

Programación Distribuida y Tiempo Real

Práctica 3

AÑO 2022 – 2º Semestre

Referencias

1. <https://www.paradigmadigital.com/dev/grpc-que-es-como-funciona/>
2. <https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>
3. <https://mqtt.org/getting-started/>
4. <https://www.rfc-editor.org/rfc/rfc959>
5. <https://learn.microsoft.com/en-us/aspnet/core/grpc/security?view=aspnetcore-6.0>
- 6.

1) Utilizar como base el programa ejemplo 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.

- Petición del cliente sin servidor activo

El primer experimento consistió en iniciar la conectividad del lado del cliente sin haber iniciado el servidor, para que este conteste a las peticiones. Por lo tanto, este experimento se llevó a cabo sin realizar ningún cambio al código brindado por la cátedra.

gRPC responde a este escenario mediante la excepción `io.grpc.StatusRuntimeException: UNAVAILABLE`.

```
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)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:279)
    at java.lang.Thread.run (Thread.java:829)
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Conexión rehusada: localhost/127.0.0.1:8080
    at sun.nio.ch.SocketChannelImpl.checkConnect (Native Method)
    at sun.nio.ch.SocketChannelImpl.finishConnect (SocketChannelImpl.java:777)
```

Excepción `io.grpc.StatusRunTimeException: UNAVAILABLE`

- Cierre del servidor en el transcurso de la comunicación

El segundo experimento consiste en finalizar el proceso del servidor abruptamente en el momento donde este se encuentre llevando a cabo la gestión de alguna solicitud proveniente de un proceso cliente.

Para producir este escenario, se implementa la función `System.exit(1)` sobre la implementación del servidor, para que este finalice una vez que este reciba una petición por parte de algún cliente.

gRPC en este caso responde mediante la excepción `io.grpc.StatusRuntimeException: UNAVAILABLE: Network closed for unknown reason`.

```
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
[WARNING]
io.grpc.StatusRuntimeException: UNAVAILABLE: Network closed for unknown reason
    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:29)
```

Excepción `io.grpc.StatusRunTimeException: UNAVAILABLE: Network closed for unknown reason`

- Cierre del cliente en el transcurso de la comunicación

El último experimento consistió en provocar el proceso del cliente abruptamente en el momento donde se encuentre esperando la respuesta a la petición que le realizó al proceso servidor.

Para el desarrollo de este escenario, se implementa del lado del cliente una clase Thread que es ejecutada cuando se encuentra en comunicación con el servidor.

La funcionalidad de esta clase Thread es permitir dormir el proceso por 0.4 segundos mediante la función sleep y posteriormente finalizar la ejecución del proceso mediante la función System.exit(1).

Una vez finalizada la ejecución de diferentes pruebas del experimento, se observa que presenta diferentes desenlaces:

Las primeras dos observaciones se hacen visibles en el momento donde el servidor ya resolvió la petición realizada por el cliente y se encuentra comunicando la respuesta a la petición. En algunos casos el cliente recibe dicha respuesta antes de finalizar con su ejecución (escenario estándar de la comunicación) y en otros casos el servidor envía la respuesta a un cliente cuya ejecución se encuentra finalizada.

```
[INFO] OS detected: classifer: linux-x86_64
[INFO] -----< pdytr.example.grpc:grpc-hello-server >-----
[INFO] Building grpc-hello-server 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
nahuel@nahuel-VirtualBox:~/Escritorio/repositorio/pdytr-grpc-demo/grpc-hello-ser
```

Caso 1: Cliente no recibe respuesta

El servidor manda información a un cliente finalizado. (Por lo que el cliente nunca recibe el mensaje)

```
[INFO] Building grpc-hello-server 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
greeting: "Hello there, Ray"
nahuel@nahuel-VirtualBox:~/Escritorio/repositorio/pdytr-grpc-demo/grpc-hello-ser
```

Caso 2: Cliente recibe respuesta

El servidor manda información a un cliente que aún no finalizó. (Por lo que el cliente recibe el mensaje antes de finalizar)

Una tercera observación (presentada en menor medida que las anteriores) es donde el servidor presentaba un error en el momento de recibir la petición por parte del cliente.

```
name: "Ray"
name: "Ray"
oct. 10, 2022 1:39:53 A. M. io.grpc.netty.NettyServerTransport notifyTerminated
INFORMACIÓN: Transport failed
java.io.IOException: Conexión reinicializada por la máquina remota
    at java.base/sun.nio.ch.FileDispatcherImpl.read0(Native Method)
    at java.base/sun.nio.ch.SocketDispatcher.read(SocketDispatcher.java:39)
    at java.base/sun.nio.ch.IOUtil.readIntoNativeBuffer(IOUtil.java:276)
    at java.base/sun.nio.ch.IOUtil.read(IOUtil.java:233)
    at java.base/sun.nio.ch.IOUtil.read(IOUtil.java:223)
    at java.base/sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:356)
    at io.netty.buffer.PooledUnsafeDirectByteBuf.setBytes(PooledUnsafeDirectB
    at io.netty.buffer.PooledUnsafeDirectByteBuf.setBytes(PooledUnsafeDirectB
```

Caso 3: Error de parte del servidor

Error presentado del lado del servidor al intentar una tercera comunicación.

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

Para producir este escenario, se modifica el código brindado agregando del lado del servidor la función sleep() para que el proceso se detenga durante 10 segundos, para posteriormente continuar con su ejecución.

Del lado del cliente se define un deadline, que permite indicar al proceso el tiempo que debe emplear en esperar una respuesta por parte del servidor, en este caso se indicó que espere tan solo 5 segundos.

Se implementa el deadline utilizando la función withDeadlineAfter(arg1, arg2) donde el primer argumento indica el tiempo a esperar, y el segundo argumento indica la unidad empleada para interpretar el tiempo definido.

Con los tiempos brindados tanto al cliente como al servidor, provocan que el tiempo de espera del cliente se agote antes de que el servidor sea capaz de devolver una respuesta

Esto provoca que gRPC responda mediante la excepción: DEADLINE_EXCEEDED: deadline exceeded after...

```
I[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
[WARNING]
io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline exceeded after 4963279052ns
    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:124)
    at pdytr.example.grpc.Client.main (Client.java:29)
```

Excepción io.grpc.StatusRunTimeException: DEADLINE_EXCEEDED

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

Se utiliza el código implementado para el ejercicio 1b, se mantiene el cliente en la espera de una respuesta por 5 segundos, pero se modifica el tiempo en el que se duerme el proceso del servidor con la función sleep(), pasando a ser en este caso de 4.5 segundos. Dejando un margen de respuesta de tan solo 0.5 segundos para que el servidor responda a la petición del cliente.

En base a ese escenario, se evalúa los resultados obtenidos empleando las peticiones de diferentes clientes. Se observa que en ciertas ocasiones se logra satisfactoriamente las comunicaciones mientras que en otras se demora demasiado tiempo, por lo que provoca el exceso de tiempo definido por el deadline.

Al realizar en diferentes tandas la prueba de iniciar un servidor y enviar un total de 10 peticiones solicitadas de los clientes. Se observa que siempre en la primera comunicación se produce el fallo provocado por el DEADLINE definido, pero que en las siguientes 9 peticiones realizadas de la tanda no presentan ningún inconveniente, terminando la comunicación de manera satisfactoria.

Por lo que se llega a la conclusión de que, una vez iniciado el servidor, éste requiere de más tiempo de cómputo de lo normal (a comparación de las posteriores comunicaciones) debido a que debe cargar en memoria principal todos los componentes necesarios para su correcta funcionalidad. Y lo que provoca que en posteriores comunicaciones, aumente su velocidad al ya poseer en memoria lo necesario.

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.

Gracias a la multiplexación de HTTP/2, gRPC es capaz de brindarnos 4 tipos de APIs con la capacidad de procesar información en Streaming de diferentes formas.

Unary: El cliente envía una única solicitud y el servidor devuelve una única respuesta.

Server streaming: El cliente envía una única solicitud y el servidor puede retornar múltiples respuestas, es decir, un flujo de respuesta.

Client streaming: Donde el cliente envía múltiples peticiones y el servidor devuelve una única respuesta.

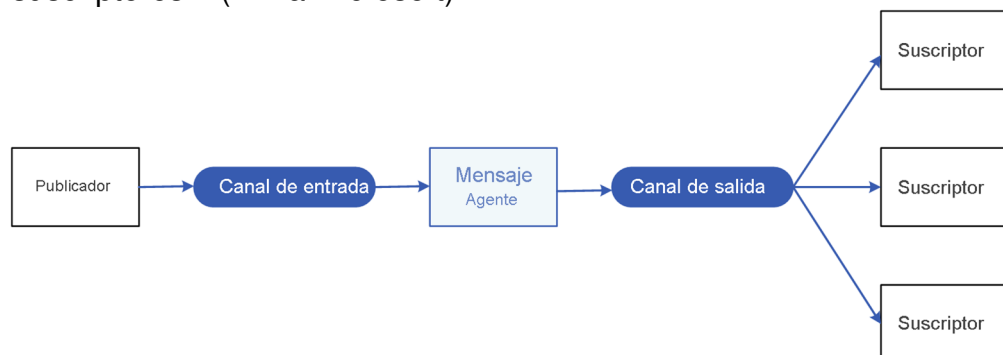
Bidirectional streaming: En la que tanto el cliente como el servidor envían mensajes entre sí de forma simultánea, es decir, sin esperar respuesta del otro lado.

a) Un sistema publicación/suscripción:

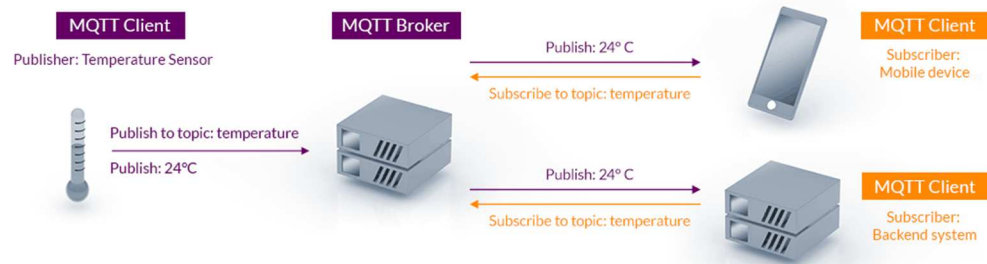
Estos sistemas hacen uso del patrón “publicador/suscriptor” el cual permite que un proceso anuncie eventos de forma asincrónica a varios procesos interesados en consumir esa información.

Este tipo de comunicación se usa en sistemas IoT en la nube. Para implementarlo se debe hacer un sistema de mensajería asincrónico.

Este tipo de servicio de mensajería desacopla al publicador de los suscriptores.² (link a Microsoft)

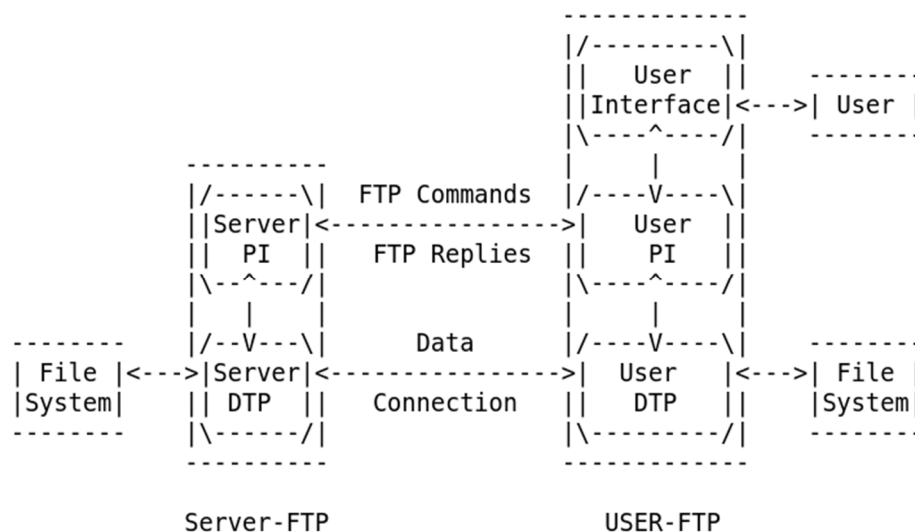


Un protocolo muy usado en IoT que usa este patrón es MQTT.3 ([link](#))



Si quisiéramos implementar una arquitectura similar en gRPC el cliente MQTT enviaría un streaming de datos al MQTT Broker, o sea usaría un streaming desde el cliente. Y en el caso de los nodos clientes que se suscriban a un tópico, en este caso temperatura, usarían un streaming desde el servidor, en donde un cliente suscrito a un tópico recibe varios eventos para ese tópico desde el servidor MQTT Broker, en este caso podría pasar que cada 15 min se evalúe un cambio de temperatura.

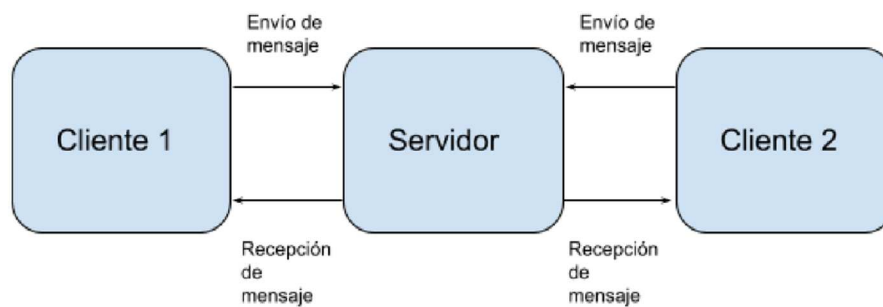
- b) Un sistema de archivos FTP permite la transferencia directa de archivos desde un cliente o varios clientes a un servidor FTP. El servidor puede proporcionar varias operaciones que involucran archivos como por ej.: subir un archivo, eliminar un archivo, descargar un archivo, etc. Como se puede observar no todas las operaciones involucran la transferencia de archivos, si un cliente desea eliminar un archivo en el servidor esto no requiere transferencia alguna, sino sólo enviar un simple comando.⁴ ([link rfc](#))



Para la implementación de este sistema puede usarse el tipo de API de gRPC unary si se trata de archivos pequeños o el envío de únicamente comandos de operación (borrar, listar, etc.) y respuestas que no involucren la transferencia de archivos. Esto se debe a que gRPC limita el tamaño de los mensajes, que por defecto es de 4MB.5

Sin embargo, si no sabemos el rango de tamaño de los archivos que vamos a transferir o si los archivos son grandes es conveniente considerar una transferencia vía streaming. Si los archivos son grandes se puede usar una transferencia vía streaming, desde el cliente o el servidor dependiendo del tipo de operación que se haga.

- c) Un sistema de chat básico, consiste principalmente de la transferencia arbitraria de mensajes entre clientes utilizando como intermediario la participación de un servidor, en el cual cada cliente le enviará mensajes sin esperar una respuesta por parte del mismo.



En base a este escenario, se consideró como mejor opción la utilización del tipo de API gRPC Bidirectional streaming, ya que nos permite realizar el envío de varios mensajes por parte del cliente emisor hacia el servidor, y desde el servidor hacia el cliente receptor. Es decir, que, en este escenario, el servidor funcionará como un mailbox de los mensajes enviados.

- 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.**

gRPC utiliza el Protocol Buffers como forma de serialización de los mensajes implementados. Este mecanismo utiliza una estructura que define datos y servicios a brindar declaradas en un archivo con extensión

proto. Dentro de esta estructura se pueden declarar el tipo, las reglas o IDs necesarios para los datos.

Por lo que tanto el cliente como el servidor conocen las peticiones con la cantidad y tipos de parámetros que las conforman antes de utilizarlas, como también los datos a recibir. Lo que permite utilizar esta metodología como herramienta para la comunicación y transferencia de información entre distintos procesos.

Por lo tanto, gRPC brinda transparencia al usuario a la hora de utilizar los mensajes, más los parámetros requeridos en cada uno de dichos mensajes para realizar las comunicaciones, y brindando la información para saber de antemano, los datos que retorna cada mensaje, gracias a la estructura previamente definida en el archivo proto.

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.
 - 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.

a)

Para la implementación de un sistema de transferencia minimalista de archivos que comuniquen a un proceso servidor, con varios procesos clientes, se opta por trabajar guiándose de la estructura del código brindado por la cátedra.

Se decidió por modificar la estructura del archivo proto llamado ahora "FileTransferService.proto" donde se indicaron dos servicios llamados (ReadRequest y ReadResponse) para la lectura de archivos y (WriteRequest y WriteResponse) para la escritura sobre el servidor.

En base a esos servicios, el servidor cuenta con los métodos read() y write() que serán utilizados en el momento que los procesos clientes presenten peticiones a sus servicios.

Para iniciar la ejecución del servidor, se realiza de la siguiente forma:

```
$ mvn -DskipTests package exec:java -  
Dexec.mainClass=ptytr.example.grpc.App
```

Para iniciar la ejecución de un proceso cliente, se realiza de la siguiente forma:

```
$ mvn -DskipTests exec:java -  
Dexec.mainClass=ptytr.example.grpc.Client -  
Dexec.args="Operation File_name"
```

Donde la principal diferencia es la presencia de argumentos necesarios para su correcta funcionalidad

El argumento "Operation" indica la operación que realizará el proceso, siendo estas la de escritura "-w" y lectura "-r".

El argumento "File_name", como su nombre lo indica, requiere el nombre del archivo que se desea escribir o leer, según sea la operación previamente seleccionada.

Para la simulación del sistema de transferencia de archivos, se optó por tener dos directorios, el primero llamado "RepositoryServer" que representa el repositorio remoto del servidor, espacio que utilizarán los clientes para realizar las operaciones. El segundo llamado "ClientLocalSpace", que representa el espacio local del cliente.

b)

En el escenario donde se estén comunicando concurrentemente varios procesos clientes con el servidor. Se presentan inconvenientes dependiendo de la operación a realizar.

En el caso donde varios procesos clientes estén realizando la operación de lectura, no se presentaría ningún inconveniente, ya que cada uno de los clientes buscarán leer el archivo que se encuentra en el repositorio del servidor, sin realizar modificaciones.

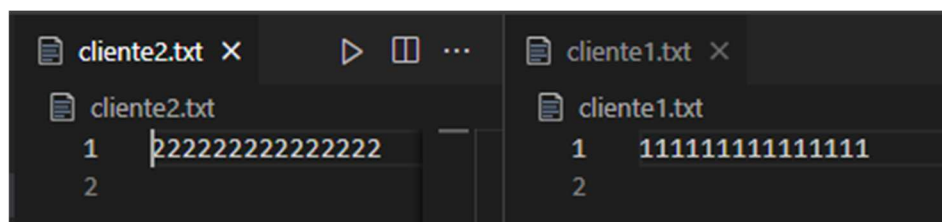
Sin embargo, en el caso donde se realice la operación de escritura de forma concurrente por varios clientes sobre un mismo archivo, este puede generar inconsistencia en su información, si no se trata la escritura de forma correcta.

Por lo tanto, poniendo énfasis en la operación de escritura, la mejor forma de tratar esta operación en un ambiente concurrente, sería tratar al archivo seleccionado por cliente como una sección crítica en donde al realizar la escritura, se realizará de a un cliente por vez, dejando a los demás procesos clientes que quieras modificar dicho archivo, en una cola de espera, pendientes de que la sección crítica sea liberada. Logrando de esta forma, la consistencia al realizar la operación.

Experimento de concurrencia:

Para visualizar la situación donde varios clientes escriben sobre un mismo archivo del servidor, se realizaron los siguientes pasos:

Se crean dos archivos en el directorio “ClientLocalSpace” con el siguiente contenido:



```
cliente2.txt x  cliente1.txt x
cliente2.txt    cliente1.txt
1  22222222222222  1  11111111111111
2                     2
```

Contenido de los archivos

Se modifica el código, exactamente en el método write() del cliente, donde la ruta de destino sea siempre el mismo archivo independientemente del archivo con el que se opere para, de esta forma, lograr la escritura concurrente sobre el mismo archivo destino. Y además se cambia el buffer_size a 2.

Mientras se encuentre el servidor operando, se inician concurrentemente la ejecución de dos clientes mediante un script.

Una vez realizado el experimento en 5 ocasiones, los resultados son los siguientes:

```
salida.txt
1  Primera prueba
2  221122222222221122
3
4  Segunda prueba
5  22222222222222
6  11
7
8  Tercera prueba
9  111111111111221
10
11 Cuarta prueba
12 1122111111112211
13
14 Quinta prueba
15 11111111111111
16 22
17
```

Resultados de escritura concurrente sobre un archivo

5) Tiempos de respuesta de una invocación

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

- a) Para este experimento construí un programa que envía un string y recibe un string, entre cliente y servidor respectivamente. La transferencia se repite 100 veces para tomar el tiempo mínimo, el promedio y la desviación estándar del elapsed time.

tiempo mínimo = 0.0015 s

tiempo promedio = 0.00502 s

desviación estándar = 0.02534215065853725

Los cálculos estadísticos se han hecho con librerías de Python, estos se pueden ver en el notebook de [este link](#).

b) Sockets en C vs GRPC en Java

Para este experimento utilicé el programa hecho en el ejercicio 1 de la práctica 2, pero en la misma máquina ejecuté el cliente y el servidor. No usé el programa de la práctica uno porque no mide el elapsed time y no mide los tiempos en segundos.

El programa en Java que usa gRPC hace lo mismo, envía arreglos de bytes de tamaños 10, 100, etc. bytes. Repite cada transferencia 200 veces para sacar el promedio del elapsed time.

Tabla de tiempos con sockets en C

cantidad de bytes	10 ¹	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶
tiempo en segundos	1.38725e-05	1.73e-05	1.7135e-05	3.696e-05	0.0002731575	0.001496395

Tabla de tiempos con gRPC en Java

cantidad de bytes	10 ¹	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶
tiempo en segundos	0.003424999999999914	0.00146000000000001	0.001357500000000009	0.001217500000000001	0.001367500000000009	0.00431499999999998

Medidas estadísticas

	sockets en C	gRPC en Java
media	0.0003091366666666667	0.002190416666666664
desviación estándar	0.0005389142680497881	0.0012171734817692215

Gráfico de tiempos de sockets en C

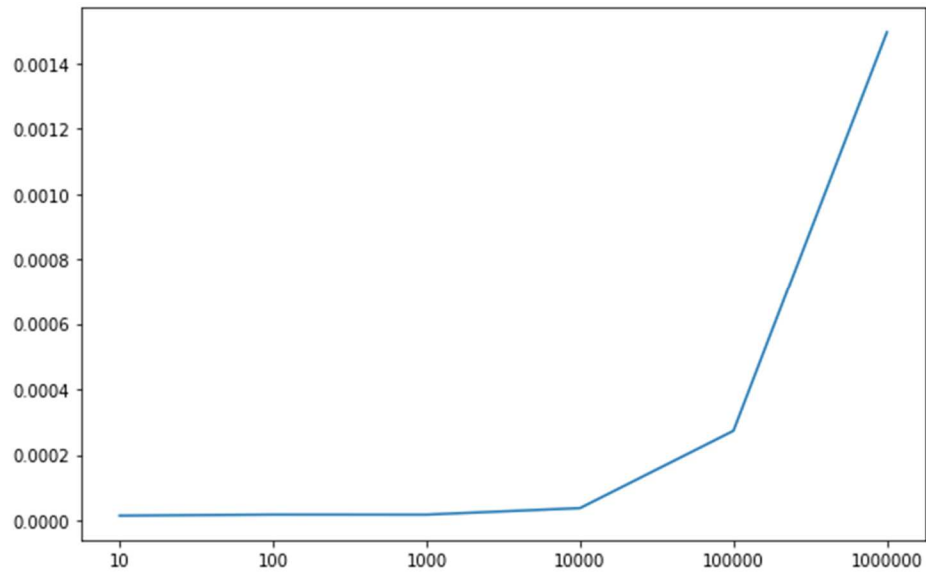
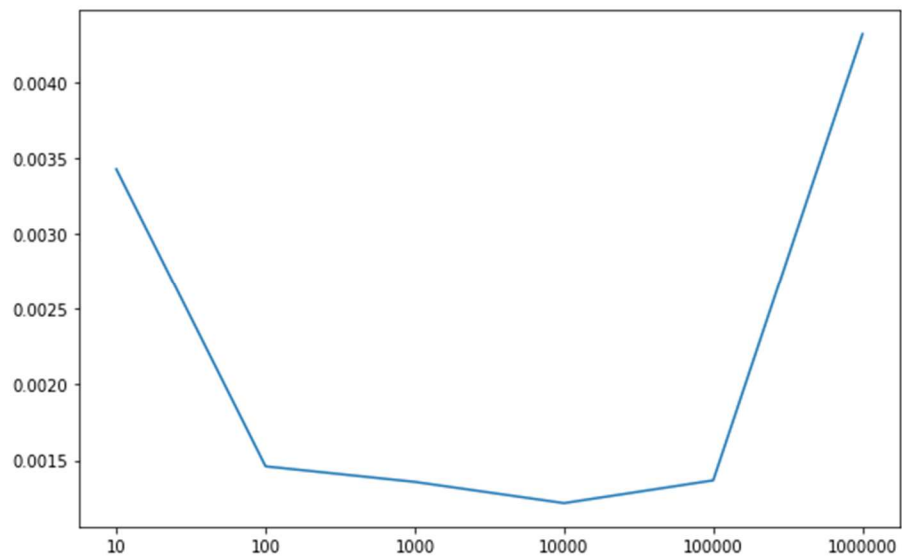


Gráfico de tiempos de gRPC en Java



Los resultados gráficos de la comparativa, junto con los cálculos estadísticos los hemos colgado en un notebook de Google ([link](#)).

La conclusión que hemos sacado sobre la comparativa de tiempos es que claramente gRPC en Java tarda mucho más tiempo en transferir datos que usar sockets en C, en algunos casos tarda hasta 100 veces más.

Sin embargo, la programación de un sistema cliente-servidor para el intercambio de datos nos pareció mucho más simple y abstracto en gRPC. Otra clara ventaja que observamos en gRPC es la alta

portabilidad ya que el código es 100% portable a diferentes lenguajes de programación. El programa escrito en sockets de C no es directamente portable a otro lenguaje como por ej. Java, sino que hay que re-escribirlo manualmente.