UNIVERSIDAD
NACIONAL
DE LA PLATA

Práctica nro. 3

Docentes: Fernando Tinetti, Gastón Marón, Alan Castelli, Julián Delekta.

Materia: Programación Distribuida y Tiempo Real.

Estudiantes: Juliana Delle Ville, Marianela Diaz Catán.

Índice

1) Utilizando como base el programa ejemplo 1 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.

Del lado del servidor:

- Exit antes de la línea de server.awaitTermination(): En este caso el servidor termina inmediatamente, mientras tanto el cliente se intenta conectar y al no haber un server corriendo genera el error: Connection refused

Del lado del cliente:

- Petición sin proceso server activo, genera error: Connection refused
- Exit antes de la línea channel.shutdownNow(): En este caso el cliente no cierra la conexión pero termina. Del lado del servidor se puede ver que llegan los datos pero ocurre un error de connection reset.
- Exit antes de realizar la petición al stub: En este caso el cliente termina sin haber realizado el request al servidor. Del lado del servidor no hay datos.

ERROR: Connection refused

```
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.1.0:java (default-cli) on project grpc-
llo-server: An exception occurred while executing the Java class. UNAVAILABLE: Connection refused: no furt
r information: localhost/[0:0:0:0:0:0:0:1]:8080 -> [Help 1]
[ERROR]
```

ERROR: Connection reset

```
oct. 22, 2023 4:35:46 P.áM. io.grpc.netty.NettyServerTransport notifyTerminated
INFO: Transport failed
java.net.SocketException: Connection reset
        at java.base/sun.nio.ch.SocketChannelImpl.throwConnectionReset(SocketChannelImpl.java:394)
        at java.base/sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:426)
        at io.netty.buffer.PooledUnsafeDirectByteBuf.setBytes(PooledUnsafeDirectByteBuf.java:288)
        at io.netty.buffer.AbstractByteBuf.writeBytes(AbstractByteBuf.java:1106)
        at io.netty.channel.socket.nio.NioSocketChannel.doReadBytes(NioSocketChannel.java:343)
        at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:123)
        at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:645)
        at io.netty.channel.nio.NioEventLoop.processSelectedKeysPlain(NioEventLoop.java:545)
        at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:499)
        at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:459)
        at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:858)
        at io.netty.util.concurrent.DefaultThreadFactory$DefaultRunnableDecorator.run(DefaultThreadFactory.java:138)
        at java.base/java.lang.Thread.run(Thread.java:833)
```

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

Para esta prueba se agregaron las siguientes modificaciones:

1. En la clase **Client.java** se creó una instancia de la clase deadline mediante la cual se invocó al método after() de dicha clase, pasándole como parámetro el tiempo de expiración, el cual se estableció en 1.
2. En la clase **GreatingServiceImpl.java** se invocó al método sleep() pasándole como parámetro el tiempo, el cual se estableció en 1000 milisegundos (1 segundo).

Como resultado se disparó la siguiente excepción:

```
io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline exceeded after
```

```
966909300ns
    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:30)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:279)
    at java.lang.Thread.run (Thread.java:833)
```

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

Para realizar este experimento se estableció el límite en el tiempo del deadline de 900 milisegundos, dado que se usó para el ejercicio anterior un tiempo de expiración de 1000 milisegundos.
Para realizar la invocación de las 10 llamadas se creó en la subcarpeta **Entrega 3/1/grpc-hello-server 1C** un script de bash llamado script.sh el cual se encarga de ejecutar el proceso Server el background y luego ejecutar 10 procesos Client. De cada ejecución del proceso Client, se desvía la salida estándar de los procesos a un archivo llamado **experimento1c.txt**

Para este ejercicio se realizaron las siguientes pruebas:

1. Se estableció el deadline en un 10% menos que el ejercicio 1b, quedando el tiempo de expiración en 900 milisegundos.
2. Se calculó el tiempo promedio de 10 ejecuciones del proceso Cliente, dando un promedio de 152 milisegundos. Para este caso fallaron 3/10 ejecuciones, todos con motivo **DEADLINE_EXCEEDED.**
3. Se calculó el 10% menos del tiempo promedio obtenido (152*0.1) dando como resultado 15.2, por lo tanto quedó un deadline de 136 milisegundos aproximadamente. Para este caso fallaron 6/10 ejecuciones. Durante esta prueba se produjo un nuevo error: ***StreamError***, este mismo se generó debido a que el cliente canceló el RPC por motivo de expiración del deadline. Cuando se cancela un RPC, el servidor debe detener cualquier cálculo en curso y finalizar su lado de la transmisión. Para este error mencionado, el servidor recibe la información de un flujo que “olvido”, motivo por el cual se desencadena este error.
   ERROR: StreamError

```
oct. 22, 2023 5:00:25 P. M. io.grpc.netty.NettyServerHandler onStreamError
ADVERTENCIA: Stream Error
io.netty.handler.codec.http2.Http2Exception$StreamException: Received DATA frame for an unknown
stream 3
```

Una posible causa de que los clientes cancelen sus requerimientos es debido al scheduler del sistema operativo en conjunto a que se tiene un deadline muy ajustado. Si el sistema operativo atrasa alguna ejecución de algún cliente, es posible que se supere el deadline establecido, generando que el proceso cliente cancele el requerimiento.

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:

APIs de gRPC

- Unary RPC: RPC unarios en los cuales el cliente envía una única solicitud al servidor y recibe una única respuesta, similar a una llamada de función normal
- Server streaming RPC: RPC de transmisión desde el servidor en los cuales el cliente envía una solicitud al servidor y recibe un flujo o secuencia de mensajes de respuesta a dicha solicitud. gRPC garantiza el orden de los mensajes dentro de una llamada RPC individual.
- Client streaming RPC: RPC de transmisión desde el cliente en los cuales el cliente escribe una secuencia de mensajes y los envía al servidor, utilizando nuevamente un flujo proporcionado. Una vez que el cliente ha terminado de escribir los mensajes, espera a que el servidor los lea y devuelva su respuesta (que es un único mensaje). Una vez más, gRPC garantiza el orden de los mensajes dentro de una llamada RPC individual.
- Bidirectional streaming RPC: RPC de transmisión bidireccional en los cuales ambas partes envían una secuencia de mensajes utilizando un flujo de lectura y escritura. Los dos flujos operan de forma independiente, por lo que los clientes y servidores pueden leer y escribir en el orden que prefieran. Por ejemplo, el servidor podría esperar a recibir todos los mensajes del cliente antes de escribir sus respuestas, o podría alternar entre leer un mensaje y escribir un mensaje (“ping-pong”), o alguna otra combinación de lecturas y escrituras. El orden de los mensajes en cada flujo se conserva

a) Un sistema de pub/sub

Si pensamos en un sistema donde se tiene un servidor broker entre el publicador y el suscriptor, sería conveniente que el servidor por cada POST realizado por el publicador, informe al resto de suscriptores. Esta acción resulta conveniente que sea asíncrona ya que el orden no importa y no impacta en la experiencia del usuario con respecto al servicio. Por lo tanto en este caso la opción que convendría utilizar para el publicador, sería un modelo **Server Streaming RPC**.
Por otro lado, el publicador por cada publicación que realiza en el servidor, genera un envío de mensajes masivo a cada suscriptor desde el servidor. Cada publicación que hace el publicador es un mensaje que envía al servidor. Dado que estos mensajes pueden ser varios, convendría usar **Client Streaming**.
Con respecto a otras APIs:

- Bidirectional streaming RPC: usarla ocuparía recursos de los cuales la mayoría de los usuarios podrían no estar usando, afectando el ancho de banda y generando más pasajes de mensajes que podrían saturar la red.
- Unary RPC: para este caso, el envío de mensajes sería un proceso tedioso, caro computacionalmente y lento. El servidor tendría que

"recorrer" todos los usuarios para establecer una conexión con los mismos, sincronizar, enviar la publicación, cerrar la conexión y repetir el proceso.

Como ejemplo, podemos mencionar un sensor que envía constantemente datos a un servidor. El sensor en esta caso, está recopilando datos como la temperatura, viento, humedad, rayos uv, entre otros parámetros, por ende envía constantemente información al servidor para ser vista por los usuarios subscriptores.
En este caso se debería contar con un sistema centralizado, donde existan varias réplicas del servidor para balanceo de carga y que sea tolerante a fallos.

b) Un sistema de archivos FTP

En un sistema de archivos FTP, un usuario puede cargar y descargar archivos. Dado que en FTP, para la carga y descarga se tienen dos canales, donde uno es para transferencia de datos y otro es de control, la elección trivial sería un módelo **Bidirectional streaming RPC.**
En el modo de control, dado que la conexión es punto a punto y el flujo de datos es pequeño, además que se utilizará simplemente un enlace, resulta más conveniente utilizar un modelo de **Unary RPC**.
Con respecto a **Client/Server streaming RPC** utilizar esta opción sería lo mismo que poner un enlace punto a punto. Para este caso, no se aprovecharían las características de envío masivo de mensajes, ya que por cada comando se envía 1 solo mensaje y se recibe un solo mensaje.
Para este modelo, es necesario controlar qué datos llegarán y cuáles no, para poder solicitar un reenvío en caso de requerirse. Por lo tanto es necesario que la recepción sea sincrónica.
Finalmente, dependiendo del uso se pueden optar por:
Tener una red de computadoras que comparten archivos de forma local, o en el caso de que el acceso sea remoto tener un punto de acceso donde se verifique la identidad y se le permita el acceso a la red. Para este último caso se deberá tener políticas de seguridad y filtros debido a que cualquier falla en la seguridad implicaría alguna filtración de datos de por ejemplo una empresa o incluso introducción de código malicioso.

c) Un sistema de chat

Para este modelo, los mensajes llegan al receptor cuando pueden ser recibidos, encontrándonos ante un sistema asíncrono. En este caso dado las características antes mencionadas, el modelo más apropiado es un modelo **Client/Server streaming RPC**, donde los clientes envían una oleada de mensajes al servidor, el servidor los guarda y luego reenvía la oleada de mensajes al usuario destino. Cabe destacar que también es necesario que todos los mensajes enviados lleguen al usuario destino sin que se pierda ninguno, ya que esto afectaría la calidad del servicio.
Para este caso sería necesario contar con un gran sistema centralizado donde haya servidores replicados para balancear la carga. Las réplicas

permitirían contar con alta disponibilidad ante la eventual falla de uno de los servidores. Finalmente, también debería haber un servidor de *Storage* que permita almacenar los mensajes una cierta cantidad de tiempo configurable.

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.

Un elemento característico de la comunicación entre procesos mediante gRPC es el principio de transparencia: la colaboración entre instancias (en parte muy) distanciadas es tan estrecha y sencilla que no se percibe ninguna diferencia en comparación con una comunicación local entre procesos internos de una máquina.

En cuanto al manejo de parámetros, los Protocol Buffers (Protobuf) cumplen varias funciones en el sistema gRPC: sirven como lenguaje de descripción de interfaz (Interface Definition Language, IDL) y describen una interfaz de manera independiente de cualquier lenguaje, es decir, no están vinculados a un lenguaje de programación específico. También definen los servicios que se deben emplear y las funciones disponibles. Para cada función, puede indicarse qué parámetros se envían con una consulta y qué valor de respuesta se puede esperar.

Por lo tanto, en el caso de los valores de retorno, estos son definidos por los Protocol Buffers y son transparentes para el cliente y el servidor. Esto significa que el cliente puede esperar un valor de retorno específico para cada llamada a procedimiento remoto.

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.
    a. Si el archivo existe, los datos se agregan al final,
    b. 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.

Para implementar un sistema remoto de archivos remotos mediante el uso de la biblioteca de gRPC se realizaron las siguientes tareas

1. Se definió el servicio FTP y las estructuras a usar en el archivo FTPService.proto, mediante el uso de Protocol Buffers. En este archivo se definieron:
    a. Messages
        i. **ReadRequest** con los campos **name** para el nombre del archivo, **position** para indicar la posición inicial de lectura en el archivo, y **amount** para indicar la cantidad de bytes a leer.
        ii. **ReadResponse** con los campos **content** para almacenar el contenido leído del archivo, **requestedReadBytes** que almacena la cantidad de bytes que solicito el proceso cliente leer, y **readBytes** que indica la cantidad de bytes efectivamente leídos
        iii. **WriteRequest** con los campos **name** para el nombre del archivo, **amount** para indicar la cantidad de bytes a escribir y el campo **buffer** que contiene los datos a escribir en el archivo.
        iv. **WriteResponse** con el campo **writtenBytes** el cual indica la cantidad de bytes que se escribieron.
    b. Service
        i. **FtpService** con los siguientes métodos RPC unarios:
            1. write
            2. read
2. Se creó el archivo **FTPServiceImpl.java** en el cual se define la clase **FTPServiceImpl** la cual contiene la lógica de lectura y escritura de archivos, utilizando las clases generadas gracias a las estructuras definidas en el archivo **FTPService.proto**. En esta clase se definieron dos métodos:
    a. **public void read(FTPService.ReadRequest request, StreamObserver<FTPService.ReadResponse> responseObserver)**. Este método es el encargado de abrir el archivo solicitado, posicionarse dentro del archivo en la posición indicada y leer la cantidad de bytes indicados por el proceso Client, y finalmente establecer como respuesta el contenido leído, la cantidad que solicitó el proceso Client leer, y la cantidad de bytes efectivamente leídos.
    b. **public void write(FTPService.WriteRequest request, StreamObserver<FTPService.WriteResponse> responseObserver).** Este método es el encargado de abrir o crear un nuevo archivo, escribir los datos solicitados por el proceso Client y finalmente establecer como respuesta la cantidad de bytes efectivamente escritos en el archivo
3. Se creo el archivo Client.java el cual consta de una clase Client la cual se encarga de los siguientes métodos:
    a. **public static void main(String[] args) throws Exception** el cual realiza las siguientes tareas:

i. Valida que se hayan recibido los argumentos necesarios para realizar las operaciones de lectura o escritura.
ii. Establece el canal.
iii. Determina si se solicitó una operación de lectura o una operación de escritura, caso contrario informa que la operación ingresada no es una operación permitida.
iv. Invoca el método correspondiente a la operación ingresada.

b. **public static void read(FtpServiceGrpc.FtpServiceBlockingStub stub, String name, long position, int amount)** el cual realiza las siguientes tareas:
   i. Recibe como parámetro los datos necesarios para generar un ***ReadRequest***.
   ii. Construye un ***ReadRequest*** configurando el nombre, "*position*" en la cual se envía una posición en el archivo y por último una cantidad de bytes a leer.
   iii. Realiza la petición y se queda esperando por la respuesta
   iv. Imprime en la salida estándar la respuesta.
c. **public static void write(FtpServiceGrpc.FtpServiceBlockingStub stub, String name, int amount, byte[] buffer)** el cual realiza las siguientes tareas:
   i. Recibe como parámetro los datos necesarios para generar un ***WriteRequest***.
   ii. Construye un ***WriteRequest*** configurando el nombre, una cantidad de datos a escribir, y un buffer que es el archivo enviado para ser escrito en el servidor.
   iii. Realiza la petición y se queda esperando por la respuesta
   iv. Imprime en la salida estándar la respuesta.

4. Se definió el archivo App.java el cual define la clase App que corresponde al proceso servidor. En esta clase se encuentra únicamente un método public **static void main(String[] args)throws Exception.** En este método se crea una instancia de server el cual agrega una implementación del servicio **FTPServiceImpl** al registro del controlador**,** inicia el servidor y espera la llamada de clientes.

Para iniciar los procesos Server y client se utilizaron los siguientes comandos:

- **Server**

```
mvn -DskipTests package exec:java -Dexec.mainClass=pdytr.ftp.grpc.App
```

- **Client (para escritura)**

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.ftp.grpc.Client
-Dexec.args="MODE DEST SIZE CONTENT"
```

ejemplo:

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.ftp.grpc.Client
-Dexec.args="w ./db/hola.txt 4 'hola que tal'"
```

- **Client (para lectura)**

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.ftp.grpc.Client
-Dexec.args="MODE SOURCE POSITION AMOUNT"
```

ejemplo:

```
mvn -DskipTests exec:java -Dexec.mainClass=pdytr.ftp.grpc.Client
-Dexec.args="r ./db/hola.txt 0 100"
```

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

Es posible tener varias invocaciones remotas ya que la API lo permite por defecto, pero, si bien las permite, no implica que sean seguras, es decir, la aplicación debe tener un manejo adecuado de los hilos y recursos para mantener la coherencia.
Para el caso de un servidor de archivos la solución concurrente que provee por defecto la API, no es segura, dado que se producirían inconsistencias si dos o más procesos intentan acceder a tal recurso. Para demostrar este hecho, se realizó un experimento en el cual dos procesos clientes, invocan a un proceso servidor e intentan ambos, realizar operaciones de escritura sobre el mismo archivo de manera concurrente. Para esto se realizaron las pruebas sobre una computadora con sistema operativo Linux/Ubuntu siguiendo los siguientes pasos:

1. Se crearon 2 archivos de texto plano llamados ClienteA.txt el cual contiene letras A, y el archivo ClienteB.txt el cual tiene escrito letras B.
2. Se creó un script de bash el cual:
    a. Inicia el proceso servidor en modo background.
    b. Se inicia el proceso client 1 en modo background, el cual escribe sobre un archivo llamado exprimentoB.txt el contenido del archivo ClienteA.txt.
    c. Se inicia el proceso client 2 en modo background, el cual escribe sobre un archivo llamado exprimentoB.txt el contenido del archivo ClienteB.txt.
    d. Se espera la finalización de los procesos cliente

e. Se mata al proceso servidor.

Una vez finalizados los procesos se puede observar en el archivo exprimentoB.txt, que el resultado no es el esperado, sino que solo se encuentra escrito el contenido de uno de los procesos cliente. Esto se produjo dado que al no haber un mecanismo de exclusión mutua, cuando ambos procesos empezaron a escribir el archivo experimentoB.txt, el puntero del mismo se encontraba al inicio del archivo, obteniendo ambos procesos la misma posición de inicio de escritura, por lo tanto cuando el último proceso en ejecutarse culmina la escritura, en vez de escribir al final del archivo, superpone su escritura a la escritura del proceso que finalizó primero, perdiéndose los datos del primer proceso en escribir el archivo experimentoB.txt.
En los resultados obtenidos de la ejecución, se observa que el archivo experimentoB.txt solo tiene letras B, habiéndo perdido (sobreescrito) la escritura del primer proceso, que escribió letras A.

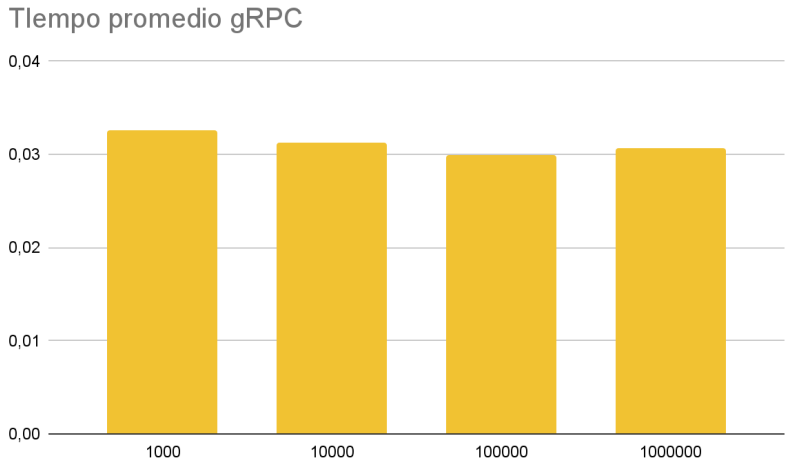
Una posible solución al problema de inconsistencia de datos, sería utilizar algún mecanismo de exclusión mutua sobre los archivos, únicamente para los procesos que intentan realizar operaciones de escritura, de modo que si un proceso intenta realizar una escritura sobre un archivo primero tendrá que validar si dicho archivo está disponible, una vez que el recurso esté disponible deberá bloquearlo, luego realizará la operación de escritura y una vez que finalice, liberará el bloqueo sobre el archivo, para que otro proceso pueda escribir en caso de que fuese requerido.

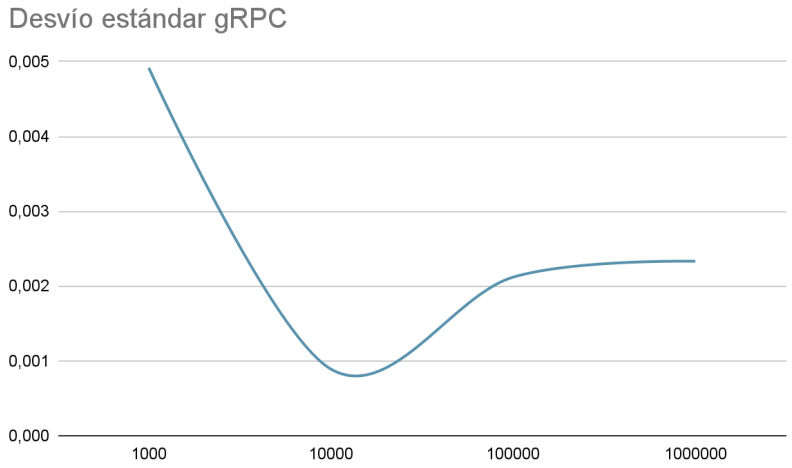
5) Tiempos de respuesta de una invocación

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 calcular el tiempo de comunicación se utilizó el ejemplo provisto por la cátedra gRPC-hello-server. Con este ejemplo se diseñó un script de bash que calcula 10 muestras de tiempo de comunicación del lado del cliente para para 10^3 datos, diez muestras para 10^4 datos, diez muestras para 10^5 datos y diez muestras para 10^6 datos.
Una vez obtenido estos tiempos, se calculó en una hoja de cálculo, el tiempo de comunicación individual dividiendo el total obtenido entre 2, ya que el proceso realiza 2 comunicaciones. Una vez obtenido los tiempos individuales se realizó el cálculo de promedio y el cálculo de la desviación estándar.

Los siguientes gráficos muestran los valores obtenidos:





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.

De la comparación realizada a partir de los promedios obtenidos para los procesos realizados con gRPC y la biblioteca de Sockets, se observa que gRPC resulta considerablemente más lento respecto al tiempo de comunicación para transferencia de datos. Una de las posibles razones por la cual gRPC es menos eficiente puede ser la cantidad de niveles de abstracción que esta herramienta posee, así como la mayor cantidad de código generado, que debe ser interpretado en tiempo de ejecución en contraposición con una solución muy simple realizada a partir de Sockets. Por otro lado, cabe destacar que gRPC utiliza stubs para realizar la comunicación, serializar y deserializar los datos, siendo esta comunicación de más alto nivel que en la solución de Sockets, donde se realiza prácticamente sobre el mismo *"Channel"*, así como validación de integridad de datos que son abstractas al desarrollador.

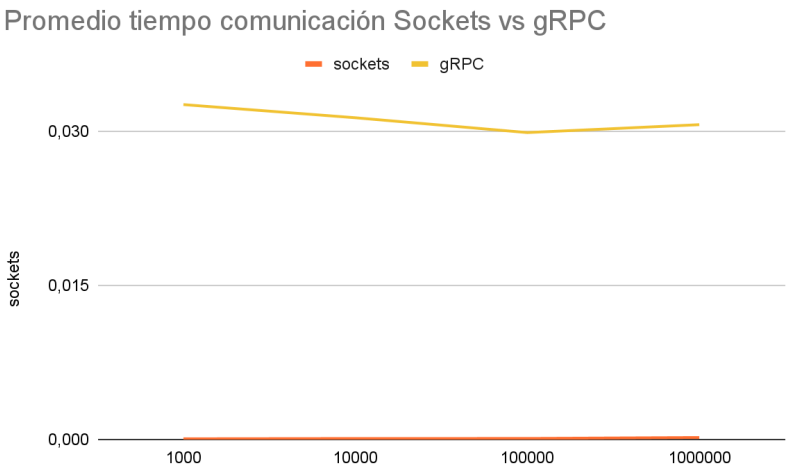
El siguiente cuadro expone una comparación entre gRPC y Sockets para algunas de las características que consideramos más destacables:

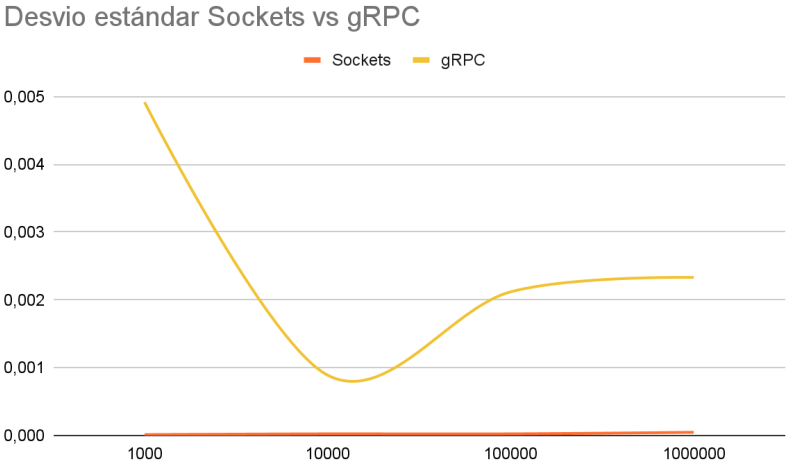
Características	gRPC	Sockets
Abstracción de la Comunicación	Proporciona una abstracción de comunicación de nivel superior y se basa en HTTP/2. Define servicios y mensajes utilizando gRPC Protocol Buffers (protobuf) y genera código cliente y servidor a partir de esas definiciones, facilitando la definición de APIs y la generación de código en múltiples lenguajes	Ofrece un control más directo sobre la comunicación de red. Se deben manejar los detalles de la conexión, lectura y escritura de datos de forma manual.
Protocolo de Comunicación	Utiliza Protocol Buffers para definir los mensajes y las interfaces de servicio. Ofrece una forma eficiente y optimizada de serialización de datos y es independiente del lenguaje	Se implementa un protocolo de mensajes de forma manual. Brinda flexibilidad.
Rendimiento	Hace uso de HTTP/2, que es un protocolo de red altamente eficiente que admite la multiplexación de varias solicitudes en una sola conexión, lo que puede proporcionar un mejor rendimiento en comparación con HTTP/1 y protocolos personalizados	Brinda un control más directo sobre la comunicación, lo que permite optimizarla según las necesidades específicas del dominio en el cual se trabaja.
Facilidad de Uso	Proporciona una abstracción más alta y una forma más sencilla de definir y consumir servicios en comparación con Java Sockets. También genera código cliente y servidor de manera automática a partir de las definiciones de servicio.	Requieren más trabajo manual y pueden ser más complejos de implementar y mantener.
Lenguajes	Admite una variedad de lenguajes de programación, lo que facilita la comunicación entre aplicaciones escritas en diferentes lenguajes.	Biblioteca que puede ser utilizada para comunicarse con otras aplicaciones Java.
Seguridad	Ofrece características integradas de seguridad, incluyendo autenticación y cifrado de datos.	La seguridad debe ser gestionada manualmente si es necesaria.
Tamaño máximo de mensaje	4KB	Integer.MAX_VALUE.

Conclusión

Aunque gRPC resultó práctico en la generación del código ahorrando tiempo de desarrollo, resultó para los alcances de este trabajo más ineficiente un proceso java usando Sockets. Asimismo, aunque Sockets presenta una buena flexibilidad y adaptabilidad al entorno en cuanto a la eficiencia, resulta menos práctica la tarea de definición y desarrollo.
Respecto a los protocolos, gRPC brinda una solución sencilla y transparente para las declaraciones de cada uno de sus servicios entre cliente y servidor, con el uso único de Sockets, se deben establecer manualmente. Por otro lado, gRPC es una solución más moderna que utiliza el protocolo http 2 mientras que sockets ofrece mejor control a bajo nivel sobre TCP/IP.
Para realizar pruebas cabe destacar que gRPC puede resultar más difícil de debugear.
Finalmente, pese a que los tiempos de gRPC son sumamente mayores a los obtenidos sobre Sockets, la solución de gRPC es mucho más robusta y completa.

Los siguientes gráficos expresan las comparaciones de los tiempos promedio de comunicacion individual de cada solución así como el desvío estándar de las mismas:





Referencias

Punto 1C

- https://grpc.io/docs/guides/cancellation/

Punto 2

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

Punto 3

- gRPC | Así funciona el procedimiento - IONOS
- Descripción de OpenConfig y gRPC en la interfaz de telemetría de Junos | Junos OS | Juniper Networks
- Microsoft PowerPoint - RPC_Todo.ppt [Modo de compatibilidad] (cinvestav.mx)

Punto 5

- https://github.com/Luminicen/PDYRT/blob/main/Entrega%203/5/Calculos%20Tiempo%20punto%205a%20-%20Hoja%201.pdf