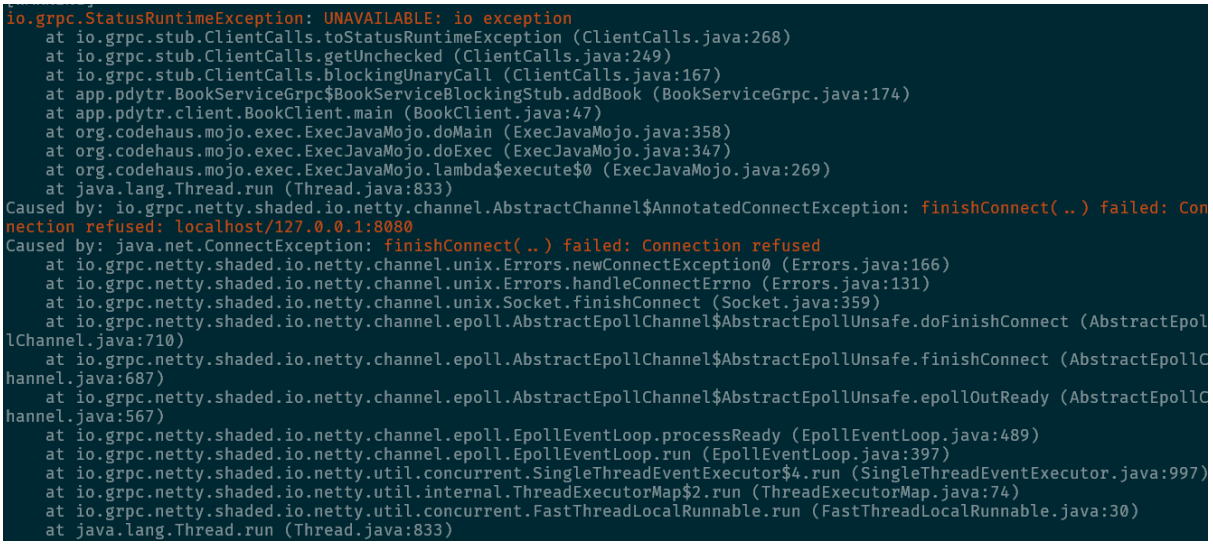
Programación Distribuida y Tiempo Real
Trabajo Práctico 3

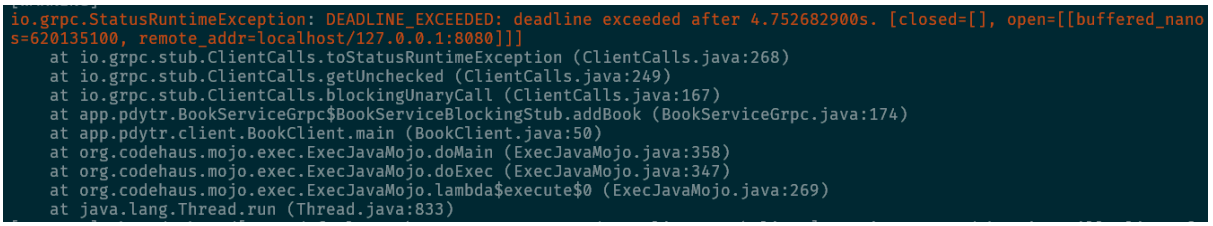
- Joaquin Tartaruga (17023/7)

- 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. Ahora bien, del lado del servidor podemos agregar un exit() antes del start() para que no reciba mensajes del cliente o terminar el servidor una vez el cliente se conecta, produciendo el siguiente error:



B) En esta seccion lo que hicimos fue agregar un sleep de 10 seg en el servidor antes de contestar el mensaje y un tiempo de espera del lado del cliente de 5 seg, provocando que el servidor nunca conteste el mensaje a tiempo (generando un deadline)



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, analizando mas 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 implementacion y mantenimiento:** su implementación es compleja, ya que se necesita de una arquitectura que soporte múltiples envios 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

- **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 implementacion 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 tambien es mas 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.

- 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 envios 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 mas 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, que luego se va a usar para devolver el historial
- Cada mensaje esta conformado por el contenido, el usuario y el timestamp que fue enviado. Para visualizar los mensajes nuevos hacemos 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 pidio 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

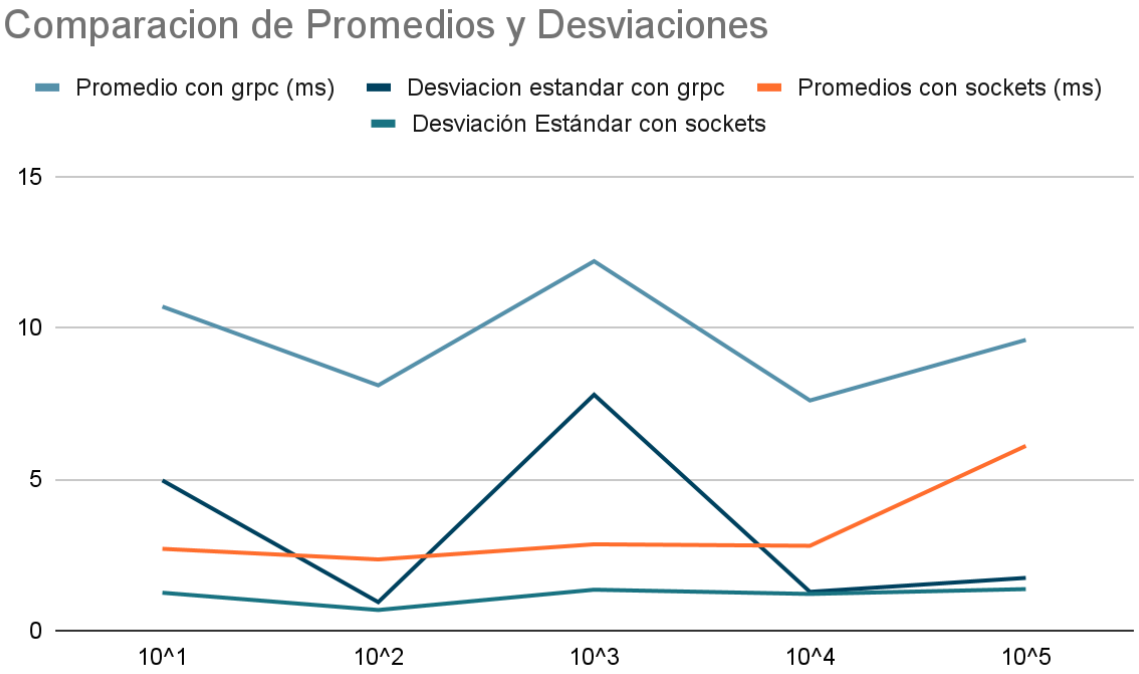
4) Para este experimento, hicimos que cinco hilos se conectaran al servidor y enviaran mensajes simultáneamente, con el objetivo de observar cómo reacciona el servidor ante múltiples mensajes recibidos al mismo tiempo. Según nuestras observaciones, no se detectaron anomalías; todos los mensajes se registraron correctamente y el archivo .txt que guarda el historial del chat se mantuvo sin inconsistencias, preservando la información de manera ordenada.

5) Al comparar los tiempos de respuesta entre gRPC y Sockets, se ve queg RPC es más lento y tiene mayor variación en sus resultados, sobre todo con mensajes grandes. Aunque gRPC ofrece ventajas como una estructura más organizada y fácil de usar, su rendimiento sufre debido a la complejidad del protocolo que utiliza. Por el contrario, Sockets muestran

tiempos más rápidos y consistentes en la mayoría de los casos, especialmente con mensajes pequeños y medianos, lo que los hace más eficientes cuando se busca velocidad. Sin embargo, Sockets pueden ser más difíciles de implementar y no ofrecen tantas funciones como gRPC. En resumen, Sockets rinden 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
15
10
5
0
10^1
10^2
10^3
10^4
10^5