



PROGRAMACIÓN DISTRIBUIDA Y TIEMPO REAL

T R A B A J O P R Á C T I C O N ° 3

Juan Cruz Cassera Botta 17072/7
Lucio Bianchi Pradas 19341/8

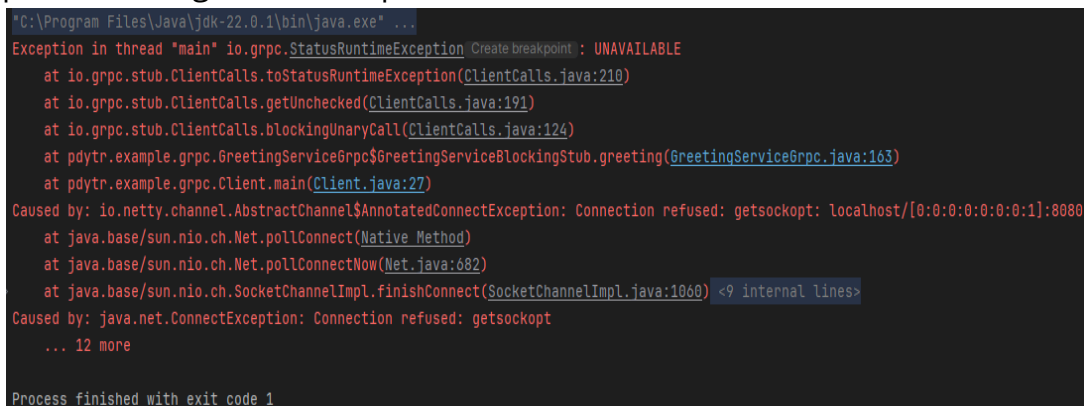


1) Manejo de Errores de Conectividad

Usando como base el programa **ejemplo 1** de gRPC, realice los siguientes experimentos para simular y observar fallos de conectividad tanto del lado del cliente como del servidor:

a) Introduzca cambios mínimos, como la inclusión de `exit()`, para provocar situaciones donde no se reciban comunicaciones o no haya un receptor disponible. Agregar screenshots de los errores encontrados.

1. Cuando corremos App.java y no corremos Cliente.java, el Servidor se queda esperando indefinidamente, debido a que está configurado de esta forma (`server.awaitTermination()`).
Técnicamente, no se produce ninguna excepción.
2. Cuando corremos Cliente.java pero no corremos App.java, se produce la siguiente excepción:



```
"C:\Program Files\Java\jdk-22.0.1\bin\java.exe" ...
Exception in thread "main" 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$GreetingServiceBlockingStub.greeting(GreetingServiceGrpc.java:163)
    at pdytr.example.grpc.Client.main(Client.java:27)
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused: getsockopt: localhost/[0:0:0:0:0:0:1]:8080
    at java.base/sun.nio.ch.Net.pollConnect(Native Method)
    at java.base/sun.nio.ch.Net.pollConnectNow(Net.java:682)
    at java.base/sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:1060) <9 internal lines>
Caused by: java.net.ConnectException: Connection refused: getsockopt
    ... 12 more

Process finished with exit code 1
```

b) Configure un DEADLINE y modifique el código (agregando, por ejemplo, la función `sleep()`) para provocar la excepción correspondiente. Agregar screenshots de los errores encontrados.

Para configurar el Deadline agregamos la siguiente instrucción en el Cliente.java luego de crear el stub:

```
stub = stub.withDeadlineAfter(5, TimeUnit.SECONDS);
```

Y para lograr provocar la excepción agregamos un `sleep` dentro de la implementación del `GreetingService`:

```
Thread.sleep(10000);
```

La excepción obtenida fue la siguiente:

```
*C:\Program Files\Java\jdk-22.0.1\bin\java.exe" ...  
Exception in thread "main" io.grpc.StatusRuntimeException: Create breakpoint : DEADLINE_EXCEEDED: deadline exceeded after 4975218400ns  
    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:32)  
  
Process finished with exit code 1
```

2) Análisis de APIs en gRPC

Describa y analice los distintos tipos de APIs que ofrece gRPC.

gRPC ofrece 4 tipos distintos de APIs:

1. API de solicitud-respuesta unaria (Unary RPC):

El cliente envía un mensaje con datos y el servidor responde con un **único** resultado.

2. API de respuesta en flujo (Server-streaming RPC):

El cliente envía una solicitud al servidor, pero en lugar de recibir una única respuesta, el servidor envía una **secuencia** (flujo) de respuestas.

3. API de solicitud en flujo (Client-streaming RPC):

El cliente envía una **secuencia de mensajes** al servidor, y una vez que envió todos ellos, el servidor procesa la información y le responde con un **único** mensaje.

4. API de bidireccional en flujo (Bidirectional-streaming RPC):

Tanto el cliente como el servidor envían y reciben **múltiples mensajes en ambos sentidos utilizando flujos**. Esto permite una comunicación fluida en tiempo real, como si fuera una sesión de chat.

Con base en el análisis, elabore una conclusión sobre cuál sería la mejor opción para los siguientes escenarios:

Nota: Desarrolle una conclusión fundamentada considerando los siguientes aspectos para ambos escenarios (pub/sub y FTP):

- **Escalabilidad:** ¿Cómo se comporta cada API en situaciones con múltiples clientes y conexiones simultáneas?
- **Consistencia vs. Disponibilidad:** ¿Qué importancia tiene mantener la consistencia de los datos frente a la disponibilidad del sistema?
- **Seguridad:** ¿Qué mecanismos de autenticación, autorización y cifrado se deben utilizar para proteger los datos y las

comunicaciones?

• **Facilidad de implementación y mantenimiento:** ¿Qué tan fácil es implementar y mantener la solución para cada API?

a) Un sistema de pub/sub: Server-streaming RPC

- En un sistema pub/sub, tenemos N clientes (suscriptores) y un servidor (publicador). El publicador envía actualizaciones o mensajes a múltiples suscriptores.
- Para este sistema, sería óptimo usar Server-streaming RPC, ya que se ajusta muy bien a su definición: nos permite que el servidor envíe un flujo de mensajes a cada cliente en respuesta a una única solicitud de cada cliente (su suscripción).
- **Escalabilidad:** es altamente escalable porque el servidor puede manejar múltiples suscriptores enviando flujos de datos a cada cliente. Para manejar más clientes simplemente deberían suscribirse. En caso de que el servidor no pueda manejar los clientes, se podría escalar vertical (agregando más recursos de hardware) u horizontalmente (agregando instancias).
- **Consistencia vs. Disponibilidad:** en un sistema pub/sub, la consistencia es menos crítica que la disponibilidad, ya que los suscriptores suelen aceptar recibir actualizaciones con cierta latencia o en desorden. Server-streaming es ideal en este caso porque asegura que los clientes reciban las actualizaciones, pero no garantiza que todos reciban la información exactamente al mismo tiempo, cosa que no es crítica.
- **Seguridad:** gRPC soporta TLS para cifrar la comunicación, lo que garantiza la confidencialidad. Se puede implementar autenticación con tokens (JSON Web Token) para que solo los clientes autorizados puedan suscribirse a los flujos de datos.
- **Facilidad de implementación y mantenimiento:** relativamente sencillo y sin necesidad de mucho mantenimiento.

b) Un Sistema de archivos FTP: Bidirectional Streaming RPC

- Un sistema de archivos FTP implica transferencias de archivos en ambas direcciones (subida y descarga) y posiblemente comandos de control.
- Bidirectional Streaming RPC es ideal en este caso porque:
 1. Para subir archivos, el cliente puede enviar un flujo de datos

al servidor.

2. Para descargar archivos, el servidor puede enviar un flujo de datos al cliente.

1. **Escalabilidad:** es más difícil de escalar que el caso anterior, ya que se deben poder manejar conexiones con múltiples clientes, que pueden implicar subida y descarga de archivos de gran tamaño. Implica un gran uso de almacenamiento por parte del servidor
2. **Consistencia vs. Disponibilidad:** en este tipo de sistema, tanto la consistencia de los datos como la disponibilidad de los mismos son ambos críticos. Los archivos que el servidor posee almacenados no deberían corromperse y ser seguros para que los clientes los descarguen, y los clientes deberían ser capaces de subir o descargar archivos en el momento que quieran, sobre todo si el sistema trata con archivos sensibles.
3. **Seguridad:** como se están transfiriendo archivos potencialmente sensibles, es esencial que las transferencias estén cifradas. Se podría cifrar la comunicación utilizando TLS. También es esencial asegurarse de que solo los usuarios autorizados puedan iniciar sesión y acceder a los archivos.
4. **Facilidad de implementación y mantenimiento:** un sistema de estas características sería complejo tanto de implementar como de mantener:
 - La implementación sería compleja debido a todas las cosas que se deben tener en cuenta: comunicación bidireccional, seguridad (cifrado y autenticación), infraestructura de almacenamiento, concurrencia a la hora de tener múltiples clientes subiendo archivos a la vez, etc.
 - Almacenar un alto volumen de archivos en el servidor requeriría constante mantenimiento de muchos discos duros, limpieza de los mismos, reemplazo de los que dejan de funcionar, etc.

3) Desarrollo de un Chat Grupal utilizando gRPC

Implemente un sistema de chat grupal simplificado utilizando gRPC, que permita la interacción entre múltiples clientes y un servidor central. El sistema debe incluir las siguientes funcionalidades:

1. Conectar: Permite a un cliente unirse al chat grupal.
 - Entrada: Nombre del cliente.
 - Salida: Confirmación de la conexión (del cliente y mensaje de bienvenida).
2. Desconectar: Maneja tanto la desconexión voluntaria de un cliente como la desconexión involuntaria (por ejemplo, mediante el uso de Ctrl-C o por pérdida de conexión).
 - Entrada: Nombre del cliente que se desconecta.
 - Salida: Confirmación de la desconexión y mensaje de despedida.
3. Enviar Mensaje: Permite a un cliente enviar un mensaje al servidor, que se encargará de retransmitirlo a todos los demás clientes conectados al chat.
 - Entrada: Nombre del cliente y contenido del mensaje.
 - Salida: Confirmación de envío y distribución del mensaje a todos los clientes conectados.
4. Historial de Mensajes: Cualquier cliente puede solicitar el historial completo de mensajes intercambiados en el chat mediante el comando especial /historial.
 - Entrada: Comando /historial.
 - Salida: Archivo de texto o PDF con el historial de mensajes (marcas de tiempo, nombres de los clientes y mensajes).

[20240428- 12:16:53] Cliente 1: Buen día grupo

[20240428- 12:18:55] Cliente 2: Buen día, ¿realizaron el punto 3 del tp3 de distribuida?

[20240428- 12:19:30] Cliente 3: Si, el punto está resuelto.

Requerimientos de Implementación:

- Servidor:
 - El servidor debe ser capaz de manejar múltiples clientes concurrentemente, permitiendo la comunicación simultánea entre ellos.
 - El servidor debe llevar un registro actualizado de los clientes conectados, eliminando a aquellos que se desconecten de manera

voluntaria o involuntaria.

- El servidor debe mantener un archivo de historial de mensajes que registre todas las conversaciones, para permitir la consulta posterior.
- Documente todas las decisiones tomadas durante el proceso de diseño e implementación del servidor, justificando los enfoques utilizados para la concurrencia y la gestión de clientes.

- Clientes:

- Implemente al menos tres clientes que se conecten al servidor, envíen mensajes y reciban los mensajes de otros usuarios.
- Los clientes deben manejar de manera adecuada las excepciones relacionadas con la desconexión involuntaria o fallos en la conectividad.

- Concurrente y Escalable: El sistema debe permitir que varias instancias de clientes se conecten y envíen mensajes de forma concurrente sin que el rendimiento se vea afectado.

Nota: Para cada funcionalidad desarrollada, el código debe estar debidamente comentado y explicado en el informe final. El informe debe detallar las decisiones clave tomadas en el diseño y construcción del sistema, así como cualquier problema encontrado y cómo fue resuelto.

Documentación del proceso de diseño:

Dado que grpc está implementado en múltiples lenguajes, decidimos realizar el chat grupal en Python debido a que ofrece una sintaxis más clara y el código era sustancialmente más corto que en Java.

El .proto que definimos posee los siguientes mensajes:

```
message Vacio {}

message Mensaje {
    string nombre = 1;
    string contenido = 2;
    string timestamp = 3;
}

message Confirmacion {
    string mensaje = 1;
}

message HistorialResponse {
```



```

        repeated Mensaje mensajes = 1;
    }

    message NombreCliente {
        string nombre = 1;
    }

```

y los siguientes servicios que hacen uso de esos mensajes:

```

rpc Conectar (NombreCliente) returns (Confirmacion);
rpc Desconectar (NombreCliente) returns (Confirmacion);
rpc EnviarMensaje (Mensaje) returns (Confirmacion);
rpc SolicitarHistorial (Vacio) returns (HistorialResponse);
rpc ChatStream (Vacio) returns (stream Mensaje);

```

Se implementaron todos los servicios pedidos en el enunciado, como Conectar, Desconectar, Enviar Mensaje y Solicitar Historial. Además, tuvimos que implementar un quinto mensaje, Chat Stream, para manejar la lógica del chat grupal. En este caso Chat Stream cumple el rol de un Server Stream, ya que el cliente envía una única solicitud al servidor, pero en lugar de una única respuesta, el servidor le envía una serie de respuestas o un flujo de datos, lo que permite recibir los mensajes enviados por los demás clientes progresivamente, sin volver a solicitarlos.

Para permitir que el cliente pueda recibir los mensajes de los demás usuarios conectados al chat grupal, y al mismo tiempo pueda enviar nuevos mensajes, tuvimos que recurrir al uso de 2 hilos:

1. El primer hilo es el que se encarga de escuchar mensajes constantemente del stream e imprimirlos en consola
2. El segundo hilo se encuentra en un loop que permite enviar mensajes por consola, además de utilizar los comandos `/salir` e `/historial`

Manejo de concurrencia / escalabilidad: Para lograr que el servidor sea capaz de manejar múltiples conectados simultáneamente:

- Se usó un `ThreadPoolExecutor` en el servidor de gRPC para manejar la concurrencia, permitiendo manejar varios hilos a la vez y distribuyendo la carga entre ellos. Para el código utilizamos un `ThreadPoolExecutor` con 3 workers
- Cada cliente se ejecuta en un hilo separado en su máquina local, y se utiliza otro hilo adicional para escuchar mensajes entrantes en paralelo a

la posibilidad de enviar mensajes.

Manejo de clientes: Decidimos que únicamente se puedan crear usuarios con nombres distintos. En caso de ingresar un nombre de usuario que ya está en uso en el chat, se pide repetidamente un nuevo nombre de usuario hasta que éste sea válido.

Además, para gestionar la desconexión, se decidieron dos enfoques:

- Voluntaria: Cuando un cliente envía un comando `/salir`, el servidor recibe el mensaje y lo elimina de la lista de clientes conectados. Acto seguido, ese cliente termina su ejecución.
- Involuntaria: El servidor detecta cuando un cliente se desconecta inesperadamente (por pérdida de conexión o cierre forzado de la aplicación), eliminándolo automáticamente de la lista.

Persistencia del Historial de Mensajes: Se decidió que el historial se escriba periódicamente a un archivo .txt en el sistema del servidor para persistencia a largo plazo. Este archivo puede ser consultado por cualquier cliente mediante el comando `/historial`. El comando descarga un pdf, que es único para cada uno de los usuarios.

4) Análisis de Concurrencia y Eficiencia

Después de implementar el sistema de chat grupal, realice un análisis sobre la concurrencia y eficiencia del servidor:

- **Concurrencia:** Diseñe un experimento para demostrar si el servidor es capaz de manejar múltiples solicitudes de clientes de manera concurrente. Esto incluye evaluar el comportamiento del servidor cuando varios clientes envían mensajes al mismo tiempo. Si se encuentran problemas de concurrencia, proponga soluciones y documente cómo se podrían aplicar.

Nota: El análisis de concurrencia y eficiencia debe incluir gráficos o tablas que muestren los resultados de las mediciones, junto con una interpretación de los mismos en el contexto del sistema de chat grupal.

Para realizar el experimento decidimos crear 10 clientes distintos, cada uno enviando 20 mensajes cada 0.1 segundos. El problema principal de concurrencia que puede ocurrir es que, como no se está protegiendo a las dos variables compartidas del servidor (la lista de chats y la lista de clientes conectados), varios clientes pueden estar diciéndole al servidor que edite estas listas y también el archivo .txt del historial a la vez, lo cual haría que se produzcan condiciones de carrera y se pisen valores.

Esto se puede evitar bloqueando el acceso al recurso compartido cuando éste se esté por usar en el Servidor, y hasta que se haya terminado de usar en ese momento.

Lo anterior se puede implementar utilizando variables mutex, las cuales en el caso de python están dadas por los locks de la librería threading.

```
import time
import threading
from cliente import Client

def enviar_mensajes_automaticos(cliente):
    for i in range(20): # Enviar 20 mensajes por cliente
        cliente.enviar_mensaje(f"Mensaje {i} del usuario {cliente.usuario}")
        print(f"Enviando: {mensaje}")
        time.sleep(0.1) # Intervalo de 0.1 segundos entre cada mensaje

if __name__ == "__main__":
    usuarios = [f"usuario_{i}" for i in range(10)] # Crear 10 clientes
    for usuario in usuarios:
        print(f"Creando cliente para: {usuario}")
        cliente = Client(usuario)
        threading.Thread(target=enviar_mensajes_automaticos, args=(cliente,)).start()
```

5) Medición de Tiempos de Respuesta

a) Diseñe un experimento que permita medir el tiempo de respuesta mínimo de una invocación en gRPC. Calcule el promedio y la desviación estándar.

b) Utilizando los datos obtenidos en la Práctica 1 (Sockets), realice un análisis comparativo de los tiempos de respuesta. Elabore una conclusión sobre los beneficios y complicaciones de cada herramienta.

Datos obtenidos en la Práctica 1 (Sockets):

Cantidad de bytes comunicación	Java (tiempo en milisegundos)	C (tiempo en milisegundos)
10^5	2,9031000	0,095
10^6	13,3698300	0,798

Decidimos realizar un experimento que calcula el tiempo mínimo, el tiempo promedio y la desviación estándar de una comunicación en gRPC al enviar 10^5 y 10^6 bytes. Luego de ejecutarlo obtuvimos los siguientes resultados:

```
Experimento enviando 10^5 bytes

Tiempo mínimo: 0.000000 ms
Tiempo promedio: 0.486823 ms
Desviación estándar: 0.527870 ms

Experimento enviando 10^6 bytes

Tiempo mínimo: 1.504421 ms
Tiempo promedio: 2.648798 ms
Desviación estándar: 0.538859 ms
```

Debido a que el experimento fue resuelto en Python, decidimos compararlo con los tiempos obtenidos en Java. Aún así, los tiempos de gRPC en Python son significativamente más bajos que los obtenidos con sockets en Java, incluso considerando que Python suele ser más lento que lenguajes compilados como Java y C.

Comparación de ambos resultados:

Cantidad de bytes comunicación	Java Sockets (tiempo en milisegundos)	Python gRPC (tiempo en milisegundos)
10^5	2,9031000	0,486823
10^6	13,3698300	2,648798

En el experimento con 10^6 bytes, los tiempos de Java aumentaron considerablemente (hasta 13 ms), mientras que gRPC en Python mostró solo un pequeño incremento (de 0.48 ms a 0.798 ms). Esto sugiere que gRPC maneja mejor la sobrecarga cuando el tamaño de los datos cambia, mientras que los sockets en Java pueden volverse menos eficientes a medida que aumentan los datos.

Podemos concluir que gRPC resulta más rápido que Sockets, esto puede ser debido a que gRPC utiliza protocolos optimizados como HTTP/2 que ofrecen multiplexación y compresión de encabezados, lo cual ayuda a reducir el tiempo de transmisión para pequeños volúmenes de datos.

En cuanto a las complicaciones, creemos que gRPC es más cómodo y legible de usar que Sockets, que resultaba muy verboso y tedioso por ser tan bajo nivel. gRPC ofrece abstracciones útiles que ayudan a agilizar el desarrollo.