurrent.SingleThreadEventExecutor$4.run (SingleThreadEventExecutor.java:997)
    at io.grpc.netty.shaded.io.netty.util.internal.ThreadExecutorMap$2.run (ThreadExecutorMap.java:74)
    at io.grpc.netty.shaded.io.netty.util.concurrent.FastThreadLocalRunnable.run (FastThreadLocalRunnable.java:30)
    at java.lang.Thread.run (Thread.java:833)
```

B) En esta sección lo que hicimos fue agregar un `sleep` de 10 seg en el proceso `addBook` antes de generar el mensaje de respuesta, y al cliente le agregamos un tiempo de espera máximo de 5 seg cuando hace la llamada, provocando que el servidor nunca conteste el mensaje a tiempo (generando un `deadline`)

```
io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline exceeded after 4.752682900s. [closed=[], open=[[buffered_nanos=620135100, remote_addr=localhost/127.0.0.1:8080]]]
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:268)
    at io.grpc.stub.ClientCalls.getUnchecked (ClientCalls.java:249)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:167)
    at app.pdytr.BookServiceGrpc$BookServiceBlockingStub.addBook (BookServiceGrpc.java:174)
    at app.pdytr.client.BookClient.main (BookClient.java:50)
    at org.codehaus.mojo.exec.ExecJavaMojo.doMain (ExecJavaMojo.java:358)
    at org.codehaus.mojo.exec.ExecJavaMojo.doExec (ExecJavaMojo.java:347)
    at org.codehaus.mojo.exec.ExecJavaMojo.lambda$execute$0 (ExecJavaMojo.java:269)
    at java.lang.Thread.run (Thread.java:833)
```

2- gRPC nos brinda 4 Apis:

- **Unary RPC:** el cliente envía una solicitud y recibe una única respuesta del servidor. Para interacciones directas donde se espera una respuesta inmediata.
- **Server Streaming RPC:** el cliente envía una solicitud y recibe múltiples respuestas del servidor en una secuencia. Cuando el servidor necesita enviar un flujo continuo de datos en respuesta a una única solicitud del cliente.
- **Client Streaming RPC:** el cliente envía una serie de mensajes al servidor y, una vez que ha enviado todos los mensajes, el servidor responde con una única respuesta. Cuando el cliente tiene que enviar muchos datos que el servidor debe procesar juntos.

- **Bidirectional Streaming RPC:** tanto el cliente como el servidor pueden enviar una secuencia de mensajes de manera independiente, lo que permite la comunicación en tiempo real entre ambos. Los mensajes fluyen en ambas direcciones sin esperar a que el otro lado termine de enviar sus mensajes.

Pub/Sub (Publicación/Suscripción)

Es un modelo de comunicación donde los publicadores envían mensajes asíncronos a un canal, y los suscriptores a dicho canal los reciben

La mejor alternativa para este sistema es el Bidireccional Streaming RPC, debido a su capacidad de manejar múltiples comunicaciones en ambas direcciones. Si se usara dicha alternativa tanto para publicadores como suscriptores, se podría permitir que los suscriptores reciban actualizaciones en tiempo real de las publicaciones sin tener que hacer solicitudes constantes, manteniendo la conexión activa y evitando sobrecarga de peticiones. En cuanto a los publicadores, pueden enviar datos continuamente sin tener que reconectarse para cada mensaje, es útil en casos donde los publicadores generan datos de manera constante y en tiempo real. En algunos casos, el servidor podría enviar mensajes de confirmación. Sin embargo, esto no siempre es necesario, ya que los publicadores generalmente solo necesitan enviar los datos, lo que provoca que no sea muy efectivo para los publicadores, solo en ciertos casos. Por lo que una mejor opción para los publicadores sería Client Streaming RPC, por lo anteriormente mencionado

Analizando más en detalle:

- **Escalabilidad:** permite gestionar muchas conexiones al mismo tiempo de forma eficiente. Los clientes pueden recibir actualizaciones de forma continua sin tener que realizar nuevas solicitudes. Además, gRPC utiliza HTTP/2, que permite manejar varias transmisiones de datos dentro de una sola conexión, ideal para manejar una gran cantidad de suscriptores y comunicaciones simultáneas.
- **Consistencia vs Disponibilidad:** el objetivo es que el sistema esté disponible y pueda enviar mensajes continuamente, incluso si la consistencia de los datos no es asegurada. Bidirectional Streaming RPC permite que los mensajes se envíen sin interrupciones, asegurando que los clientes reciban la información rápidamente. Sin embargo, es posible lograr consistencia, siempre y cuando se garantice que todos los clientes reciban mensajes continuamente.
- **Seguridad:** gRPC tiene mecanismos para autenticar usuarios, verificar permisos y cifrar los datos transmitidos usando TLS (Transport Layer Security). Bidirectional Streaming RPC puede usar estos mecanismos para garantizar que solo los usuarios autorizados puedan enviar y recibir mensajes, asegurando la seguridad del sistema.
- **Facilidad de implementación y mantenimiento:** su implementación es compleja, ya que se necesita de una arquitectura que soporte múltiples envíos y recepciones, además de un sistema cuya lógica sea capaz de gestionarlos. Pero se compensa con la flexibilidad que ofrece gRPC para manejar datos en tiempo real. Gracias al uso de HTTP/2 el mantenimiento es sencillo, ya que permite gestionar múltiples conexiones. Además el streaming evita tener que abrir nuevas conexiones para enviar varios datos, reduciendo la carga del servidor

Sistema de Archivos FTP

Sistema basado en la transferencia de archivos, donde los clientes pueden cargar, descargar y manipular archivos en un servidor remoto.

Quizás las opciones más viables para este sistema son las de Unary RPC o Client Streaming RPC. La primera es más sencilla y garantiza consistencia en los datos ya que trata cada operación de forma independiente, mientras que la otra puede ser más útil cuando se necesitan cargar grandes volúmenes de datos. Siguiendo con esa lógica, si la invertimos descubrimos que usando Server Streaming RPC se vuelven eficientes las descargas de grandes volúmenes de datos. Sin embargo, Bidirectional Streaming RPC no es tan eficiente aunque lo parezca, ya que ambos extremos podrían enviar mensajes sin pausa, lo cual dificultará la coordinación y comprometería la fiabilidad de la transferencia

- **Escalabilidad:** las operaciones suelen ser bastante directas, como subir o descargar un archivo, y estas pueden manejarse bien con un solo envío y respuesta. Si se trata de archivos grandes, Client Streaming RPC es útil porque permite enviar el archivo en varias partes.
- **Consistencia vs Disponibilidad:** con Unary RPC cada operación se trata de forma independiente, simplificando el manejo de errores y asegurando la integridad de datos. En este sistema, la consistencia es lo más importante, asegurando que los archivos no estén corruptos o haya pérdidas de datos, incluso aunque el sistema no esté disponible siempre al 100%
- **Seguridad:** gRPC ofrece seguridad con métodos de autenticación y cifrado usando TLS. También permite el uso de mecanismos como tokens de autenticación para asegurarse de que solo las personas autorizadas puedan realizar acciones
- **Facilidad de implementación y mantenimiento:** Unary RPC permite una sencilla implementación, siguiendo un modelo sencillo de Solicitud-Respuesta que se ajusta a las operaciones básicas de FTP. El manejo de errores también es más simple respecto al streaming, lo que reduce la complejidad y mejora la gestión del sistema

3) Para resolver este inciso usamos la api **Unary RPC**, ya que es sencilla de programar y configurar. Sin embargo, no es la solución más adecuada para este chat en tiempo real, más adelante detallaremos las decisiones.

- En el archivo de Protocol Buffer definimos:
 - **Message:** Representa un mensaje del chat, incluyendo el usuario que lo envía (user), el contenido del mensaje (content) y la marca de tiempo (timeStamp).
 - **User:** Representa a un usuario en el sistema, con un ID (id) y un nombre (name).
 - **ConnectRequest/ConnectResponse:** Utilizados cuando un usuario se conecta al chat. El ConnectRequest envía la información del usuario, y el ConnectResponse devuelve un mensaje de bienvenida y la confirmación del usuario.

- DisconnectRequest/DisconnectResponse: Se utilizan para manejar la desconexión de un usuario, donde DisconnectResponse incluye un mensaje de despedida.
 - SendMessageRequest/SendMessageResponse: Envía un mensaje del usuario y confirma su envío.
 - GetMessagesRequest/GetMessagesResponse: GetMessagesRequest pide mensajes nuevos desde una marca de tiempo específica, y GetMessagesResponse devuelve una lista (repeated) de los mensajes nuevos.
 - HistoryRequest/HistoryResponse: Recupera el historial de mensajes para el usuario, devuelto en un solo campo messages.
 - Definimos además, cinco operaciones RPC (Remote Procedure Call) para el chat: Connect, Disconnect, History, SendMessage, y GetMessages, con los mensajes de solicitud y respuesta asociados.
-
- Cada mensaje hacia el servidor implica un apertura/cierre de conexión hacia el servidor, esto genera un alto overhead de mensajes cuando haya varios, inclusive un mismo cliente produce esto ya que, de forma periódica (cada 5 seg), el cliente le pide al servidor que le mande los mensajes nuevos
 - Para manejar concurrencia y múltiples envíos Unary RPC gestiona cada solicitud como una aislada, por lo que no debería haber problemas. El servidor utiliza un pool de hilos para crear un hilo por cada solicitud
 - Para almacenar mensajes y usuarios se usa un tipo de lista llamado **CopyOnWriteArrayList**, que hace una copia de la lista por cada escritura, asegurando que no se pierdan mensajes. Esto provoca un problema que, lamentablemente, va a arrastrar a todo el ejercicio, genera mucho costo cuanto más mensajes sean enviados
 - En cuanto a escalabilidad, tarde o temprano no va a ser escalable, por lo mencionado anteriormente
 - Cada vez que se envía un mensaje o hay una conexión/desconexión al chat se registra en un .txt, cuyo contenido lo devolvemos cuando los clientes hacen una llamada al proceso History.
 - Cada mensaje está conformado por el contenido, el usuario y el timestamp que fue enviado. Para visualizar los mensajes nuevos, hacemos llamado al proceso GetMessages, que realiza un filtrado en la lista de mensajes donde nos traemos aquellos cuyo timestamp sea superior a uno recibido y luego filtramos para eliminar los que sean del usuario que pidió los mensajes
 - En la ejecución del cliente se crean dos hilos, donde uno es responsable de enviar mensajes, y el otro de estar constantemente chequeando si hay nuevos mensajes, llamando al proceso GetMessages
 - Cuando el cliente se desconecta, se notifica a los demás participantes y se elimina de la lista de miembros del servidor, sin embargo todos sus mensajes se siguen manteniendo en el historial.

4) Para este experimento, planteamos los siguientes objetivos: evaluar el rendimiento del servidor frente a conexiones simultáneas de múltiples clientes, medir el tiempo de respuesta del servidor ante mensajes enviados concurrentemente y verificar la consistencia del archivo `Historial.txt` para evitar pérdida o duplicación de mensajes. El experimento

consistirá en realizar pruebas de carga mediante conexiones concurrentes al servidor en distintos escenarios. Probaremos con diferentes cantidades de clientes (por ejemplo, 1, 5, 10, 20 y 50 hilos). Implementaremos este proceso usando hilos para representar múltiples clientes conectados y enviando mensajes al mismo tiempo; cada hilo actuará como un cliente enviando un mensaje de texto al servidor. Mediremos los tiempos de respuesta promedio.

	Cantidad de Clientes				
	1	5	10	20	50
Promedio de Tiempo de respuesta (ms)	13.00	15.20	31.50	136.15	147.82

Como conclusión podríamos decir que aunque el sistema maneja una carga moderada de usuarios con buenos tiempos de respuesta, a partir de 20 conexiones concurrentes la latencia comienza a ser considerable, lo que indica que podría ser necesario replantear la estrategia de comunicación si se espera un tráfico significativamente mayor. En cuanto a la consistencia del Historial, observamos que la mantiene debido a que la cantidad de líneas escritas corresponden a las esperadas, por cada cliente le corresponden 3 líneas (conexión, mensaje, desconexión).

5) En el experimento, se midieron los tiempos de respuesta para comparar gRPC y Sockets usando mensajes de varios tamaños, desde 10 hasta 10^6 bytes. Se enviaron 10 mensajes por cada protocolo y se registró el tiempo promedio y la desviación estándar. Al comparar los tiempos de respuesta, se observa que gRPC es más lento y tiene mayor variación en sus resultados, especialmente con mensajes grandes. Aunque gRPC ofrece ventajas como una estructura más organizada y fácil de usar, su rendimiento es más bajo debido a la complejidad del protocolo que utiliza (HTTP/2). Mientras tanto, Sockets muestran tiempos más rápidos y consistentes en la mayoría de los casos, principalmente con mensajes pequeños y medianos, lo que los hace más eficientes cuando se busca velocidad. Sin embargo, los Sockets pueden ser más difíciles de implementar y no ofrecen tantas funciones como gRPC. Por lo tanto, Sockets rinde mejor para aplicaciones que requieren alta velocidad, mientras que gRPC es útil cuando se necesitan más características y facilidad de desarrollo.

Tamaño de Mensaje	Promedio con grpc (ms)	Desviación estándar con grpc
10^1	10.7	4.96
10^2	8.1	0.94
10^3	12.2	7.79
10^4	7.6	1.28

10 ⁵	9.6	1.74
10 ⁶	24.6	6.35

Tamaño de Mensaje	Promedios con sockets (ms)	Desviación Estándar con sockets
10 ¹	2.70	1.25
10 ²	2.35	0.68
10 ³	2.85	1.35
10 ⁴	2.80	1.21
10 ⁵	6.10	1.37

Comparacion de Promedios y Desviaciones

— Promedio con grpc (ms)
 — Desviacion estandar con grpc
 — Promedios con sockets (ms)
 — Desviación Estándar con sockets

