Programacion distribuida y tiempo real

Alumnos:

- Felipe Dioguardi - 16211/4
- Julian Marques de Abrantes - 15966/0

PRACTICA 3

1. **Utilizando como base el programa ejemplo1 de gRPC:**

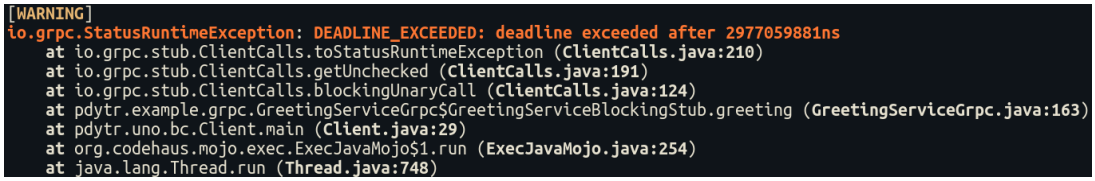
   Mostrar experimentos donde se produzcan errores de conectividad del lado del cliente y del lado del servidor.

   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.**

   - El primer experimento consistió simplemente en correr el cliente antes que el servidor. El resultado fue un `io.grpc.StatusRuntimeException: UNAVAILABLE`, causado por un `Connection refused: localhost/127.0.0.1:8080`.
   - En el segundo experimento se agregó la línea `System.exit(125)` al mensaje `greeting()` de la implementación del servicio. Esto hizo que cuando un cliente se quisiera conectar con él, el servidor se cierre. En el cliente se levantó la excepción `io.grpc.StatusRuntimeException: UNAVAILABLE: Network closed for unknown reason`.
   - Finalmente, en el tercer experimento, se agregó un thread del lado del cliente que haga un `System.exit()` luego de 0.35 segundos. De esta forma, tiene tiempo de enviar el mensaje al servidor, finalizar su propia ejecución, y así hacer que la respuesta sea enviada a un cliente terminado. Luego de varias ejecuciones (no siempre funciona debido a la naturaleza concurrente del experimento) vimos que aun cuando el cliente se cierra luego de enviar el pedido, el servicio envía su respuesta sin problemas. Ambas partes salen sin producir excepciones.

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

Se modificó el código tal como se pide en el enunciado, poniendo un `sleep()` en el servidor que excediese el deadline definido. La excepción correspondiente fue un `DEADLINE_EXCEEDED` del lado del cliente.



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

Luego de establecer un deadline en el cliente de 3000 milisegundos, y un `sleep()` en el servidor de 2500 milisegundos, verificamos que hay ocasiones en las que el servidor logra responder al pedido, mientras que otras tarda demasiado tiempo en el cómputo, y se excede el deadline. Un patrón particular que notamos (en un único ambiente de ejecución) es que con esos valores de `sleep()` y deadline, el error siempre se produjo en la primera consulta recibida por el servidor. Según parece, es en este momento en el que más cómputo adicional realiza. Como reiniciamos 4 veces el servidor, tuvimos cuatro “primeras consultas”, por lo que contamos 4 errores sobre 10. Las otras 6 consultas fueron posteriores a las fallidas, y todas se completaron exitosamente.

Al poner un deadline de 3000 milisegundos y un `sleep()` de 3000 milisegundos, las consultas siempre fallaron. Lo mismo sucedió con cada prueba con un `sleep()` mayor al deadline.

2. **Describir y analizar los tipos de API que tiene gRPC. Desarrolle una conclusión**
acerca de cuál es la mejor opción para los siguientes escenarios:

a. **Un sistema de pub/sub.**

b. **Un sistema de archivos FTP.**

c. **Un sistema de chat.**

gRPC posee las siguietnes tipos de api:

- Unary RPC: Método más simple de RPC, donde el cliente manda un solo request y recibe una sola respuesta.
- Server streaming RPC: Similar al unary, pero el servidor manda un stream de mensajes al cliente como respuesta. Por último, el status y la metadata del servidor son enviados al cliente.
- Client streaming RPC: Similar al unary, pero el cliente manda un stream de mensajes al servidor en lugar de un solo request. El servidor envía un solo mensaje en respuesta junto a su estado y metadata.
- Bidirectional streaming: En el bidirectional streaming RPC, la llamada es iniciada por el cliente al invocar el método y el server recibe la metadata, nombre del método y deadline. El server puede elegir si esperar un streaming de mensajes del cliente o reenviar la metadata inicial.

a. **Un sistema de pub/sub:** en este tipo de sistemas el servidor se ve como un repositorio de pequeños recursos, y el cliente como un agente que solicita estos recursos al cliente. Como en cada solicitud se pide un recurso, la API unary resulta la más apropiada.

b. **Un sistema de archivos FTP:** en este escenario tuvimos varias consideraciones. Determinamos que el mejor tipo de API gRPC es el unary RPC. Teniendo en cuenta que un FTP consiste en el pedido y envió de un archivo, la comunicación consiste en 1 request de lectura/escritura de un archivo por parte del cliente y una única respuesta por parte del servidor. Como gRPC tiene la capacidad de enviar archivos bastante grandes en un solo mensaje, no siempre resulta necesario implementar un algoritmo con una ventana que parta el contenido en varios envíos. Aun así, si el programador lo prefiriera, esto sigue siendo una posibilidad utilizando la API unary. Otra posibilidad, que cede un poco más de control a gRPC, es la de usar un server streaming para la operación de lectura. Esto sería útil en caso de que se quiera enviar el archivo leído en varias partes (para que el servidor lo haga de forma estándar). Con la misma lógica podría usarse un client streaming para la operación de escritura.

c. **Un sistema de chat:** Considerando que el chat es un cliente y un servidor enviando mensajes constantemente sin necesidad de una respuesta, decidimos que la mejor API es la bidirectional streaming. Este tipo de API permite a un cliente enviar tantos mensajes como desee y permite que el

servidor vaya respondiendo esos mensajes cuando lo desee y en la cantidad que desee sin que estos se pisen entre sí.

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.**

   Tanto el servidor como el cliente utilizan protocolos de transmisión de datos estándar y una interfaz común, definidas en el archivo con extensión `proto`. En consecuencia, el cliente sabe desde el momento de la compilación cuáles son las peticiones que puede realizar al servicio, conociendo la firma exacta de los mensajes. Esto significa que el cliente conoce exactamente, desde el momento de la compilación, la cantidad y el tipo de los parámetros que conforman cada mensaje a enviar al servicio, así como también conoce la estructura de la respuesta que se le devolverá.

   Entonces, como ambos conocen como (des-)serializar los mensajes de su contraparte gracias a esta interfaz compartida, puede decirse que gRPC es transparente tanto para el manejo de parámetros como para el de los valores de retorno.

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.**

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

      Primero se definió en el archivo `FtpService.proto` la interfaz a implementar por el servicio `FtpService`, que actuaría como servidor. El mismo cuenta con los métodos `read()` y `write()`, y ambos implementan la API unary (como ya se mencionó anteriormente, provee una forma segura de enviar grandes

cantidades de datos, y permite implementaciones sencillas con y sin el uso de ventanas definidas por el programador).

Definimos 4 mensajes:

- El `ReadRequest` para realizar el pedido de lectura. Encapsula un campo string para simbolizar el nombre del archivo, un offset de tipo entero para indicar una posición desde la cual iniciar a leer, y otro entero para la cantidad de bytes a leer.
- El `WriteRequest` para realizar el pedido de escritura. Contiene un string para indicar el nombre del archivo, un entero para definir la cantidad de bytes a escribir, y un `ByteString` para enviar los datos.
- El `ReadResponse` para retornar el resultado de la lectura, con un campo `ByteString` que contiene los datos, y un campo entero con la cantidad de bytes leídos.
- El `WriteResponse` para retornar el resultado de la escritura, con un único campo entero representando la cantidad de datos escritos.

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).**

gRPC soporta ejecución concurrente de múltiples RPCs. Cuando un evento RPC llega, se encola en el `serverBuilder.executor()` determinado durante la construcción de servidor. En caso de no especificar ninguno, se utiliza uno por defecto basándose en la cantidad de hilos necesarios para la ejecución.

De esta manera, es posible que un mismo servicio mantenga conexiones con varios clientes al mismo tiempo, aunque cada método de su interfaz pública sea “atómico” (por ejemplo, varios clientes pueden llamar varias veces al método `write()` de un servidor FTP, pero la computación de cada llamada se hará secuencialmente).

Nuevamente, como en la practica anterior, la concurrencia no es apropiada ya que al no ser una instrucción atómica, los llamados se pisan, produciendo así una mezcla de los datos cuando dos clientes intentan escribir sobre el mismo archivo. Como posible solución, debería crearse una carpeta para que cada cliente pueda guardar allí sus archivos y evitar esa mezcla de información, tal y como se propuso en la entrega anterior.

Como experimento diseñamos dos clientes que envian en simultaneo dos archivos, "a.txt" y "b.txt" y lo guardan en el servidor en un archivo "copy.txt". Este archivo resulta en una mezcla de ambos archivos "a.txt" y "b.txt".

Nota: diseñe un experimento con el que se pueda demostrar fehacientemente que dos o más invocaciones remotas se ejecutan concurrentemente o no.

5. **Tiempos de respuesta de una invocación**

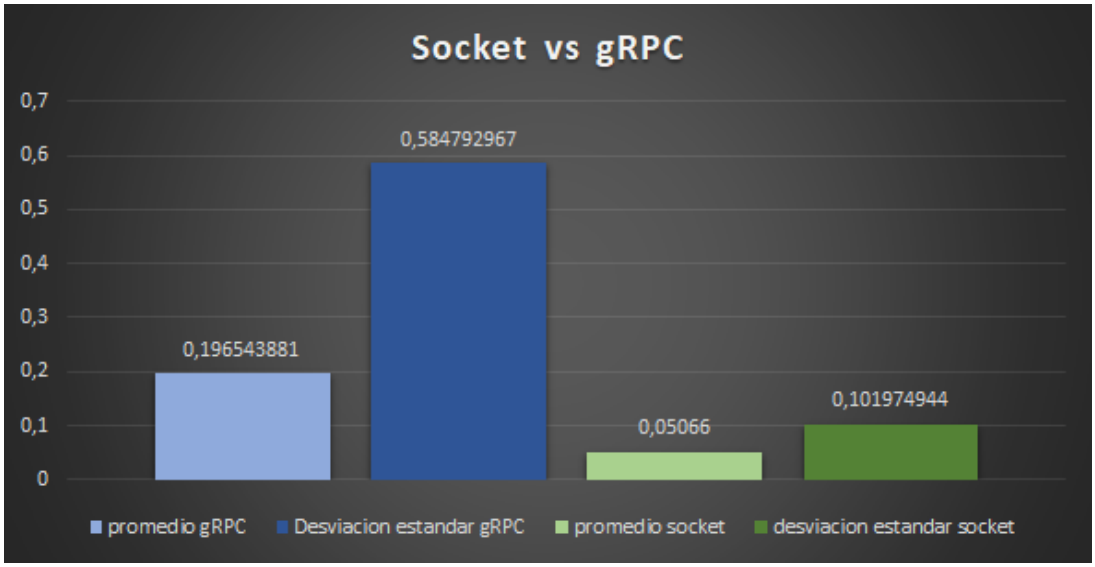
   a. **Diseñe un experimento que muestre el tiempo de respuesta mínimo de una invocación con gRPC. Muestre promedio y desviación estándar de tiempo respuesta.**

   Para el experimento desarrollamos un servidor y un cliente nuevos, el servidor posee un método que recibe y retorna mensaje vacío, y el cliente realiza una única llamada a ese método para evaluar el tiempo mínimo de respuesta. Los resultados (expresados en milisegundos) fueron los siguientes:

   - El tiempo mínimo de comunicación fue de 0,080945

   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.**

Para realizar la comparación de tiempos, agregamos al servidor un método que recibe y devuelve un mensaje con un arreglo de bytes. Modificamos al cliente para que envíe un arreglo de 1000 bytes al servidor gRPC (la misma cantidad de datos con la que habíamos evaluado el rendimiento de los Sockets en C). Los resultados fueron los siguientes:



Puede verse a partir de estos resultados como el tiempo promedio de respuesta de los Sockets en C es mucho menor al tiempo mínimo de respuesta de gRPC. Lo mismo sucede con la desviación estándar.

▼ Ventajas gRPC

- Una de las principales ventajas de gRPC su transparencia. Esta permite que el código sea sencillo, y agrega extensibilidad a la arquitectura, permitiendo brindar soporte y compatibilidad para diferentes lenguajes. El uso de las interfaces de .proto, facilita también la utilización de la herramienta haciendo más fácil la generación del código.
- Otra ventaja es que el envío de los datos es eficiente, ya que estos se realizan de forma binaria.
- gRPC provee diferentes APIs que permiten transmitir datos de diversas maneras, todo gracias a la utilización de HTTP2.

▼ Desventajas gRPC

- La principal desventaja radica en las complicaciones para poder utilizar la herramienta. Tanto como por el uso del manejador de

paquetes Maven, la utilización del archivo POM que contiene las dependencias y los problemas a la hora de compilar el código, hacen que gRPC sea complicado de empezar a utilizar.

- Otra de las desventajas de gRPC es que depende de una red estable y potente y de que la red, el tráfico de datos, los protocolos de transmisión y el cliente y el servidor no se conviertan en presa fácil de los hackers. Además, no es compatible con el multicasting por lo que no es posible enviar mensajes a múltiples clientes en simultaneo.
- Por último, como los datos enviados mediante gRPC no poseen un formato legible por humanos, debugging de las aplicaciones se vuelve costoso.

▼ Ventajas Socket

- La principal ventaja de la utilización de Sockets en java es su sencillez. Simplemente se programan el cliente y el servidor y ya pueden enviarse mensajes.
- Otra ventaja es que son compatibles con casi todos los lenguajes de programación y está disponible en todos los sistemas operativos.
- Además, esta herramienta genera poco tráfico en la red.

▼ Desventajas Socket

- Una desventaja es que no es completamente transparente.
- Únicamente soporta el envío de datos mediante bytes, por lo que tanto el cliente como el servidor necesitan mecanismos para poder interpretarlos.
- Es difícil reutilizar código debido a que cada aplicación tiene su propia forma de interpretar los datos y de transmitir los mismos.